

# Dual Study of Canvas Fingerprinting based Authentication: A Novel Spoofing Attack and the Countermeasure

Zengrui Liu  
Texas A&M University  
lzh@tamu.edu

Nitesh Saxena  
Texas A&M University  
nsaxena@tamu.edu

**Abstract**—Browser fingerprinting is a tracking technique used to distinguish individual users. By leveraging unique fingerprint features or combining multiple ones, websites can not only identify users but also monitor their online activities. A specific aspect of browser fingerprinting, known as canvas fingerprinting, generates distinct values based on the characteristics of users' devices. This unique trait of canvas browser fingerprinting can be employed in challenge-response authentication, enabling user verification without requiring additional actions and potentially replacing the need for two-factor authentication. Furthermore, canvas fingerprinting can serve as an alternative to cookies, facilitating functionalities like the “Remember me” feature.

This paper introduces an implementation of man-in-the-middle attack called “CRSlash” that targets prevalent challenge-response authentication methods, with a particular focus on canvas fingerprinting based challenge-response authentication. In the case of CRSlash, an attacker only needs to obtain a challenge from the targeted device once. Subsequently, they can successfully navigate the authentication process. Our investigation reveals that existing challenge-response authentication methods relying on canvas fingerprinting are vulnerable to this attack. This vulnerability persists in both one-time authentication scenarios and continuous authentication setups. The outcomes of the attack demonstrate that prior canvas authentication methods are inadequate in countering this new threat. In response to this security concern, we propose a novel approach to canvas fingerprinting-based challenge-response authentication, which we call “CanvasDict.”

In the CanvasDict process, the website creates a distinct authentication dictionary using the user's browser fingerprint during the registration phase. Later, during the login phase, the website selects random challenges from this dictionary for the authentication process. Through in-depth analysis, we ascertain the effectiveness of our approach in thwarting the aforementioned attack. Our evaluation encompasses two modes of CanvasDict, and the results underscore the success of CanvasDict in neutralizing the potential risks posed by the attack. This paper highlights the significance of browser fingerprinting, specifically focusing on canvas fingerprinting, as a means of user tracking and authentication. It sheds light on the vulnerabilities of existing challenge-response authentication methods and proposes an innovative solution to bolster security in the realm of canvas fingerprinting-based authentication.

## I. INTRODUCTION

Security protection during user login is a concern for both users and websites. Traditional protection focuses on the password method. Users only need to remember the password, and websites find its implementation easy to set up. However, relying solely on passwords is not secure. Many users may

reuse the same password across different websites. According to Das et al. [9], 43% of users reuse the same password across different websites. Subsequent research by Pearman et al. [19] gathered password information from 154 participants, revealing that each participant might use the same password on nearly 6 different websites. Therefore, other factors should be considered and incorporated into the login process.

Implementing two-factor authentication is a favorable choice for enhancing login security. Nonetheless, there are still risks and drawbacks associated with two-factor authentication. It typically relies on mobile devices such as phones or tokens to receive a digital code or push notification. Users might find themselves unable to log in if they lose or damage their device and have to reset passwords and set up two-factor authentication again [6]. Two-factor authentication can also be inconvenient, as users need to input the code from the mobile device or use an app during login [7]. Additionally, the time-limited nature of this authentication method can be a hassle [4]. If users opt for text message-based two-factor authentication, it exposes them to message-related attacks, presenting significant security issues. In such cases, there arises a need for a new technology that offers verification without requiring extra equipment, thus ensuring the security of user accounts.

Laperdrix et al. [13] introduced a novel authentication approach employing canvas fingerprinting. Canvas fingerprinting stands as one of the browser fingerprinting features. Most browser fingerprinting features consist of static values, like User-Agent and browser languages. These features might be the same for different devices using the same browser version. In contrast, the same canvas image can yield distinct values across different devices. Factors such as supported or unsupported fonts, hardware, and software play a role in shaping the final canvas fingerprinting value. However, the challenge-response authentication based on canvas fingerprinting introduced in this study is not secure.

In this paper, we investigate fingerprinting attack “CRSlash” against the existing canvas fingerprinting-based challenge-response authentication and propose a new canvas fingerprinting approach “CanvasDict” capable of thwarting such attacks. Our key contributions can be summarized as follows:

- 1) In our study, we introduce a pioneering browser finger-

printing attack known as CRSlash, which possesses the capability to overcome the previously resilient challenge-response authentication based on canvas fingerprinting. Unlike previous approaches, this attack liberates attackers from the requirement of having similar devices or configurations as their victims. Instead, they can replicate the victim’s device environment and produce identical canvas fingerprints using the same set of challenges. With just one instance of enticing the victim to access a website controlled by the attacker, the latter gains the ability to bypass the ongoing operation of the challenge-response authentication system. CRSlash essentially represents an implementation of a man-in-the-middle attack.

- 2) In our study, we introduce a novel canvas authentication technique named CanvasDict. This method boasts two distinct protective modes: “All in Once” and “Sequential”. During the user registration process, CanvasDict constructs a comprehensive dictionary of canvas fingerprints by selectively incorporating features related to random fonts. Subsequent to user registration, the “All in Once” mode randomly selects font-related features from the pre-created dictionary and transmits them to the user’s device, thereby generating canvas data for subsequent comparison. On the other hand, the “Sequential” mode requires the generated data to be verified against dictionary pairs. Any mismatch between the pairs instantly leads to a failure in authentication.
- 3) Our study comprehensively evaluates the efficacy of the attack we devised against CanvasDict. We meticulously compare the capabilities of our attack with those of assaults previously thwarted by conventional canvas fingerprinting challenge-response authentication techniques. Our findings conclusively demonstrate that none of the attacks, regardless of whether they utilize the “All in Once” or “Sequential” mode, can successfully breach CanvasDict’s defense mechanisms. This underscores the robustness of CanvasDict and its unmatched ability to resist even the most sophisticated attacks, further solidifying its position as a leading solution in canvas fingerprinting-based challenge-response authentication.

## II. RELATED WORK

### A. Browser Fingerprinting

Browser fingerprinting is the process through which a browser generates an identifier by gathering data while interacting with the device, encompassing elements like the browser itself, the operating system, and hardware. This amassed data can serve a multitude of purposes, including authentication for remote websites. Several research studies have highlighted that various methods, such as JavaScript, HTML5, CSS, and HTTP headers, can be employed to collect this data. JavaScript functions as a scripting language for web applications, offering diverse APIs capable of retrieving browser and system information useful for fingerprinting, including details like time zone and plugins as demonstrated by Eckersley [11], and navigator and screen information as shown

by Nikiforakis et al. [18]. HTML5, as a markup language for web content presentation, was proven by Mowery et al. [17] to utilize the new “canvas” element for browser fingerprinting.

### B. Canvas Fingerprinting

Canvas fingerprinting, explored by Acar et al. [8], is a browser fingerprinting method providing user identification. This method uses HTML canvas to craft shapes then converts into strings. Developers can customize creations with letters, numbers, fonts attributes and artistic adjustments in canvas image dimensions and background. The “toDataURL()” API offers a way to transform canvas images into strings. [1].

Englehardt et al. [12] explored canvas fingerprinting on the top 1 million websites, revealing its use for client tracking. Laperdrix et al. [14] emphasized canvas fingerprinting as a powerful user identification technique in 2016. Daud et al. [10] addressed canvas fingerprinting by employing a randomized approach to prevent consistent feature values across different clients.

Laperdrix et al. [13] proposed a canvas fingerprinting-based authentication method, demonstrating uniqueness in images from 99.9% of over 1.1 million devices. They explored potential attacks and advocated for canvas fingerprinting in authentication systems. Rivera et al. [21] achieved 82% accuracy in distinguishing browser instances, rising to 92% for users with different browsers or operating systems. Lin et al. [15] identified canvas fingerprinting techniques on certain tax and ridesharing websites.

## III. BACKGROUND

### A. Canvas Fingerprinting Challenge Response Authentication

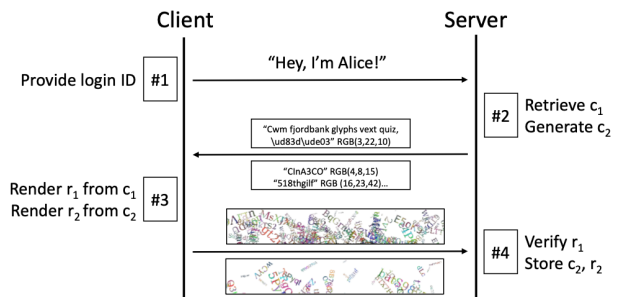


Fig. 1: Overview of Challenge Response Authentication Protocol [13].

The authentication technique known as canvas fingerprinting-based challenge-response authentication employs one of the browser fingerprinting methods known as canvas fingerprinting, as discussed in the work by Laperdrix et al. [13]. The process of challenge-response authentication is illustrated in Figure 1. When a user initially accesses a website, the browser sends the associated user ID to the server. In response, the server generates and sends two challenges, labeled as  $c_1$  and  $c_2$ , to the browser. It’s important

to note that  $c_1$  is retained from the user’s previous visit, while  $c_2$  is freshly generated by the server for each new session.

Upon receiving these challenges, the browser renders them and employs the JavaScript API “toDataURL()” to convert the rendered canvas images into two image strings,  $r_1$  and  $r_2$ . These image strings are then transmitted back to the server. The server uses  $r_1$  to verify whether it matches the string obtained during the user’s previous visit. If  $r_1$  successfully matches the stored string,  $r_2$  is stored alongside  $c_2$  for use in future authentication attempts. On the other hand, if  $r_1$  does not match, the user is directed towards alternative authentication methods, such as text-based or two-factor authentication, ensuring security.

Challenges like  $c_1$  and  $c_2$  are essentially objects that encapsulate randomized parameter settings, including attributes such as string content, string size and rotation, curve attributes, color gradients, the count of strings and curves, as well as the presence of shadows. By employing these parameters, the browser generates canvas images, subsequently converting them into strings, for instance, in the format “data:image/png;base64,iVBORw0KGgoAAAANS...” . Once these strings are rendered, the resulting image closely resembles the one depicted in Figure 2.

Incorporating this canvas fingerprinting-based challenge-response authentication mechanism enhances security by leveraging unique browser attributes to establish a robust user identification and verification process. This method ensures a dynamic and reliable approach to authentication, where the challenges’ parameters contribute to a diverse range of image outputs, making it significantly challenging for malicious actors to replicate or forge legitimate responses.



Fig. 2: Example of rendered image.

### B. Fingerprinting Spoofing

Liu et al. [16] posed a significant threat capable of compromising the privacy and security of applications that rely on browser fingerprinting techniques, which is called Gummy Browser. Gummy Browser offers three distinct attack methods: script injection, script modification, and manipulation of the browser’s built-in settings and debugging tools.

The Gummy Browser has the ability to emulate various JavaScript-based browser fingerprinting methods, including techniques like canvas fingerprinting. In the case of script modification, an attacker can overwrite the JavaScript API “toDataURL()” and assign it either a fixed or random string. Similarly, in script injection, an attacker can set breakpoints within scripts that call the “toDataURL()” JavaScript API, allowing them to replace the original values with fixed or random strings. Both of these methods enable the Gummy Browser to effectively spoof canvas fingerprinting, thereby allowing manipulation of features such as  $r_1$  and  $r_2$ .

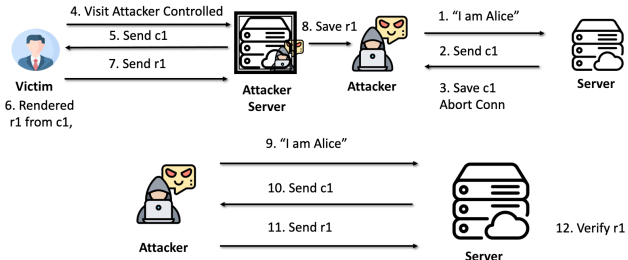


Fig. 3: Overview of Attack Model of CRSlash against regular Challenge Response authentication

One crucial aspect of Gummy Browser’s approach is that it consistently employs genuine feature values to imitate different characteristics, such as canvas fingerprinting features in the context of this research. Consequently, when an attack is executed, the visited website or victim might remain unaware that the visit originates from an attacker rather than the legitimate user.

Lin et al. [15] introduced FP-Spoofier, a tool with the ability to modify HTTP header information, such as the User Agent, through a browser extension.

## IV. CRSLASH DESIGN

In this section, we will initially outline the preconditions of CRSlash in Section IV-A, followed by an elucidation of the attack model of CRSlash against challenge-response authentication in Section IV-B. Subsequently, we enhance the attack, with a focus on canvas fingerprinting challenge-response authentication, detailed in Section IV-C. The complete intricacies of our attack implementation are expounded upon in Section IV-D. The process of obtaining challenges is delineated in Section IV-E, while the method of spoofing responses is explicated in Section IV-F.

### A. Preconditions

We have to set some preconditions to make sure that CRSlash can work. First of all, We assume that the attacker, referred to as  $A$ , has already acquired the victim’s, denoted as  $V$ , login credentials, including their username and password. It is possible that user may use the same username password combinations on different websites, and  $A$  may try the login credentials obtained from  $W_{random}$  on targeted website  $W_{target}$ . The objective of this attack is to bypass the challenge-response authentication method, thereby exposing its vulnerabilities, particularly in the context of canvas fingerprinting-based authentication.

Furthermore, we presume that the targeted website  $W_{target}$  employs the challenge-response authentication method. The attacker  $A$  possesses the capability to access the scripts, labeled as  $S$ , from  $W_{target}$ . These scripts  $S$  are loaded when any user, be it the victim, attacker, or others, visits  $W_{target}$ , ensuring uniformity among all visitors.

Upon obtaining the scripts  $S$ , the attacker  $A$  can deploy them on their controlled website  $W_{lure}$ . By enticing the victim  $V$  to

visit  $W_{lure}$ , the attacker  $A$  can initiate challenges and collect corresponding responses. Notably, no HTML content needs to be hosted on  $W_{lure}$ .

### B. Attack Model of CRSlash

Figure 3 illustrates the attack model of CRSlash. The attacker  $A$  initiates the attack by visiting the website  $W_{target}$  using the victim  $V$ 's username and password for authentication. Following the design of challenge-response authentication, the attacker receives challenge  $c_1$  from the server, constituting Steps 1 and 2.

To manipulate the authentication process, the attacker can set breakpoints within the scripts responsible for challenge-response authentication. One possible method is in browser DevTools [5] [3]. Despite these scripts generating canvas images and responses, the manipulated response won't be transmitted back to the server.

Alternatively, the attacker can employ browser extensions. Chrome and Firefox extensions can obstruct the loading of scripts from the target website and substitute them with custom scripts. This obviates the need for breakpoints and mitigates concerns about response times if the target website enforces server-side delays. Concurrently, the attacker can use the extension to save challenge  $c_1$  in JSON format locally, facilitating its later deployment on website  $W_{lure}$ . Additionally, the attacker can selectively insert event listeners to replace specific functions, allowing focused interference. For instance, the extension could monitor AJAX functions used to send objects to the server. Upon AJAX invocation, a pre-designed AJAX function lacking a return value could replace the original, preventing data transmission. This strategy prevents the delivery of a response, forcing the attacker to address consecutive authentication challenges during subsequent login attempts, following the retrieval of responses from the victim. This corresponds to Step 3 in Figure 3.

By extracting the challenge, the attacker  $A$  lures the victim  $V$  to  $W_{lure}$  (Step 4). Since the attacker  $A$  already possesses the username, and considering that numerous websites employ email addresses or phone numbers as usernames, it becomes simpler for  $A$  to dispatch emails or texts to reach  $V$  and entice  $V$  into visiting  $W_{lure}$ . At this point,  $W_{lure}$  integrates scripts designed to generate canvas images and responses using challenge  $c_1$  (Step 5). As the victim  $V$  accesses  $W_{lure}$ , their browser computes challenge  $c_1$ , subsequently transmitting response  $r_1$  to the server of  $W_{lure}$  (Steps 6 and 7).

Subsequently, the attacker  $A$  utilizes the spoofing method to spoof response  $r_1$  (Step 9), which is saved from  $W_{lure}$  before (Step 8), dispatching it to  $W_{target}$  (Step 11). Consequently,  $W_{target}$ 's server receives a response from the attacker  $A$  (Step 11), albeit one generated within  $V$ 's browser (Step 12). By inspecting  $W_{target}$ 's script, we identified the function responsible for response calculation. We then incorporated the obtained  $V$ 's response into the extension's event listener. Upon revisiting  $W_{target}$ , response  $r_1$  is transmitted to the server, undergoing rendering and transformation in alignment with challenge  $c_1$ .

### C. CRSlash against Canvas Fingerprinting Challenge Response Authentication

Through our analysis of Canvas Fingerprinting Challenge Response Authentication, we have refined our attack model as illustrated in Figure 4.

The attacker, denoted as  $A$ , initiates the process by accessing the target website  $W_{target}$ . By leveraging the victim's credentials (username and password), the attacker gains access (Step 1). Following the challenge-response authentication design, the server presents two challenges, labeled as  $c_1$  and  $c_2$  (Step 2). This differs from the standard challenge-response authentication. In this context of canvas fingerprinting challenge-response authentication, the user receives two challenges. The purpose of  $c_1$  is for verification, while the purpose of  $c_2$  is intended for future verification.

The attacker can strategically set breakpoints within the scripts associated with challenge-response authentication, given that canvas fingerprinting is related to script-based fingerprinting. As these scripts load in the attacker's browser, the canvas images are drawn, and responses are generated. However, these responses are not transmitted back to the server. The attacker can save  $c_1$  (Step 3) and then abort the connection (Step 4). Here, we only save  $c_1$ , although it's also acceptable to save both  $c_1$  and  $c_2$ .

Similar to the attack model against challenge-response authentication, the attacker can also exploit browser extensions in this canvas fingerprinting challenge-response authentication attack model. Chrome and Firefox extensions can prevent scripts from the website from loading, replacing them with customized scripts sourced from the attacker. This approach obviates the need for breakpoints and concerns regarding response timing, especially if the server at  $W_{target}$  implements a waiting period. Furthermore, the attacker can use the extension to download challenges  $c_1$  and  $c_2$  in JSON format, storing them locally for future use on the lure website  $W_{lure}$ . The extension also allows event listeners to selectively replace specific functions, avoiding the complete replacement of scripts. For instance, by monitoring the AJAX function responsible for returning objects to the server, the extension can intercept the event and replace the existing AJAX function with a predetermined one that doesn't yield any return value. This action prevents responses from being sent to the server, enabling the attacker to potentially face additional authentication challenges during subsequent logins after acquiring responses from the victim.

With the challenges obtained, the attacker  $A$  proceeds to entice the victim  $V$  to visit the lure website  $W_{lure}$ . Within  $W_{lure}$ , embedded scripts create canvas images and generate responses, specifically  $r_1$ , using challenges  $c_1$ . Once the victim  $V$  accesses the lure website (Step 5), their browser receives  $c_1$  (Step 6). Subsequently, it computes challenge  $c_1$  (Step 7) and forwards response  $r_1$  to the  $W_{lure}$  server (Step 8). At this point, the attacker can save  $r_1$  for future logins to bypass authentication (Step 9).

Following this, the attacker  $A$  employs a fingerprinting

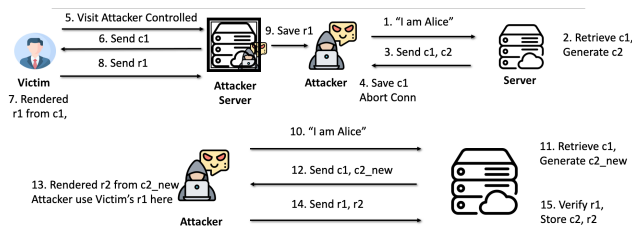


Fig. 4: Overview of Attack Model of CRSlash against Canvas Fingerprinting Challenge Response Authentication.

spoofing method, such as Gummy Browser, to forge responses and dispatch them to the target website  $W_{target}$ . However, a notable limitation of the Gummy Browser extension lies in its inability to natively replace responses. To address this, we’ve developed an extension that incorporates Gummy Browser’s spoofing technique. By analyzing the script of the  $W_{target}$  website, we identify the function responsible for response calculation. We preload the responses obtained from the victim  $V$  into the extension’s event listener. As the attacker  $A$  revisits  $W_{target}$  (Step 10), the server retrieves  $c_1$  and generates  $c_{2-new}$  (Step 11). These are then sent to the attacker  $A$  (Step 12), who in turn receives responses  $r_1$  and  $r_{2-new}$  (Step 14). These responses are derived from challenges  $c_1$  and  $c_{2-new}$  respectively (Step 13). Importantly,  $c_2$  might be dynamically generated with each visit; hence, we employ  $c_{2-new}$  in this context instead of  $c_2$ .

In this CRSlash scenario, although the server sends two challenges to the attacker, they only need to preserve  $c_1$  instead of both, as  $c_1$  remains constant, while  $c_2$  changes before the attacker successfully completes authentication. Even if the server doesn’t explicitly indicate which challenge is  $c_1$  and which is  $c_2$ , the attacker only needs to undergo authentication twice to obtain four challenges. The challenge that appears twice among these four challenges should be identified as  $c_1$ . In Step 13, we note that the attacker can use their own device to generate  $r_{2-new}$  instead of obtaining it from the victim. According to the canvas fingerprinting-based challenge-response authentication protocol,  $c_{2-new}$  will be utilized for future verification. By rendering from the attacker’s device, the attacker can pass continuous or future authentication using their own device, avoiding the need to continuously lure the victim into visiting  $W_{lure}$  to obtain responses from the victim’s device.

#### D. Real-Time Phishing

This attack involves a real-time phishing approach, where Attacker  $A$  aims to rapidly obtain responses from Victim  $V$ . Upon receiving challenges  $c_1$  and  $c_2$  from the target website  $W_{target}$ , Attacker  $A$  must promptly update the spoofed website  $W_{lure}$  by incorporating challenge  $c_1$  into the response computation. This action can be executed within seconds.

Once the update of website  $W_{lure}$  is completed, the attacker needs to swiftly entice Victim  $V$  to access the spoofed website

$W_{lure}$ , thus characterizing the attack as a real-time phishing attack.

Upon obtaining responses  $r_1$  from the manipulated website  $W_{lure}$ , the attacker can expeditiously substitute relevant variables in the scripts of the legitimate website  $W_{target}$  and deactivate the breakpoint in a brief span of time.

It’s important to note that whenever the attacker confronts the challenge-response authentication process, if the breakpoint is set prior to the execution of the functions responsible for transmitting response  $r_1$  to the server, the server will not consistently receive  $c_1$  and  $r_1$  as the validation data. Consequently, this implies that  $c_1$  will remain constant for Attacker  $A$  as long as Victim  $V$  does not visit website  $W_{target}$ . However,  $c_2$  will exhibit variability across distinct visits due to its random generation. Since the Attacker  $A$ ’s device is capable of generating  $c_2$  and transmitting  $r_{2A}$  to the server, following the canvas fingerprinting challenge-response authentication protocol,  $r_{2A}$  will take on the role of the new  $r_1$ , and the associated  $c_2$  will be regarded as the new  $c_1$ . This implies that as long as the victim  $V$  has not logged in, the attacker will have control over this account. Consequently, the attacker will consistently bypass the canvas fingerprinting challenge-response authentication, whether it’s in the context of continuous authentication or subsequent logins.

Although Attacker  $A$  could potentially obtain  $r_1$  from Victim  $V$  at a later time, disregard  $r_2$ , and dispatch an arbitrary response  $r_{random}$  to the server, a subsequent visit by Victim  $V$  to  $W_{target}$  would expose the inconsistency between the calculated response  $r_{2V}$  based on  $c_{2-new}$ , and the server-stored response  $r_{2A}$  supplied by the attacker. Consequently, Victim  $V$  would encounter alternative authentication measures, such as 2FA, and eventually realize the prior intrusion.

Such an outcome contradicts the attacker’s intentions. To evade detection by Victim  $V$  or the legitimate website  $W_{target}$ , the attack must remain concealed. Hence, the attack strategy must adhere to real-time phishing principles.

#### E. Challenges Acquisition

When users visit a website, their browsers invariably load all embedded scripts. Challenges can be allocated either on the server side or on the client side. In Laperdrix et al. [13], challenges are allocated on the server side, whereas in the shared code for challenge-response authentication [20], challenges are assigned on the client side. Irrespective of their point of allocation, these challenges must be incorporated into canvas drawings and subsequently converted into strings on the client side. By identifying the JavaScript API “toDataURL()”, the methodology for generating canvas images and the pertinent variables employed in the drawing process can be ascertained. These variables represent the challenges.

An examination of the open-source code [20] reveals that the function “generateChallenge()” within the script file “generator.js” is responsible for generating challenges, while the function “getData()” in the script file “picassauth” is utilized for crafting canvas images and their conversion into strings. By introducing a breakpoint prior to the “return” statement within

the “generateChallenge()” function, the challenge specifics can be extracted using the “console.log” function, with the challenge remaining unaffected in the response calculation until the breakpoint is removed.

#### F. Response Spoofing

Upon obtaining challenges  $c_1$  and  $c_2$  from website  $W_{target}$ , Attacker A can modify the scripts on  $W_{lure}$  to obtain responses  $r_1$  and  $r_2$ .

Before removing the initial breakpoint in Attacker A’s browser, a secondary breakpoint can be established just before the invocation of the JavaScript API “toDataURL()”. At this juncture, the attacker has two choices: employing Gummy Browser script modification or script injection.

Given that responses  $r_1$  and  $r_2$  manifest as scripts following the “toDataURL()” call, in script modification, the attacker can twice overwrite the “toDataURL()” API with the values of  $r_1$  and  $r_2$  before releasing the second breakpoint on each occasion. Alternatively, in script injection, the attacker can substitute the script with a specific string. For instance, in the shared code [20], the response calculation transpires in the line “var data = canvas.toDataURL();” within the “getData()” function. Here, the attacker could replace “canvas.toDataURL()” with a predetermined string like “var data = data:image/png;base64,iVBORw0KGgoAAAANS”, subsequently releasing the second breakpoint.

### V. EXPERIMENTAL SETTINGS

We employed two devices for our experiments: a 2013 version MacBook Pro running MacOS 11.6 as the victim’s device, and a Windows desktop with Windows 11 as the attacker’s device. The IP address of the MacBook Pro is located in Texas, while the IP address of the Windows desktop is situated in California. The experiment was conducted in March 2023. The challenge-response authentication code for the canvas fingerprinting, which we tested, was sourced from GitHub [20].

We successfully deployed the website’s code on an AWS EC2 instance situated in the Northern Virginia location, referred to as  $W_{target}$ . Following the deployment, we employed an attacker’s device to access the website 50 times, resulting in the generation of 50 distinct challenges. Subsequently, these challenges were integrated into a separate website hosted on another AWS EC2 instance, also located in Northern Virginia and identified as  $W_{lure}$ .

Upon the  $W_{lure}$  server receiving the 50 responses, we proceeded to utilize the attacker’s device to transmit these responses to the server of  $W_{target}$ . This series of actions forms a comprehensive overview of our approach.

Before conducting the experiment, we initially employed two devices, both equipped with the same browser, to generate identical challenges. We then converted the responses to strings using the “toDataURL()” API. The results revealed a noteworthy observation: irrespective of whether the two devices utilized Firefox or Chrome, the strings derived from the two devices differed significantly. This finding strongly

suggests that the canvas fingerprints between these two devices are entirely distinct.

To elaborate further, this outcome underscores the inherent uniqueness of canvas fingerprints as they vary across devices, even when utilizing the same browser. Such distinctions further solidify the efficacy of canvas fingerprinting as a technique for device identification and tracking in online environments.

#### A. Extension Design

We have delineated the functions used to generate challenges in section IV-E, and we have devised an extension to modify these functions for our experimental objectives. Upon the attacker’s initial encounter with the challenge-response authentication, subsequent to employing the victim’s credentials, the extension can download the challenges to the local disk, halt challenge calculations, and intercept the transmission of responses back to the server. Once the attacker obtains the responses from the victim, and subsequently faces the challenge-response authentication again, the extension can substitute the calculated responses with the victim’s authentication.

Given that the code [20] lacks server-side functionality, our extension will not handle any Ajax code. Instead, it will exclusively consist of event listeners for challenge and response-related functions to facilitate downloads and replacements. Our extension obstructs response calculation, resulting in the server never receiving responses from the attacker. Consequently, the displayed image will appear blank, and the hash value will be empty as well.

#### B. Image Comparison Implementations

For image comparison, we have adopted two methods: hash value comparison and histogram comparison.

In hash value comparison, we begin by importing two rendered canvas images: one downloaded from the victim’s device and the other obtained from the attacker’s device (this image is acquired from the victim). Subsequently, we compute the hash value of both images and verify if the two hash values are identical.

In histogram comparison, we also import two rendered canvas images, mirroring the procedure employed in hash value comparison. We then employ the Python library “OpenCV” [2] to calculate the histogram similarity between the two images.

### VI. RESULTS AND EVALUATION

We have tested pairs of images – one from the victim’s device and the other from the attacker’s – across 50 different challenges. The results of the image comparisons indicate that all 50 pairs of images are all identical. Examples of these pairs can be seen in Figure 5 and Figure 6, as well as Figure 8 and Figure 9. Figure 7 illustrates the histogram differences between Figure 5 and Figure 6.

#### A. What is the pass rate based on Image comparison?

The two images are identical and stem from the same challenge – one from the victim’s device and the other from the attacker’s acquisition. This outcome is not surprising



Fig. 5: Rendered image using response in browser Chrome.



Fig. 6: Rendered image using response from Chrome in browser Firefox.



Fig. 7: Difference between Figure 5 and Figure 6



Fig. 8: Rendered image using another response in Chrome.



Fig. 9: Rendered image using another response in Firefox.

since the attacker’s image originates from the victim as well. Consequently, the comparison essentially involves two images sourced solely from the victim’s device and derived through the same set of challenges. As a result, the similarity score is inherently 1, rendering all 50 response pairs identical in nature. This underscores the consistent nature of the comparison results.

#### B. Will the victim find that he was attacked before?

Our non-real-time attack involves sending  $r_2$  back to the server. This value,  $r_2$ , is calculated on the attacker’s device. Consequently, during the victim’s subsequent visit, they will fail the challenge-response authentication. This attack strategy is detailed in [13].

In accordance with the mechanism outlined in the mentioned study, users might still be required to undergo two-factor authentication if the generated images cannot be matched or if the similarity score falls below the baseline. As a result, victims might remain unaware of whether they have been targeted in the past. The sole observable change for them would be the necessity to complete two-factor authentication, as opposed to breezing through challenge-response authentication as before.

#### C. What is the pass rate based on Image comparison?

The pass rate, as determined through image comparison, relies on the degree of similarity existing between two given images. In this scenario, the comparison takes place between

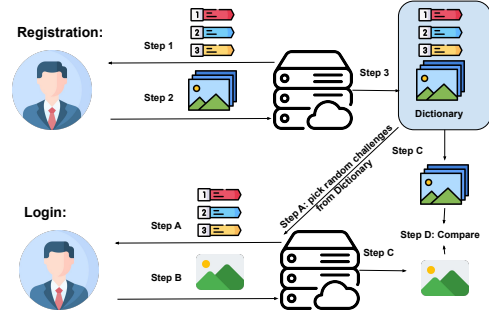


Fig. 10: Overview of new challenge response authentication protocol.

images retrieved from the victim’s device and those procured by the attacker. Notably, there emerges a noteworthy observation: several pairs of images exhibit a striking level of resemblance. This phenomenon can be anticipated, considering that the attacker’s images are themselves derived from the victim’s repository. As a result, the essence of the comparison essentially revolves around two images that share their origin on the victim’s device, both having undergone identical challenges and conditions.

Hence, the resultant similarity score consistently registers at 1 across all 50 response pairs, reinforcing the notion that these images exhibit an indistinguishable likeness.

#### D. Will the victim realize that they were attacked?

In our non-real-time attack scenario, the server receives  $r_2$ , which has been computed on the attacker’s device. Consequently, when the victim makes a subsequent visit, their challenge response authentication will fail to succeed. This situation is in accordance with the mechanism outlined in [13]. In cases where the rendered images do not align or if the similarity score drops below the baseline, the user might still be prompted for two-factor authentication. As a result, the victim could potentially remain oblivious to whether they have been targeted in an attack. The only noticeable change for them would be the requirement to undergo two-factor authentication, as opposed to the immediate success they are accustomed to with challenge response authentication. This change in authentication behavior might be the sole indication of a potential attack.

## VII. CANVASDICT DESIGN

In this section, we will present the intricacies involved in designing our innovative challenge-response authentication method: CanvasDict. Furthermore, we will provide a comprehensive understanding of how this novel authentication approach effectively counters the array of attacks meticulously delineated in section IV.



(a) MacOS Chrome (b) MacOS Firefox (c) Windows Chrome (d) Windows Firefox

Fig. 11: Sample of the letter “Q” using the following conditions: font style: No style, font size: 60px, font color: red.

### A. Authentication Design

To begin with, the website server must possess a collection of font styles, font sizes, and font colors. This information will be utilized to generate personalized canvas image data for each user. We refer to this collection as the “Dictionary”. Figure 10 illustrates the mechanism behind our designed challenge-response authentication approach. Examples of font styles include: “Helvetica”, “Arial”, “Verdana”, etc. Examples of font sizes include: 20px, 40px, 60px, while examples of font colors encompass: Red, blue, green, white, black. Meanwhile, the font styles list should also include “no styles”.

1) *Dictionary Creation*: Upon user registration on this website, the system employs a random selection of font styles, font sizes and fonts colors from an extensive list. The web server is equipped with a dataset of font styles(including no styles), font sizes and colors. Whenever a user attempts to register an account, the server selects a random assortment of font styles(including no styles), font sizes and colors. These selections are combined to form a challenge data package, which is then dispatched to the user’s browser (Step 1). Notably, the challenge data package varies among different users attempting registration, as well as for the same user registering multiple times using the same device and browser.

For example, our calculations indicate that the complete list encompasses 50 font styles(including no styles), 100 distinct font sizes and 30 unique font colors. In the current registration instance, the server has opted for 3 font styles(Helvetica, Verdana, no styles), 3 font sizes (3, 7, 21) and 3 font colors (red, pink, green). Consequently, there exists a total of 27 (3 \* 3 \* 3) potential combinations of font sizes and colors, which the server utilizes to formulate challenges for users.

Following this, the user’s browser initiates the assembly of a “Dictionary.” Using the 27 predetermined font combinations, the browser generates 27 canvas images for each character found within the sets of alphabetical letters (“a” to “z,” “A” to “Z”) and numbers (“0” to “9”). These resulting images, serving as responses to the challenges, are subsequently transmitted back to the server (Step 2). The server catalogues this collection of data as the “Dictionary” within the database, and this dataset is utilized for future login authentication purposes (Step 3). Concurrently, the font combinations employed in the creation of the “Dictionary” are also stored for reference.

For consistent font styles, font sizes and font colors applied to identical letters or numbers, it is essential that various de-



(a) Windows Chrome Device 1 no style (b) Windows Chrome Device 2 no style (c) Windows Chrome Device 1 Arial (d) Windows Chrome Device 2 Arial

Fig. 12: Left two images: sample of the letter “Q” using the following conditions: font style: no style, font size: 60px, font color: red in two Windows devices. Right two images: sample of the letter “Q” using the following conditions: font style: Arial, font size: 60px, font color: red in two Windows devices.

vices and browsers produce distinct canvas image values. This necessitates testing the effectiveness of our canvas drawing method in preserving the individuality of canvas fingerprinting.

To conduct our experiments, we utilized two devices: a 2013 MacBook Pro running MacOS 11.6 and a Windows desktop with Windows 11. The graphical representation in Figure 11 showcases four diverse canvas images generated by employing the same font combination (the letter “Q” with no font style, a font size of 60px and a font color of red) on both the MacBook Pro and Windows desktop, using the Chrome and Firefox browsers. These figures were derived from their respective canvas image strings, obtained through the utilization of the “toDataURL()” API.

At an initial glance, the four figures might seem similar, but it’s crucial to recognize that the underlying canvas structures are actually distinct from each other.

- 1) In MacOS-Chrome combination, the string is: “data:image/png;base64,.../AAAAABJRU5Erkkgg==”.
- 2) In MacOS-Firefox combination, the string is: “data:image/png;base64,...hgzAAAAASUVORK5CYII=”.
- 3) In Windows-Chrome combination, the string is: “data:image/png;base64,...VxRCAAAAAEIFTkSuQmCC”.
- 4) In Windows-Firefox combination, the string is: “data:image/png;base64,...F7pFAAAAAEIFTkSuQmCC”.

The preceding comparison involved a macOS device and a Windows device. Furthermore, we conducted tests to analyze distinctions among various Windows devices. The comprehensive comparison findings can be found in Figure 12a and 12b. Additionally, we contrasted the canvas strings across diverse canvas images.

- 1) In Windows-Chrome-1 combination, the string is: “data:image/png;base64,...VxRCAAAAAEIFTkSuQmCC”.
- 2) In Windows-Chrome-2 combination, the string is: “data:image/png;base64,...pTKAAAAAASUVORK5CYII=”.

The findings have revealed that the two canvas compositions are distinctly dissimilar from each other.

2) *CanvasDict-All in Once*: Firstly, let’s introduce the “All in Once” authentication method. In this approach, all font combinations are simultaneously provided to the user to generate corresponding canvas images.

During the authentication process, when the user encounters a challenge, the server randomly selects a set of characters. The number of characters matches the available font combinations. These font combinations are drawn from the dataset that corresponds to the fonts chosen by the user during their registration. The website creates a dictionary based on these selections.

For instance, if a user's registration involved 50 font styles, font sizes ranging from 1px to 100px, and font colors encompassed options like "red", "green", "black", "blue", "grey", "pink", "yellow", and more, totaling 30 colors, these parameters are considered. Subsequently, during login, the server might designate characters like "a", "H", "o", and the number "7." The corresponding font combinations could be "Helvetica, 7px, red," "Verdana, 7px, green", "Verdana, 21px, green", and "No style, 3px, pink".

It's important to note that font styles, font sizes and colors are limited to the combinations established during the user's registration process, as outlined in the "Dictionary" generated by the website. The server transmits these four font combinations to the user, enabling the browser to create associated canvas images, which are then converted into canvas strings.

Upon receiving these canvas image strings, the server compares them against the stored canvas image strings of the font combinations stored in the database. Successful authentication occurs when all image pairs match. In cases where image pairing fails, such as the second pair not matching, the authentication process fails immediately.

It's crucial to emphasize that not all possible combinations are generated during this stage. Only combinations that were used in constructing the "Dictionary" are considered for selection. To illustrate, in the given example, the selection is limited to three font styles (Helvetica, Verdana, No styles), three font sizes (3, 7, 21) and three font colors (red, pink, green). This contrasts with the larger pool of 50 font styles, 100 font sizes and 30 font colors that are available.

3) *CanvasDict-Sequential*: Our second designed CanvasDict method is named "Sequential". In this approach, each font combination undergoes independent authentication one by one.

During the authentication process, the server randomly selects characters in correspondence with the number of available font combinations. Similar to the "All in Once" authentication approach, all font combinations must originate from the font combination dataset linked to the user's registration. This dataset is created when the user registers on the website and establishes their personalized dictionary. To illustrate further, let's consider the following scenario: the server may choose characters such as "a", "H", "o", and the number "7". This selection leads to the formation of font combinations such as "Helvetica, 7, red", "Verdana, 7, green", "Verdana, 21, green", and "No styles, 3, pink". The server then transmits the initial character "a" along with the corresponding font combination "7, red" to the user. Subsequently, the user's browser generates a canvas image, which is sent back to the server. Upon receiving the image, the server compares

it with the stored image of the character "a", utilizing the font combination "7, red" from the database. If a match is identified, the server proceeds to send the subsequent character "H" along with its associated font combination "7, green" to the user, thus repeating the process. This sequential pattern continues until all four pairs of images are either successfully matched or encounter failure. The successful matching of all pairs grants the user authentication clearance. However, the authentication process immediately fails if any pair, such as the second pair in this instance, does not match.

Similar to the preceding method, only the combinations utilized in constructing the "Dictionary" are generated at this particular stage. In the presented example, the selection is restricted to three font styles (Helvetica, Verdana, No style), three font sizes (3, 7, 21) and three font colors (red, pink, green), as opposed to the broader range of 50 font styles, 100 font sizes and 30 font colors.

4) *Discussion of Font Style*: In our experiment, we made an intriguing observation: even when using the same browser (e.g., Chrome), differences emerged in the canvas images and canvas strings when the operating system differed (e.g., MacOS and Windows). These differences were evident despite identical settings for font style, font size, font color, and the same letter/number input.

In contrast, we noticed a distinct behavior when maintaining uniformity in certain parameters. Specifically, within the same browser (e.g., Chrome) and a consistent operating system (e.g., Windows), we observed that canvas images and canvas strings remained identical. This phenomenon occurred when utilizing font styles that were universally supported on both devices (e.g., Arial) and employing font colors that were compatible (e.g., red). Notably, the similarity extended across canvas images even when comparing two distinct Windows devices.

Figure 12c and 12d illustrate a side-by-side comparison of two canvas images. Both images were generated using font style: Arial, font size: 60px, and font color: red. Remarkably, despite originating from separate Windows devices, the canvas images and corresponding strings exhibited complete congruence. This outcome can be attributed to the fact that both devices supported the "Arial" font.

However, it's worth highlighting that this consistent behavior did not manifest when we opted for the "No style" font setting, as demonstrated in Figure 11. This discrepancy underscores the rationale behind advocating for the inclusion of a "No style" option during the registration and login processes.

### B. *CanvasDict's Resilience Against Attacks*

In this section, we elucidate how our authentication method thwarts CRSlash enumerated in section IV. We maintain the assumption that the attacker can access credentials such as usernames, passwords from the victim, and can inspect authentication function scripts.

In our authentication process, the sole opportunity to acquire the complete list of font styles, font sizes, and font colors in the dictionary occurs during registration. Importantly, the

attacker lacks the capability to obtain the victim’s full set of challenges during the victim’s registration. As a consequence, the attacker is unable to prearrange responses. Additionally, any attempt by the attacker to mimic victim registration in order to glean font style, size, and color information would be futile. For each user, be it victim or attacker, the selection of styles, sizes, and colors is distinct due to the random nature of the registration process. Consequently, during registration, the attacker can glean no information relevant to the victim.

During the login process, attackers face two distinct scenarios in the context of both “All in Once” and “Sequential” authentication methods.

1) *All in Once against CRSlash*: Upon login, when the attacker successfully provides the correct username and password, they receive characters or numbers accompanied by font combinations for the current authentication attempt. It’s crucial to recognize that each user’s login prompts involve different characters/numbers and font combinations each time. This means that even if the attacker acquires the font combinations for a given login attempt, these combinations may not be applicable in future login attempts. For instance, in the example with 27 font combinations, even if the server offers 30 font styles, sizes, and colors, the attacker might need at least 30 attempts to potentially access the complete list—assuming they are extraordinarily fortunate. Furthermore, failed login attempts prompt the victim to undergo alternative authentication methods, such as phone call or text verification, which can trigger updates to the dictionary with new font combinations. Consequently, any data the attacker gathered in previous attempts becomes obsolete. Thus, during login, the attacker remains unable to bypass authentication using any pre-existing information.

2) *Sequential against CRSlash*: In the login process, when the attacker successfully enters the correct username, password, and cookies, they only receive one character or number accompanied by a single font combination. Failure to pass the authentication for the first character/font combination hinders any further progress for the attacker. Each login attempt presents different characters/numbers and font combinations, with varying totals for each attempt. Consequently, even if the attacker obtains a font combination for a given login attempt, these combinations are unlikely to be useful in future attempts. As with the “All in Once” method, failed login attempts prompt the victim to experience alternative authentication methods, potentially leading to dictionary updates.

## VIII. DISCUSSION

In this section, we will first discuss the canvas diversity among different fonts, then delve into the detection methods for our CRSlash, then the limitations inherent in both CRSlash and CanvasDict.

### A. Canvas Diversity

Through our experiments, we observed that certain fonts produce identical canvas images, posing a potential security risk to CanvasDict. To ensure a high level of security for

TABLE I: The number of unique canvas strings and unique fonts. the first column corresponds to the tested font colors, while the number in the first row signifies the font size. Columns 2 through 12 display the counts of unique canvas strings for various font sizes and color settings. The last column provides the count of common unique fonts associated with specific fonts across all different font size settings.

Fonts	1	2	3	4	5	10	15	20	40	70	100	Unique Fonts
Red	119	146	149	150	150	158	163	152	153	153	153	84
Blue	119	146	149	150	150	158	163	152	153	153	153	84
Green	123	146	149	150	151	158	163	152	153	153	153	90
Pink	124	146	149	150	151	158	163	152	153	153	153	93
Black	119	146	149	150	150	158	163	152	153	153	153	84
Gold	124	146	149	150	151	158	163	152	153	153	153	93

CanvasDict, we assessed canvas diversity across various font combinations. Our MacBook Pro experiment devices helped us identify a total of 246 fonts. The results of canvas diversity are presented in Table I. Notably, the number of unique canvas strings varies under different combinations. For instance, when the font size is 2 and the color is red, there are 146 unique canvas strings. Within these 146 unique canvas strings, 84 fonts generate uniqueness specifically with the red color. Across all colors, 83 fonts can generate unique canvas strings. The findings indicate that out of the total 246 fonts, we can utilize a minimum of 83 fonts for implementing CanvasDict. Nevertheless, there is potential to incorporate additional fonts into CanvasDict, considering that different font colors have distinct sets of fonts that can generate unique canvas strings.

### B. Attack Detection

Concerning the previous challenge-response authentication detailed in [13], CRSlash retains its undetectable nature. This phenomenon arises from the attacker’s exploitation of the challenge received by the server, coupled with the victim’s browser generating the corresponding response based on this challenge. Consequently, challenges that match on the server side will invariably lead to successful authentication attempts for the attacker. However, it is important to acknowledge that the attacker may still encounter failures in their attempts to pass the authentication process.

When significant alterations transpire on the victim  $V$ ’s device, such as updates to the operating system (e.g., transitioning from Windows 10 to Windows 11) or updates to the browser (e.g., upgrading from Chrome 70 to Chrome 80), the stored response on the server will no longer synchronize with the most recently generated one. While this discrepancy effectively thwarts the attacker’s authentication endeavor, it also impedes the victim  $V$ ’s authentication success. The server only discerns that the two responses cannot be harmonized and subsequently redirects the visitor to alternative authentication methods. Nevertheless, the server remains oblivious to whether the visitor is indeed the  $V$ , the attacker  $A$ , or a different individual.

The prior challenge-response authentication approach introduced in [13] also posited that continuous authentication might serve as a defense mechanism against attacks. Nevertheless, even under such circumstances, our attack remains undetected.

Our attack strategy solely necessitates the acquisition of the newly generated response  $r_1$  from victim  $V$ . Subsequent responses are exclusively generated by the browser of attacker  $A$ , rather than victim  $V$ 's browser. Given that the server does not validate the raw fingerprinting data, the attacker is exempted from the obligation to forge any fingerprints or acquire new responses from victim  $V$  after successfully passing the initial authentication phase.

In the context of CanvasDict, our attack proves unsuccessful, triggering attack detection mechanisms following multiple failed attempts.

### C. Limitations

1) *Limitation of CRSlash*: CRSlash operates in a non-real-time manner. Once the attacker obtains challenges from the targeted website  $W_{target}$ , there exists a plausible scenario where the victim  $V$ , might have visited the website before the attacker receives a response.

Another noteworthy limitation pertains to a situation where even if the attacker successfully navigates through authentication and compromises victim  $V$ 's account, there remains a possibility that victim  $V$  would detect the breach. One instance that underscores this limitation is when victim  $V$ 's device remains unaltered, exhibiting no updates or modifications.

2) *Limitation of CanvasDict*: Primary among the limitations of CanvasDict is its substantial storage demand on the server side. Each unique combination of letter/number fonts necessitates separate storage for the corresponding rendered image. One potential solution entails reducing the number of combinations, perhaps by trimming the 50 font styles, 100 font sizes, and 30 font colors down to 10 each. The other limitation pertains to the usage of CanvasDict on mobile platforms. Testing two iPhone 13 Pro Max devices using the same font combination yielded identical responses. Despite possessing identical iOS versions, the devices differed in their browser (Chrome) versions. This outcome suggests that CanvasDict is unsuitable for mobile platforms. Presently, a definitive solution has not been identified. This area will be explored in forthcoming research endeavors.

## IX. CONCLUSIONS

In this paper, we present an attack model CRSlash that targets the current challenge-response authentication specifically. We conducted experiments to evaluate the effectiveness of this attack on challenge-response authentication systems based on canvas fingerprinting. The results reveal that the existing challenge-response authentication methods based on canvas fingerprinting are incapable of countering this attack, whether employed for one-time authentication or continuous authentication. To tackle this vulnerability, we developed a novel challenge-response authentication method based on canvas fingerprinting, known as CanvasDict. Subsequently, we assessed the efficacy of our proposed attack against two different modes of CanvasDict, and the findings demonstrated that CanvasDict effectively mitigates the attack.

## REFERENCES

- [1] Htmlcanvaselement: toDataURL(). <https://developer.mozilla.org/en-US/docs/Web/API/HTMLCanvasElement/toDataURL>.
- [2] opencv-python 4.6.0.66. <https://pypi.org/project/opencv-python/>.
- [3] Pause your code with breakpoints. <https://developer.chrome.com/docs/devtools/javascript/breakpoints/>.
- [4] The pros and cons of two-factor authentication (2fa). <https://messente.com/blog/most-recent/2fa-pros-and-cons>.
- [5] Set a breakpoint. [https://firefox-source-docs.mozilla.org/devtools-user/debugger/how\\_to/set\\_a\\_breakpoint/index.html](https://firefox-source-docs.mozilla.org/devtools-user/debugger/how_to/set_a_breakpoint/index.html).
- [6] What happens if i lose my device with 2fa on it? <https://cryptocurrencyfacts.com/what-happens-if-i-lose-my-device-with-2fa-on-it/>.
- [7] What is two-factor authentication (2fa)? and why you need one? <https://www.acronis.com/en-us/blog/posts/two-factor-authentication/>.
- [8] Gunes Acar, Christian Eubank, Steven Englehardt, Marc Juarez, Arvind Narayanan, and Claudia Diaz. The web never forgets: Persistent tracking mechanisms in the wild. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 674–689, 2014.
- [9] Anupam Das, Joseph Bonneau, Matthew Caesar, Nikita Borisov, and XiaoFeng Wang. The tangled web of password reuse. In *NDSS*, volume 14, pages 23–26, 2014.
- [10] Nor Izyani Daud, Galoh Rashidah Haron, and Siti Suriyati Syd Othman. Adaptive authentication: Implementing random canvas fingerprinting as user attributes factor. In *2017 IEEE Symposium on Computer Applications & Industrial Electronics (ISCAIE)*, pages 152–156. IEEE, 2017.
- [11] Peter Eckersley. How unique is your web browser? In *International Symposium on Privacy Enhancing Technologies Symposium*, 2010.
- [12] Steven Englehardt and Arvind Narayanan. Online tracking: A 1-million-site measurement and analysis. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 1388–1401, 2016.
- [13] Pierre Laperdrix, Gildas Avoine, Benoit Baudry, and Nick Nikiforakis. Morellian analysis for browsers: Making web authentication stronger with canvas fingerprinting. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 43–66. Springer, 2019.
- [14] Pierre Laperdrix, Walter Rudametkin, and Benoit Baudry. Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 878–894. IEEE, 2016.
- [15] Xu Lin, Panagiotis Iliia, Saumya Solanki, and Jason Polakis. Phish in sheep's clothing: Exploring the authentication pitfalls of browser fingerprinting. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1651–1668, 2022.
- [16] Zengrui Liu, Prakash Shrestha, and Nitesh Saxena. Gummy browsers: Targeted browser spoofing against state-of-the-art fingerprinting techniques. In Giuseppe Ateniese and Daniele Venturi, editors, *Applied Cryptography and Network Security*, pages 147–169. Cham, 2022. Springer International Publishing.
- [17] Keaton Mowery and Hovav Shacham. Pixel perfect: Fingerprinting canvas in HTML5. In Matt Fredrikson, editor, *Proceedings of W2SP 2012*. IEEE Computer Society, May 2012.
- [18] Nick Nikiforakis, Alexandros Kapravelos, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *2013 IEEE Symposium on Security and Privacy*, pages 541–555. IEEE, 2013.
- [19] Sarah Pearman, Jeremy Thomas, Pardis Emami Naeini, Hana Habib, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, Serge Egelman, and Alain Forget. Let's go in for a closer look: Observing passwords in their natural habitat. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 295–310, 2017.
- [20] Plaperdr. Plaperdr/morellian-canvas: Repository of the article "morellian analysis for browsers: Making web authentication stronger with canvas fingerprinting". <https://github.com/plaperdr/morellian-canvas>.
- [21] Esteban Rivera, Lizzy Tengana, Jesús Solano, Christian López, Johana Flórez, and Martín Ochoa. Scalable and secure html5 canvas-based user authentication. In *International Conference on Applied Cryptography and Network Security*, pages 554–574. Springer, 2022.