(Cross-)Browser Fingerprinting via OS and Hardware Level Features

Yinzhi Cao
Lehigh University
yinzhi.cao@lehigh.edu

Song Li
Lehigh University
sol315@lehigh.edu

Erik Wijmans†
Washington University in St. Louis
erikwijmans@wustl.edu

***Abstract*—In this paper, we propose a browser fingerprinting technique that can track users not only within a single browser but also across different browsers on the same machine. Specifically, our approach utilizes many novel OS and hardware level features, such as those from graphics cards, CPU, and installed writing scripts. We extract these features by asking browsers to perform tasks that rely on corresponding OS and hardware functionalities.**

Our evaluation shows that our approach can successfully identify 99.24% of users as opposed to 90.84% for state of the art on single-browser fingerprinting against the same dataset. Further, our approach can achieve higher uniqueness rate than the only cross-browser approach in the literature with similar stability.

I. Introduction

Web tracking is a debatable technique used to remember and recognize past website visitors. On the one hand, web tracking can authenticate users—and particularly a combination of different web tracking techniques can be used for multi-factor authentication to strengthen security. On the other hand, web tracking can also be used to deliver personalized service—if the service is undesirable, e.g., some unwanted, targeted ads, such tracking is a violation of privacy. No matter whether we like web tracking or whether it is used legitimately in the current web, more than 90% of Alexa Top 500 websites [39] adopt web tracking, and it has drawn much attention from general public and media [6].

Web tracking has been evolving quickly. The first-generation tracking technique adopts stateful, server-set identifiers, such as cookies and evercookie [21]. After that, the second-generation tracking technique called fingerprinting emerges, moving from stateful identifiers to stateless—i.e., instead of setting a new identifier, the second-generation technique explores stateless identifiers like plug-in versions and user agent that already exist in browsers. The second-generation technique is often used together with the first to restore lost cookies. Both first and second generation tracking are constrained in a single browser, and nowadays people are developing third-generation tracking technique that tries to achieve cross-device tracking [16].

The focus of the paper is a 2.5-generation technique in between the second and the third, which can fingerprint a user not only in the same browser but also across different browsers on the same machine. The practice of using multiple browsers is common and promoted by US-CERT [42] and other technical people [12]: According to our survey,[1] 70% of studied users have installed and regularly used at least two browsers on the same computer.

The proposed 2.5-generation technique, from the positive side, can be used as part of stronger multi-factor user authentications even across browsers. From another angle, just as many existing research works on new cyber attacks, the proposed 2.5-generation tracking can also help to improve existing privacy-preserving works, and we will briefly discuss the defense of our cross-browser tracking in Section VII.

Now, let us put aside the good, the bad and the ugly usages of web tracking, and look at the technique itself. To fingerprint different browsers installed on the same machine, one simple approach is to use existing features that fingerprint single browser. Because many existing features are browser specific, the cross-browser stable ones are not unique enough even when combined together for fingerprinting. That is why the only cross-browser fingerprinting work, Boda et al. [14], adopts IP address as a main feature. However, IP address, as a network-level feature, is excluded from modern browser fingerprinting in the famous Panopticlick test [5] and many other related works [10, 20, 26, 32, 34, 36]. The reason is that IP address changes if allocated dynamically, connected via mobile network, or a laptop switches locations such as from home to office—and is unavailable behind an anonymous network or a proxy.

In the paper, we propose a (cross-)browser fingerprinting based on many novel OS and hardware level features, e.g., these from graphics card, CPU, audio stack, and installed writing scripts. Specifically, because many of such OS and hardware level functions are exposed to JavaScript via browser APIs, we can extract features when asking the browser to perform certain tasks through these APIs. The extracted features can be used for both single- and cross-browser fingerprinting.

†The author contributed to the paper when he was a REU student at Lehigh University.

NDSS '17, 26 February - 1 March 2017, San Diego, CA, USA

http://dx.doi.org/10.14722/ndss.2017.23152

[1]More details about our experiment can be found in Appendix A.

Let us take WebGL, a 3D component implemented in browser canvas object, for example. While canvas, especially the 2D part, has been used in single-browser fingerprinting [9, 32], WebGL is actually considered as "too brittle and unreliable" even for a single browser by a very recent study called AmIUnique [26]. The reason for such conclusion is that AmIUnique selects a random WebGL task and does not restrict many variables, such as canvas size and anti-aliasing, which affect the fingerprinting results.

Contrasting with this conclusion drawn by AmIUnique, we show that WebGL can be used not only for single- but also for cross-browser fingerprinting. Specifically, we ask the browser to render more than 20 tasks with carefully selected computer graphics parameters, such as texture, anti-aliasing, light, and transparency, and then extract features from the outputs of these rendering tasks.

Our principal contribution is being the *first* to use many novel OS and hardware features, especially computer graphics ones, in both single- and cross-browser fingerprinting. Particularly, our approach with new features can successfully fingerprint 99.24% of users as opposed to 90.84% for AmIUnique, i.e., state of the art, on the same dataset for single-browser fingerprinting. Moreover, our approach can achieve 83.24% uniqueness with 91.44% cross-browser stability, while Boda et al. [14] excluding IP address only have 68.98% uniqueness with 84.64% cross-browser stability.

Our secondary contribution is that we make several interesting observations for single- and cross-browser fingerprinting. For example, we find that the current measurement of screen resolution, e.g., the one done in AmIUnique, Panopticlick [5, 17] and Boda et al. [14], is unstable, because the resolution changes in Firefox and IE when the user zooms in or out the web page. Therefore, we take the zoom level into consideration, and normalize the width and height in screen resolution. For another example, we find that both DataURL and JPEG formats are unstable across different browsers, because these formats are with loss and implemented differently in multiple browsers and the server side as well. Therefore, we need to adopt lossless formats for server-client communications in cross-browser fingerprinting.

Our work is open-source and available at https://github.com/Song-Li/cross_browser/, and a working demo is at http://www.uniquemachine.org.

The rest of the paper is organized as follows. We first present all the features including old ones adopted and modified from AmIUnique and new ones proposed by us in Section II. Then, we introduce the design of our browser fingerprinting including the overall architecture, rendering tasks, and mask generation in Section III. After that, we talk about our implementation in Section IV, and data collection in Section V. We evaluate our approach and present the results in Section VI. Next, we discuss the defense of our fingerprinting in Section VII, some ethics issues in Section VIII, and related work in Section IX. Our paper concludes in Section X.

II. Fingerprintable Features

In this section, we introduce fingerprintable features used in this paper. We start from features used in prior works, and then introduce some features that need modification especially for cross-browser fingerprinting. Next, we present our newly-proposed features.

Although there are no restrictions for features on single-browser fingerprinting, our cross-browser features need to reflect the information and operation of the level below the browser, i.e., the OS and hardware level. For example, both vertex and fragment shaders expose the behaviors of GPU and its driver in the OS; the number of virtual cores is a CPU feature; the installed writing scripts are OS-level features. The reason is that these features in the OS and hardware level are relative more stable across browsers: all browsers are running on top of the same OS and hardware.

Note that if an operation, especially the outputs of the operation, is contributed by both the browser and the underlying (OS and hardware) levels, we can use it for single-browser fingerprinting, but need to get rid of the browser factor in cross-browser fingerprinting. For example, when we render an image as a texture on a cube, the texture mapping is an GPU operation but the image decoding is a browser one. Therefore, we can only use PNG, a lossless format, for cross-browser fingerprinting. For another example, the dynamic compression operation of audio signals is performed by both the browser and the underlying audio stack, and we need to extract the underlying features. Now let us introduce these features used in the paper.

A. Prior Fingerprintable Features

In this part of the section, we introduce fingerprintable features that we adopted from state of the art. There are 17 features presented in the Table I of the AmIUnique paper [26], and we have all of them for our single-browser fingerprinting. More detailed can be found in their paper. Because many of such features are browser specific, we adopt a subset with 4 features for cross-browser fingerprinting, namely screen resolution, color depth, list of fonts, and platform. Some of these features need modifications and are introduced below.

B. Old Features with Major Modifications

One prior feature, screen resolution, needs refactoring for both single- and cross-browser fingerprinting. Then, we introduce another fingerprintable feature, the number of CPU virtual cores. Lastly, two prior features need major modifications for cross-browser fingerprinting.

Screen Resolution. The current measurement of screen resolution is via the "screen" object under JavaScript. However, we find that many browsers, especially Firefox and IE, change the resolution value in proportion to the zoom level. For example, if the user enlarges the webpage with "ctrl++" in Firefox and IE, the screen resolution is inaccurate. We believe that the zoom level needs to be considered in both single- and cross-browser fingerprinting.

Specifically, we pursue two separate directions. First, we adopt existing work [13] on the detection of zoom levels based on the size of a div tag and the device pixel ratio, and then adjust the screen resolution correspondingly. Second, because the former method is not always reliable as acknowledged by the inventors, we adopt a new feature, i.e., the ratio between

screen width and height, which does not change with the zoom level.

In addition to screen resolution, we also find that some other properties, such as availHeight, availWidth, availLeft, availTop, and screenOrientation, are useful in both single- and cross-browser fingerprinting. The first four represents the available screens for the browser excluding system areas, such as the top menu and the tool bar of a Mac OS. The last one shows the position of the screen, e.g., whether the screen is landscape or portrait, and whether the screen is upside down.

Number of CPU Virtual Cores. The core number can be obtained by a new browser feature called hardwareConcurrency, which provides the capability information for Web Workers. Now, many browsers support such feature, but some, especially early versions of browsers, do not. If not supported, there exsits a side channel [1] to obtain the number. Specifically, one can monitor the finishing time of payload when increasing the number of web workers. When the finishing time increases significantly at a certain level of web workers, the limit of hardware concurrency is reached, making it useful to fingerprint the number of cores. Note that, some browsers, such as Safari, will cut the number available cores to Web Workers by half, and we need to double the number for cross-browser fingerprinting.

The number of cores is known by the inventor to be fingerprintable [2] and this is one of the reasons that they call it hardwareConcurrency rather than cores. However, the feature is never being used or measured in prior arts of browser fingerprinting.

AudioContext. AudioContext provides a bundle of audio signal processing functionalities from signal generation to signal filtering with the help of audio stack in the OS and the audio card. Specifically, existing fingerprinting work [18] uses OscillatorNode to generate a triangle wave, and then feed the wave into DynamicsCompressorNode, a signal processing module that suppresses loud sounds or amplifies quiet sounds, i.e., creating a compression effect. Then, the processed audio signal is converted to the frequency domain via AnalyserNode.

The wave in the frequency domain differs from one browser to another on the same machine. However, we find that peak values and their corresponding frequencies are relatively stable across browsers. Therefore, we create a list of bins with small steps on both the frequency and value axes, and map the peak frequencies and values to the corresponding bins. If one bin contains a frequency or value, we mark the bin as one and otherwise zero: such list of bins serve as our cross-browser feature.

In addition to the wave processing, we also obtain the following information from the destination audio device: sample rate, max channel count, number of inputs, number of outputs, channel count, channel count mode, and channel interpretation. Note that to the best of our knowledge, none of existing fingerprinting works have used such audio device information for browser fingerprinting.

List of Fonts. The measurement in AmIUnique is based on Flash plugin, however Flash is disappearing very fast, which is also mentioned and acknowledged in their paper. At the time of our experiment, Flash has already become little supported to obtain the font list. Instead, we adopt the side-channel method mentioned by Nikiforakis et al. [36], where the width and height of a certain string is measured to determine the font type. Note that not all fonts are cross-browser fingerprintable because some fonts are web specific and provided by browsers, and we need to apply a mask shown in Section III-C to select a subset. Another thing worth noting is that we are aware that Fifield et al. [20] provide a subset of 43 fonts for fingerprinting, however their work is based on single-browser fingerprinting and not applicable in our cross-browser scenario.

C. Newly-proposed Atomic Fingerprintable Features

In this and next subsection, we introduce our newly-proposed fingerprintable features. We first start with atomic features, and by atomic, we mean that the browser exposes either an API or a component directly to the JavaScript. Then, we will introduce composite features, which usually requires more than one API and component to collaborate.

Line, curve, and anti-aliasing. Line and curve are 2D features supported by both Canvas (2D part) and WebGL. Anti-aliasing is a computer graphics technique used to diminish aliasing by smoothing jaggies, i.e., jagged or stair-stepped lines, in either single line/curve object or the edge of a computer graphics model. There are many existing algorithms [4] for anti-aliasing, such as first-principles approach, signal processing approach, and mipmapping, which make anti-aliasing fingerprintable.

Vertex shader. A vertex shader, rendered by GPU and the driver, converts each vertex in a 3D model to its coordinate in a 2D clip-space. In WebGL, a vertex shader may accept data in 3 ways: attributes from buffers, uniforms that always stay the same, and texture from fragment shader. A vertex shader is usually combined with a fragment shader described below when rendering a computer graphics task.

Fragment shader. A fragment shader, rendered by GPU and the driver as well, processes a fragment, such as a triangle outputted by the rasterization, into a set of colors and a single depth value. In WebGL, fragment shader takes data in the following ways:

- *Uniforms.* A uniform value stays the same for every pixel in a fragment during a single draw call. Therefore, uniforms are non-fingerprintable features, and we list it here for completeness.
- *Varyings.* Varyings pass values from the vertex shader to the fragment shader that interpolates between these values and rasterizes the fragment, i.e., drawing each pixel in the fragment. The interpolation algorithm varies in different computer graphics cards, and thus varyings are fingerprintable.
- *Textures.* Given a setting of mapping between vertexes and texture, a fragment shader calculates the color of each pixel based on the texture. Due to the limited resolution of the texture, the fragment shader needs to interpolate values for a target pixel based on these pixels in the texture surrounded by the target. The texture interpolation algorithm also differs from one graphic card to another, making texture fingerprintable.

  Textures in WebGL can be further classified into several categories: (1) normal texture, i.e., the texture that we

introduced above; (2) depth texture, i.e., a texture that contains depth values for each pixel; (3) animating texture, i.e., a texture that contains video frames instead of static images; and (4) compressed texture, i.e., a texture that accepts compressed format.

Transparency via Alpha Channel. Transparency, a feature provided by GPU and the driver, allows the background to be intermingled with the foreground. Specifically, alpha channel with a value between 0 and 1 composites background and foreground images into a single, final one using a compositing algebra. There are two fingerprinting points in an alpha channel. First, we can use one single alpha value to fingerprint the compositing algorithm between background and foreground. Second, we can fingerprint the changes of transparency effects when the alpha value increases from 0 to 1. Because some graphics cards adopt discrete alpha values, some jumps may be observed in the changes of transparency effects.

Image encoding and decoding. Images can be encoded and compressed in different formats, such as JPEG, PNG, and DataURL. Some of the formats, such as PNG, are lossless, while some, such as JPEG, are compressed with loss of information. The decompression of a compressed images is a fingerprintable feature, because different algorithms may uncover different information during decompression. According to our study, this is a single-browser feature, and cannot be used for cross-browser.

Installed writing scripts (languages). Writing scripts (systems), or commonly known as written languages, such as Chinese, Korean, and Arabic, require the installation of special libraries to display due to the size of the libraries and locality of the languages. Browsers do not provide APIs to access the list of installed languages, however such information can be obtained via a side channel. Specifically, a browser with a particular language installed will display the language correctly, and otherwise show several boxes. That is, the existence of boxes can be used to fingerprint the presence of that language.

D. Newly-proposed Composite Fingeprintable Features

Now, let us introduce our newly-proposed composite fingeprintable features, which are rendered by more than one browser API or component, and sometimes with additional algorithms built atop of browser APIs.

Modeling and multiple models. Modeling, or specifically 3D modeling in this paper, is a computer graphics process of mathematically describing an object via three-dimensional surfaces. The vertexes of a model are handled by the vertex shader, and the surface by the fragment shader. Different objects are represented by different models, and may interact with each other especially when techniques below, such as lighting, exist.

Lighting and shadow mapping. Lighting is the simulation of light effects in computer graphics, and shadow mapping is to test whether a pixel is visible under a certain light and add corresponding shadows. There are many types of lighting, such as ambient lighting, directional lighting, and point lighting, which differ in the sources of the light. Additionally, many effects are accompanied by lights, such as reflection, translucency, light tracing, and indirect illumination, when lights interact with one computer graphics model or multiple models. WebGL does not provides direct APIs for lights and shadows, and some WebGL libraries (such as three.js) provides high-level APIs built on top of WebGL's vertex and fragment shaders for lights and shadows.

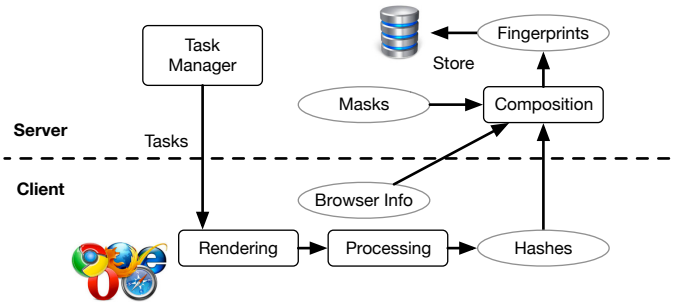

Fig. 1: System Architecture

Camera. Camera, or specifically pinhole camera model, maps 3D points in a space onto 2D points in an image. In WebGL, a camera is represented by a camera projection matrix handled by the vertex and fragment shaders, and can be used to rotate and zoom in and out an object.

Clipping Planes. Clipping restricts the rendering operations within a defined region of interest. In 3D rendering, a clipping plane is some distance away from and perpendicular to the camera so that it can prevent rendering surfaces that are too far from the camera. In WebGL, clipping planes are performed by the vertex and fragment shaders with additional provided algorithms.

III. Design

A. Overall Architecture

Figure 1 shows the system architecture. First, the task manager at the server side sends various rendering tasks, such as drawing curves and lines, to the client side. Note that the rendering tasks also involve obtaining OS and hardware level information, like screen resolution and timezone. Then, the client-side browser renders these tasks by invoking a specific API or a combination of APIs, and produces corresponding results, e.g., images and sound waves. Then, these results, especially images, are converted into hashes so that they can be conveniently sent to the server. Meantime, the browser also collects browser-specific information, such as whether anti-aliasing and compressed textures are supported, which will be used at the server side for fingerprints composition.

Next, when the server collects all the information from the client side, the server will start to composite fingerprints. Specifically, a fingerprint is generated from a list of hashes from the client side and a mask that is a list of one or zero corresponding to the hash list—we perform an "and" operation between the list of hashes and the mask, and then generate another hash as the fingerprint. The mask for single-browser fingerprinting is straightforward, a list of all ones. The mask for cross-browser fingerprinting is composited from two sources. First, the collected browser information will contribute to the mask: if the browser does not support anti-aliasing, the bit values in the mask for all tasks that involve anti-aliasing are zero. Second, we will have a different mask for each browser pair, e.g., Chrome vs. Firefox and Chrome vs. Windows Edge.

In the next two sections, we first introduce our rendering tasks at client side, and then our fingerprints composition, especially how to generate the masks.

B. Rendering Tasks

In this section, we introduce different rendering tasks proposed in this work. Before that, let us first present the basic canvas setting below. The size of the canvas is 256×256. The axes of the canvas are defined as follows. [0, 0, 0] is the middle of the canvas, where x-axis is the horizontal line that increases to the right, y-axis is the vertical line that increases to the bottom, and z-axis increases when moving far from the screen. An ambient light with the power of [R: 0.3, G: 0.3, B: 0.3] on a scale of 1 is present, and a camera is placed at the location of [0, 0, -7]. These two components are necessary, because otherwise the model is entirely black. In the rest of the paper, unless specified, such as Task (d) with 2D features and other tasks with additional lights, we use the same basic settings in all the tasks.

Note that unlike the settings in AmIUnique [26], our canvas setting is reliable when the condition of the current window changes. Specifically, we tested three different changes: window size, side bar, and zoom-level. First, we manually change the window size, and find that the contents in the canvas remain the same both visually and computationally in terms of hash value. Second, we zoom in and out the current window, and find that the contents change visually according to definition, but the hash value remain the same. Lastly, we open a browser console as a side bar, and find that the canvas contents also remain the same similar to changing window size. Now let us introduce our rendering tasks from Task (a) to (r).

Task (a): Texture. The task in Figure 2(a) is to test the regular texture feature in the fragment shader. Specifically, a classical Suzanne Monkey Head model [19] is rendered on a canvas with a randomly-generated texture. The texture, a square with a size as 256×256, is created by randomly picking a color for each pixel. That is, we generate three random values uniformly between 0 and 255 for three primary colors—red, green and blue—at one pixel, mix three primary colors together, and use it as the color for the pixel.

We choose this randomly-generated texture rather than a regular one, because this texture has more fingerprintable features. The reasons are as follows. When a fragment shader maps a texture to a model, the fragment shader needs to interpolate points in the texture so that the texture can be mapped to every point on the model. The interpolation algorithm differs from one graphic card to another, and the difference is amplified when the texture changes drastically in color. Therefore, we generate this texture in which colors change greatly between each pair of adjacent pixels.

Task (b): Varyings. This task, shown in Figure 2(b), is designed to test the varying feature of the fragment shader on a canvas. Different varying colors are drawn on six surfaces of a cube model with a specification of the color of four points on each surface. We choose this varying color to enlarge the color differences and changes on each single surface. For example, when blue is abundant (such as 0.9 with a scale of 1) on one vertex of a surface, the other vertex will lack blue (such as 0.1) and have more green or red color. Additionally, a camera is placed at the location of [0, 0, -5] for the purpose of comparison with Task (c).

Task (b'): Anti-aliasing+Varyings. The task in Figure 2(b') is to test the anti-aliasing feature, i.e., how browsers smooth the edge of models. Specifically, we adopt the same task in Task (b), and add anti-aliasing. If we enlarge Figure 2(b'), we will find that the edges of both models are smoothed.

Task (c): Camera. The task in Figure 2(c) is to test the camera feature, i.e., a projection matrix fed into the fragment shader. Every setting in this task is the same as Task (a) except for the camera, which is moved to a new location of [-1, -4, -10]. The same cube looks smaller than the one in Task (a), because the camera is moved further from the cube (the z-axis is -10 as opposed to -5).

Task (d): Lines and Curves. The task in Figure 2(d) is to test lines and curves. One curve and three lines with different angles are drawn on a canvas. Specifically, the curve obeys the following function: $y = 256 - 100cos(2.0\pi x/100.0) + 30cos(4.0\pi x/100.0) + 6cos(6.0\pi x/100.0)$, where [0, 0] is the left and top of the canvas, x-axis increases to the right, and y-axis increases to the bottom. The starting and ending points of three lines are {[38.4, 115.2], [89.6, 204.8]}, {[89.6, 89.6], [153.6, 204.8]}, and {[166.4, 89.6], [217.6, 204.8]}. We choose these specific lines and curves so that we can test different gradients and shapes.

Task (d'): Anti-aliasing+Lines and Curves. Task (d') is an anti-aliasing version of Task (d).

Task (e): Multi-models. The task in Figure 2(e) is to test how different models influence each other in the same canvas. In addition to the Suzanne model, we introduce another model that looks like a single-person armed sofa (called sofa model), and put two models in parallel. Another randomly-generated texture following the same procedure described in Task (a) is mapped to the sofa model.

Task (f): Light. The task in Figure 2(f) is to test the interaction of a diffuse, point light and the Suzanne model. A diffuse, point light causes diffuse reflection when illuminating an object. Specifically, the light is white with the same values across RGB, the power of the light is 2 for each primary color, and the light source is located at [3.0, -4.0, -2.0].

We choose a white light source in this task because the texture is colorful, and a single-color light may diminish some subtle differences on the texture. The power of the light is also carefully chosen, because a very weak light will not illuminate the Suzanne model, making it invisible, but a very strong light will make everything white and diminish all the fingerprintable features. In a small scale experiment with 6 machines, when increasing the power from 0 to 255, we find that when the light power is 2, the pixel differences among these machines are the maximum. The light position is randomly chosen and does not affect the feature fingerprinting results.

Task (g): Light and Models. The task in Figure 2(g) is to test the interaction of a single, diffuse, point light and two models, because one model may create a shadow on another when illuminated by a point light. Every setting of light is the same as Task (f), and the models are the same as Task (e).

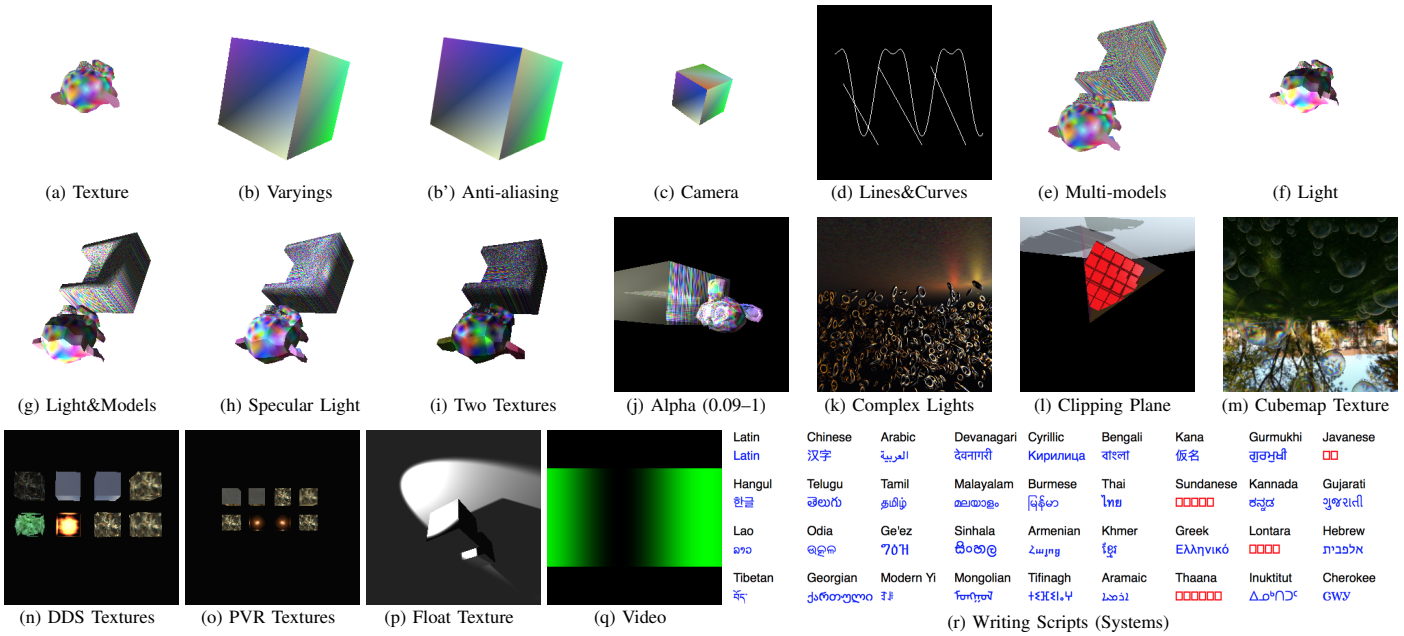

Fig. 2: Client-side Rendering Tasks for the Purpose of Fingerprinting

Task (h): Specular Light. The task in Figure 2(h) is to test the effects of a diffuse point light with another color and a specular point light on two models. Similar to diffuse point light, a specular point light will cause a specular reflection on an object. Specifically, both lights are located at [0.8, -0.8, -0.8], the RGB of the diffuse point light is [0.75, 0.75, 1.0], and the RGB of the specular light is [0.8, 0.8, 0.8].

There are two things worth noting. First, we choose the specific camera location because it is closer to the models and has bigger effects. Particularly, one may notice the spot on the back of the sofa model illuminated by the specular point light. Second, although the color of the diffuse point light is towards blue, but still has much red and green. We want to test other colors, but white light is still the best for fingerprinting given that the texture is colorful.

Task (h'): Anti-aliasing+Specular Light. Task (h') is an anti-aliasing version of Task (h).

Task (h"): Anti-aliasing+Specular Light+Rotation. Task (h') is the same as Task (h') but with 90 degree rotation.

Task (i): Two Textures. The task in Figure 2(i) is to test the effects of mapping two different textures to the same objects. On top of Task (h), i.e., every other setting is the same, we map another layer of randomly-generated texture to both the Suzanne and sofa model.

Task (j): Alpha. The task in Figure 2(j) consisted of 8 sub-tasks is to test the effects of different alpha values. Specifically, we put the Suzanne and sofa models in parallel, and change the alpha values chosen from this specific set, {0.09, 0.1, 0.11, 0.39, 0.4, 0.41, 0.79, 1}, where 0 means completely transparent and 1 no transparency.

Again, there are two things worth noting. First, we choose this value set carefully to reflect different alpha values and small value changes: three representative values {0.1, 0.4, 0.8} as well as their nearby values are selected. Values are augmented in 0.01, because many GPUs do not accept smaller steps. Second, the Suzanne and sofa models are positioned so that they are partially overlapped and the hidden structure of the sofa model is visible when the model becomes transparent. For example, the arm of the sofa model is partially visible when viewing from the back of the model.

Task (k): Complex Lights. The task in Figure 2(l) is to test complex light features, such as reflection, moving lights, and light tracing among multiple models. Specifically, we generate 5,000 metallic ring models with different angles randomly placed on the ground and piled together. For reliability, we use a seeded random number generator with the same random seed every time so that the test can be repeated on different browsers and machines. Two point light sources, yellow and red, towards the bottom are circling around in the right top corner of the entire scene. When lights illuminate the rings underneath, other rings also get illuminated through reflection and two colors from different sources are intermingled together.

Note that we choose single-color light sources because the models are not colorful, and lights with colors will illuminate more details on the rings. Furthermore, lights with different colors will interact with each other and create more detailed effects.

Task (k'): Anti-aliasing+Complex Lights. Task (k') is an anti-aliasing version of Task (k).

Task (l): Clipping Plane. The task in Figure 2(n) is to test the movement of a clipping plane and the FPS. Specifically, we put a static positive tetrahedron on the ground, illuminate it with collimated light, and move the clipping plane so that the observer feels that the tetrahedron is moving. The captured image in Figure 2(n) is upside down when the clipping plane moves to that position.

Task (m): Cubemap Texture+Fresnel Effect. The task in Figure 2(n) is to test cubemap texture and fresnel effect in

light reflection. Particularly, cubemap texture [7] is a special texture that utilizes the six faces of a cube as the map shape, and fresnel effect is an observation that the amount of reflected light depends on the viewing angle. We create a cubemap texture with a normal campus scene, and put several transparent bubbles on top of the texture for the fresnel effect. All the bubbles are moving randomly and bumping to each other in animation.

Task (n): DDS Textures. DDS Textures refer to those that use DirectDraw Surface file format, a special compressed data format with the S3 Texture Compression (S3TC) algorithm. There are five different variations of S3TC from DXT1 to DXT5, and each format has an option to enable mipmapping, a technique to scale high-resolution texture into multiple resolutions within the texture file. Because DXT2 is similar to DXT3 and DXT4 similar to DXT5, Task (p) only tests DXT1, DXT3, and DXT5 with and without mipmapping in each column as shown in Figure 2(p). For comparison, we also include an uncompressed texture with ARGB format in the rightmost column. There are two gray cubes in Figure 2(p) because DXT3 and DXT5 with mipmapping is unsupported on that specific machine.

Task (o): PVR Textures. PVR texture, or called PVRTC texture, is another texture compression format adopted mostly by mobile devices, such as all iPhone, iPod Touch, and iPad as well as some Android products. Based on the size of data blocks, there are two modes: 4 bit mode and 2 bit mode. Further, there are two popular versions, v1 and v3, and we can choose to enable mipmapping as well. In total, Task (q), shown in Figure 2(q), has 8 subtasks that enumerate different combinations of bit mode, version, and mipmapping. Similarly, a gray cube means that the format is not supported.

Task (p): Float Textures. Float texture, or called floating point texture, uses floating points instead of integers to represent color values. A special type of floating point texture is depth texture that contains the data from the depth buffer for a particular scene. Task (r), shown in Figure 2(r), is adopted from an existing online test [15] for the purpose of rendering float and depth textures.

Task (q): Video (Animating Textures). The task in Figure 2(s) is to test the decompression of videos. Specifically, we create a two-second static scene video from a PNG file with three different compression formats (namely WebM, high quality MP4, and standard MP4), maps the video as an animating texture to a cube, and capture six consecutive frames from the video.

Note that although all the videos are created with one single PNG file, the captured frames are different because the compression algorithm is with loss. We choose six consecutive frames because JavaScript only provides an API to obtain frames at a certain time but not with certain frame numbers—six consecutive frames can make sure that the target frame is within the set based on our experiment.

Task (r): Writing Scripts. The task in Figure 2(t) is to obtain the list of supported writing scripts, such as Latin, Chinese, and Arabic, in a browser. Because none of existing browsers provide an API to obtain the list of supported writing scripts, we adopt a side channel to test the existence of each writing script. Specifically, the method is as follows. The name of each writing script in its own language is rendered in the browser. If the writing script is supported, the rendering will succeed; otherwise, a set of boxes will be shown instead of the script. Therefore, we can detect the boxes to test whether the browser supports the script: For example, Figure 2(t) shows that Javanese, Sudanese, Lontara and Thaana are not supported in that specific tested browser. Our current test list has 36 writing scripts obtained from Wikipedia [8] and ranked by their popularity.

Algorithm 1 Cross-browser Mask Generation

Input:
M : the set of all possible masks.
$H_{browser,machine} = \{Hash_{task1}, Hash_{task2}, Hash_{task3}, ...\}$: the hash list for all the rendering tasks on one browser of a specific machine.
$H_{browser} = \{H_{browser,machine1}, H_{browser,machine2}, ...\}$: the hash list for a browser.
$HS = \{H_{chrome}, H_{firefox}, H_{opera}, ..\}$: the overall hash list.
Process:
for all possible $\{h_{browser1}, h_{browser2}\} \subset HS$ **do**
  $Max_{uniq} \leftarrow 0$
  $Max_{mask} \leftarrow null$
  for $mask$ in M **do**
    $FS \leftarrow \{\}$
    $Count \leftarrow 0$
    for $m_1 \in h_{browser1}$ and $m_2 \in h_{browser2}$ **do**
      if $m_1 \& mask == m_2 \& mask$ and $m_1 \& mask \notin FS$ **then**
        $Count++$
        $FS.add(m_1 \& mask)$
      end if
    end for
    $Uniq \leftarrow Count/size(h_{browser1})$
    if $Uniq > Max_{uniq}$ **then**
      $Max_{uniq} \leftarrow Uniq$
      $Max_{mask} \leftarrow Mask$
    end if
  end for
  Max_{mask} is the mask for browser 1 and 2.
end for

C. Fingerprints Composition

In this section, we present how to form a fingerprint at the server side based on the hashes from the client-side rendering tasks. As mentioned, a fingerprint is a hash computed from an "and" operation of the hash list of all the tasks and a mask. The mask is straightly all ones for single-browser fingerprinting, and computed from two sub-masks for cross-browser fingerprinting. We have talked about the first sub-mask computed from the fact whether a browser support certain functionalities in Section III-A, and now will discuss the second sub-mask, which differs for every browser pair.

The generation of the mask for every two browsers is a training-based approach. Specifically, we use a small subset to obtain a mask that optimizes both the cross-browser stability and the uniqueness. Note that similar to false positive and negative, these two numbers, i.e., cross-browser stability and uniqueness, are two sides of a coin: When the cross-browser stability increases, uniqueness decrease, and vice versa. Let us think about two extreme examples. If we use single-browser features, the cross-browser stability is zero but the uniqueness is the highest. At contrast, if we use only one feature, e.g., platform, the cross-browser stability is 100% but the uniqueness is very low.

Algorithm 1 shows the training procedure of the mask for every browser pair. We adopt a brute-force search: though not the most efficient but the most effective and complete. Due to

the small size of the training data, we realize that brute force is possible and produces the best result. Specifically, we first enumerate every browser pair (Line 1), and then every possible mask (Line 4). For each mask, we go through the training data (Line 7), and make sure to select the mask that maximizes the cross-browser stability multiplying the uniqueness (Line 8–11 and 14–17).

IV. Implementation

Our open-source implementation, excluding all the open-source libraries (e.g., three.js, a JavaScript 3D library, and glMatrix, a JavaScript library for matrix operations), has approximately 21K Lines of Code (LoC). Specifically, our approach involves approximately 14K lines of JavaScript, 1K lines of HTML, 2.4K lines of Coffeescript, 500 lines of C code, and 3.7K lines of Python code.

We now divide our code into client and server, and describe below. The client-side code has a manager in JavaScript that is generated from Coffeescript. The manager performs three jobs: (1) loading all the rendering tasks, (2) collecting all the results from the rendering tasks as well as browser information, and (3) sending the results to a snippet of JavaScript that performs hashes and then communicates with the server-side code. Tasks (n) and (o) are written in C and converted to JavaScript via Emscripten. All other rendering tasks are written in JavaScript directly: Tasks (k)–(m) are written with the help of three.js, and the rest tasks are directly using either WebGL or JavaScript APIs. All rendering tasks have used glMatrix for vector and matrix operations.

The server side of our implementation is written in Python, serving as a module of an Apache server. Our server-side code can be further divided into two parts: the first with 1.2K LoC for communicating with the client-side code and storing hashes into a database and images into a folder, and the second with 2.5K LoC for the analysis such as generating and applying masks on the collected fingerprints.

V. Data Collection

We collect data from two crowdsourcing websites, namely Amazon Mechanical Turks and MacroWorkers. Specifically, we instruct crowdsourcing workers to visit our website via two different browsers at their own choice, and if they visit the website via three browsers, they will get paid by a bonus. After visiting, our website will provide a unique code for each worker so that she can input it back to the crowdsourcing website to get paid and optional bonus. Note that in our data collection, in addition to hashes, we also send all the images data to the server—such a step is not needed if deploying our approach.

To ensure that we have the ground truth data, we insert a unique identifier as part of the URL that each crowdsourcing worker visits, e.g., http://oururl.com/?id=ABC. The unique identifier is stored at the client-side browser as a cookie so that if the user visits our website again, she will get the same identifier. Additionally, we allow one crowdsourcing worker to take the job only once. For example, the number of Human Intelligence Tasks (HITs) in MTurks is one for each worker.

In total, we have collected 3,615 fingerprints from 1,903 users within three months. Some users just visit our website with one browser and does not finish the two-browser task. We use all the fingerprints directly for single-browser fingerprinting. For cross-browser fingerprinting, the dataset is divided equally into ten parts for each browser pair if there is enough data: one for the generation of masks, and the other nine for testing.

TABLE I: Normalized Entropy for Six Attributes of the Dataset Collected by Our Approach, AmIUnique, and Panopticlick (The last two columns are copied from the AmIUnique paper)

	Ours	AmIUnique	Panopticlick
User Agent	0.612	0.570	0.531
List of Plugins	0.526	0.578	0.817
List of Fonts (Flash)	0.219	0.446	0.738
Screen Resolution	0.285	0.277	0.256
Timezone	0.340	0.201	0.161
Cookie Enabled	0.001	0.042	0.019

A. Comparing Our Dataset with AmIUnique and Panopticlick

The purpose of this part of the section is to compare our dataset with AmIUnique and Panopticlick in the metrics of normalized Shannon's entropy invented in the AmIUnique paper. Specifically, Equation 1 shows the definition according to their paper:

$$NH = \frac{H(X)}{H_M} = \frac{-\sum_i P(x_i) log_2 P(x_i)}{log_2(N)} \quad (1)$$

$H(X)$ is the Shannon's entropy where X is a variable with possible values $\{x_1, x_i, ..\}$ and P(X) a probability function. H_M is the worse case scenario in which every fingerprint has the same probability and we have the maximum entropy. N is the total number of fingerprints.

Table I shows the comparison result where the statistics for AmIUnique and Panopticlick are obtained from Table III of the AmIUnique paper. We observe that the normalized entropy values of our dataset are very similar to datasets used in past approaches except for list of fonts and timezone.

First, the normalized entropy of list of fonts drops 0.22 from AmIUnique and 0.52 from Panopticlick. The reason as explained by AmIUnique is that Flash is disappearing. By the time that we collect data, the percentage of browsers with Flash support decreases even more when compared with AmIUnique. To further validate our dataset, we also calculate the normalized entropy for the list of fonts collected by JavaScript. The value is 0.901, very close to the one from Panopticlick.

Second, the normalized entropy of timezone increases 0.139 from AmIUnique and 0.179 from Panopticlick. The reason is that our crowdsourcing workers from MicroWorkers are very international, spanning from Africa and Europe to Asia and Latin America. Specifically, MicroWorkers allow us to create campaigns targeting different regions all over the world, and we did create campaigns for each continental.

Another thing worth noting is that the normalized entropy of cookie enabled is almost zero for our dataset. The reason is that we collect data from crowdsourcing websites, where

TABLE II: Overall Results Comparing AmIUnique, Boda et al. excluding IP Address, and Our Approach ("Unique" means the percentage of unique fingerprints out of total, "Entropy" the Shannon entropy, and "Stability" the percentage of fingerprints that are stable across browsers. We do not list cross-browser number for AmIUnique and single-browser number for Boda et al. in the table, because these number are very low and their approaches are not designed for that purpose.)

	Single-browser		**Cross-browser**		
	Unique	Entropy	Unique	Entropy	Stability
AmIUnique [26]	90.84%	10.82			
Boda et al. [14]			68.98%	6.88	84.64%
Ours	99.24%	10.95	83.24%	7.10	91.44%

workers need to get paid with cookie enabled. If they disable cookies, they cannot even log into the crowdsourcing website. At contrast, both AmIUnique and Panopticlick attract general web users in which a small percentage may disable cookies. In general, there are very few people disabling cookies, because cookies are essential for many modern web functionalities.

VI. Results

In this section, we first give an overview of our results, and then break down the results by different browser pairs and features. Lastly, we present some interesting observation.

A. Overview

We first give an overview of our results for both single- and cross-browser fingerprinting. Specifically, we compare our single-browser fingerprinting with AmIUnique, state of the art, and our cross-browser fingerprinting with Boda et al. excluding IP address. Note that although many new features, e.g., these in AmIUnique, emerge after Boda et al., these features are browser specific and we find that the features used in Boda et al. are still the ones with the highest cross-browser stability.

We now introduce how we reproduce the results for these two works. AmIUnique is open-source [3], and we can directly download the source code from github. Boda et al. provides an open testing website (https://fingerprint.pet-portal.eu/), and we can download the fingerprinting JavaScript directly. We believe that the direct usage of their source code minimizes all the possible implementation biases.

The overall results of AmIUnique, Boda et al., and our approach are shown in Table II. Let us first take a look at single-browser fingerprinting. We compare our approach with AmIUnique in terms of uniqueness and entropy. Uniqueness means the percentage of unique fingerprints over the total number of fingerprints, and entropy is the Shannon entropy. The evaluation shows that our approach can uniquely identify 99.24% of users as opposed to 90.84% for AmIUnique, counting to 8.4% increase. For the entropy, the maximum value is 10.96, and both approaches, especially ours, are very close to the maximum. That is, non-unique fingerprints in both approaches are scattered in small anonymous groups.

Then, let us look at the metrics for cross-browser fingerprinting. In addition to uniqueness and entropy, we also calculate another metrics called cross-browser stability, meaning the percentage of fingerprints that are stable across different browsers on the same machine. Although we select features that are stable across browser most of time, fingerprints from different browsers might still differ. For example, screen resolutions could be different for Boda et al., if the user chooses different zoom levels in two browsers. For another example, GPU rendering might be different for our approach, if one browser adopts hardware rendering but another software rendering.

Now let us look at the cross-browser fingerprinting results for Boda et al. and our approach. Table II shows that our approach can identify 83.24% of users as opposed to 68.98% for Boda et al. This is a huge increase with 14.26% difference. The cross-browser stability also increases from 84.64% for Boda et al. to 91.44% for our approach. One of the reasons is that we make existing features, such as screen resolution and the list of fonts, more stable across different browsers. The entropy also increases from 6.88 for Boda et al. to 7.10 for our approach.

B. Breakdown by Browser Pairs

In this part of the section, we break down our results by different browser pairs shown in Table III. There are six different types of browsers, and a category called others including some uncommon browsers, such as Maxthon, Coconut, and UC browser. The table is a lower triangular matrix due to its symmetric property: If we list all the numbers, the upper triangle is exactly the same as the lower. The main diagonal of the table represents single-browser fingerprinting, and the other part cross-browser. There are two N/A because Apple gives up the support of Safari on Windows, and Microsoft never support Internet Explorer and Edge Browser on Mac OS, i.e., Safari does not co-exist with IE and Edge. There are two dashes as well for others and Edge/IE/Safari, because we do not observe any such pairs in our dataset.

Let us first look at the main diagonal. The stability for single browser is obviously 100% because we are comparing a browser to itself. The browser with lowest uniqueness is Mozilla Firefox, because Firefox hides some information, e.g., the WebGL render and vendor, for privacy reasons. The uniqueness for IE and Edge is 100%, showing that both browsers are highly fingerprintable. The uniqueness for Opera, Safari, and other browsers is also 100%, but due to the small number of samples in our dataset, we cannot draw further conclusions for these browsers.

Then, we look at the lower triangle of the matrix except the main diagonal, which shows the uniqueness and stability for cross-browser fingerprinting. First, the cross-browser stability for all pairs is very hight ($> 85\%$) except for other browsers and Opera vs. IE. Because the number of such pairs is small, it is hard for us to generate a mask with reasonable cross-browser stability.

Second, the uniqueness for IE and Edge vs. the rest is relatively low when compared with other pairs. The reason is that both IE and Edge are independently implemented by Microsoft with fewer open-source libraries. That is, the common part shared between IE/Edge and the rest is much less than these among the rest browsers. At contrast, the uniqueness between

TABLE III: Cross-browser Fingerprinting Uniqueness and Stability Break-down by Browser Pairs

Browser	Chrome	Firefox	Edge	IE	Opera	Safari	Other
Chrome	99.2% (100%)						
Firefox	89.1% (90.6%)	98.6% (100%)					
Edge	87.5% (92.6%)	97.9% (95.9%)	100% (100%)				
IE	85.1% (93.1%)	91.8% (90.7%)	100% (95.7%)	100% (100%)			
Opera	90.9% (90.0%)	100% (89.7%)	100% (100%)	100% (60.0%)	100% (100%)		
Safari	100% (89.7%)	100% (84.8%)	N/A	N/A	100% (100%)	100% (100%)	
Other	100% (22.2%)	100% (33.3%)	-	-	100% (50%)	-	100% (100%)

Note: The format of each cell is as follows – Uniqueness (Cross-browser Stability).

IE and Edge is very high: 100% uniqueness with 95.7% cross-browser stability, meaning that IE and Edge probably share a considerable amount of code.

Third, it is interesting to compare IE and Edge. The uniqueness of Edge Browser is higher than IE for all browser pairs. The reason is that Edge Browser introduces more functionalities, such as a full implementation of WebGL obeying the standard, which exposes more fingerprinting aspects.

C. Breakdown by Features

In this part of the section, we break down our results by different features and show it in Table IV. Specifically, Table IV can be divided into two parts: the first part above AmIUnique row showing the features adopted by AmIUnique, the second part below the first showing all the new features proposed by our approach. Now let us look at different features.

1) Screen Resolution and Ratio: The single-browser entropy for screen resolution and ratio is 7.41, while the entropy for the width and height ratio drops significantly to 1.40. The reason is that many resolutions, e.g., 1024×768 and 1280×960, share the same ratio. The cross-browser stability for screen resolution is very low (9.13%), because users often zoom in and out the web page as mentioned before. The cross-browser stability for the width and height ratio is high (97.57%) but lower than 100%, because some users adopt two screens and put two browsers in separate ones.

2) List of Font: Due to the ongoing disappearance of Flash, the entropy for the list of fonts obtained from Flash is as low as 2.40, and at contrast the entropy for the list from JavaScript is as high as 10.40. That means the list of fonts is still a highly fingerprintable feature, and we need to obtain the feature using JavaScript in the future.

Note that although the entropy for the font list from JavaScript is high, it does not take a significant portion in our fingerprinting. When we remove this feature, the single-browser uniqueness of our approach only drops from 99.24% to 99.09%, less than 0.2% difference. That is, our approach can still fingerprint users with high accuracy without the font list feature.

3) Anti-aliasing: Tasks (b), (b'), (d), (d'), (h), (h'), (k) and (k') are related to anti-aliasing. The entropy for single-browser fingerprinting increases for (b), (d) and (h) when anti-aliasing is added, but decreases for (k). The reason is that (b), (d) and (h) has fewer edges, and anti-aliasing will add more fingerprintable contents; at contrast, (k) contains many small edges on each of the beans, and anti-aliasing will occupy the contents of the beans and diminish some fingerprintable contents inside of the beans.

Now let us look at cross-browser fingerprinting. The cross-browser stability is the opposite of the single-browser entropy: it decreases for (b), (d) and (h), but increases for (k). The reason is that anti-aliasing is not supported for all browsers on the same machine, making the stability decrease for (b), (d) and (h). For similar reason, because anti-aliasing diminishes some fingerprintable contents inside the bean, the cross-browser stability increases for (k).

4) Line&Curves: Task (d) tests the effects of line and curves. The entropy is low (1.09) and the cross-browser stability is high (90.77%), because both lines and curves are simple 2D operations and do not differ too much across browsers and machines. We manually compare those cases that are different across machines or browsers, and find that the major difference lies in the starting and ending point where there are one or two pixels shifting.

5) Camera: When comparing the single-browser entropy for Task (b) and (c), we find that the entropy decreases when a camera is added. The reason is that the purpose of the added camera is to zoom out the cube, which diminishes subtle differences on the surface. The cross-browser stabilities for (b) and (c) are very similar due to the similarity between (b) and (c).

6) Texture: Let us first compare normal, DDS, PVR, cubemap and float textures. The entropies for float and cubemap textures are higher than all other textures, because float and cubemap textures have more information, e.g., the depth in float textures and a cube mapping for cubemap textures. The entropy for PVR textures is very low (0.14), because PVR textures are mostly supported on Apple mobile devices, such as iPhones and iPads. As our dataset is collected from crowdsourcing workers, very few of them will use Apple mobile devices to perform the crowdsourcing tasks. Another interesting observation is that the cross-browser stability for DDS textures is low (68.18%). The reason is that DDS, a Microsoft format, is unsupported on many browsers.

Second, let us look at two textures, i.e., Task (i). Compared with Task (h), another layer of texture is added, but the entropy for both single- and cross-browser fingerprinting decrease. The reason is that the texture used in our tasks is carefully created so that it can contain more fingerprintable features.

TABLE IV: Entropy and Cross-browser Stability by Features

Feature	Single-browser	Cross-browser	
	Entropy	Entropy	Stability
User agent	6.71	0.00	1.39%
Accept	1.29	0.01	1.25%
Content encoding	0.33	0.03	87.83%
Content language	4.28	1.39	10.96%
List of plugins	5.77	0.25	1.65%
Cookies enabled	0.00	0.00	100.00%
Use of local/session storage	0.03	0.00	99.57%
Timezone	3.72	3.51	100.00%
Screen resolution and color depth	7.41	3.24	9.13%
List of fonts (Flash)	2.40	0.05	68.00%
List of HTTP headers	3.17	0.64	9.13%
Platform	2.22	1.25	97.91%
Do Not Track	0.47	0.18	82.00%
Canvas	5.71	2.73	8.17%
WebGL Vendor	2.22	0.70	16.09%
WebGL Renderer	5.70	3.92	15.39%
Use of an Ad blocker	0.67	0.28	70.78%
AmIUnique	10.82	0.00	1.39%
Screen Ratio	1.40	0.98	97.57%
List of fonts (JavaScript)	10.40	6.58	96.52%
AudioContext	1.87	1.02	97.48%
CPU Virtual cores	1.92	0.59	100.00%
Normalized WebGL Renderer	4.98	4.01	37.39%
Task (a) Texture	3.51	2.26	81.47%
Task (b) Varyings	2.59	1.76	88.25%
Task (b') Varyings+anti-aliasing	3.24	1.66	73.95%
Task (c) Camera	2.29	1.58	88.07%
Task (d) Lines&Curves	1.09	0.42	90.77%
Task (d') (d)+anti-aliasing	3.59	2.20	74.88%
Task (e) Multi-models	3.54	2.14	81.15%
Task (f) Light	3.52	2.27	81.23%
Task (g) Light&Model	3.55	2.14	80.94%
Task (h) Specular light	4.44	3.24	80.64%
Task (h') (h)+anti-aliasing	5.24	3.71	70.35%
Task (h") (h')+rotation	4.01	2.68	75.09%
Task (i) Two textures	4.04	2.68	75.98%
Task (j) Alpha (0.09)	3.41	2.36	86.25%
Task (j) Alpha (0.10)	4.11	3.02	75.31%
Task (j) Alpha (0.11)	3.95	2.84	75.80%
Task (j) Alpha (0.39)	4.35	3.06	82.75%
Task (j) Alpha (0.40)	4.38	3.10	82.58%
Task (j) Alpha (0.41)	4.49	3.13	81.89%
Task (j) Alpha (0.79)	4.74	3.12	72.63%
Task (j) Alpha (1)	4.38	3.07	82.75%
Task (k) Complex lights	6.07	4.19	66.37%
Task (k') (k)+anti-aliasing	5.79	3.96	74.45%
Task (l) Clipping plane	3.48	1.93	76.61%
Task (m) Cubemap texture	6.03	3.93	58.94%
Task (n) DDS textures	4.71	3.06	68.18%
Task (o) PVR textures	0.14	0.00	99.16%
Task (p) Float texture	5.11	3.63	74.41%
Task (q) Video	7.29	2.32	5.48%
Task (r) Writing scripts (support)	2.87	0.51	97.91%
Task (r) Writing scripts (images)	6.00	1.98	5.48%
All cross-browser features	10.92	7.10	91.44%
All features	10.95	0.00	1.39%

When we add two textures together, some of these features are diminished, making two-texture task less fingerprintable.

7) Model: Let us compare Tasks (a) and (e) as well as Tasks (f) and (g) for the effect of models. Compared to (a) and (f), a sofa model is added to (e) and (g), and the entropy increases a little bit, i.e., 0.03 for both tasks. The conclusion is that the Sofa model does introduce more fingerprintable features but the increase is very limited.

8) Light: Tasks (a), (e), (f), (h), and (k) are related to lights. Let us first look at Task (f) in which a diffuse, point light is added to Task (a). The entropy only increases 0.01 for both single- and cross-browser fingerprinting, showing that the diffuse, point light has little impact in fingerprinting. As a comparison, the effect of a specular light is more apparent because the entropy for Task (h) is an increase of >0.9 when compared to Task (e) in both single- and cross-browser fingerprinting. Lastly, let us look at Task (k), a complex light example. The entropy for Task (k) is the highest among all tasks except for video, because there are 5,000 models and lights with different colors are reflected among all the models and intermingled together.

9) Alpha: Task (j) tests alpha values from 0.09 to 1. It is interesting that different alpha values have very different entropies. In general, the trend is that when the alpha value increases, the entropy increases as well but with many fallbacks. We did not test continuous alpha values in our large-scale experiment, but perform a small-scale one among five machines. Specifically, we compare the differed pixels between each Alpha value image and a standard one, and find that the fallbacks are mainly caused by software rendering, which approximates alpha values. Additionally, we observe some patterns in the fallbacks, which happens in an approximate 0.1 incremental step.

10)Clipping Planes: Task (l) is to test the effect of clipping planes, yielding 3.48 single-browser entropy and 1.93 cross-browser entropy with 76.61% stability. The entropy is similar to the one with pure texture, because clipping planes are implemented in JavaScript and do not contribute to fingeprinting much.

11) Rotation: Task (h") is a rotation of Task (h'). The entropy decreases and the cross-browser stability increases. The reason is that the front of the Suzanne model and the inside of the sofa model has more details. When we rotate both models to another angle, the fingerprintable details decreases and correspondingly the stability increases.

12) AudioContext: The AudioContext that we measure is the cross-browser stable one, i.e., the destination audio device information and the converted waves. The entropy is 1.87, much smaller than the entire entropy of the entire wave—which is 5.4 as measured by Englehardt et al. [18].

13)Video: Task (q) is testing the video feature. The entropy for video is the highest (7.29) among all of rendering tasks, because decoding video is a combination of the browser, the driver, and sometimes the hardware as well. At contrast, the cross-browser stability for video is very low (5.48%) and the entropy also drops to 2.32. The reason is that similar to image encoding and decoding, both WebM and MP4 video formats are with loss and decoded by the browser. We do not find a universal lossless format for videos as we do for images.

14) Writing Scripts: Writing scripts are tested in Task (r). We further divide Task (r) into two parts for the purpose of cross-browser fingerprinting. The first part, we call it writing scripts (support), only contains the information of whether certain writing scripts are supported, i.e., a list of zeros and ones where one means supported and zero not. As mentioned, we obtain the information via box detection. The second part, we call it writing scripts (images), is the images rendered at the client-side. The single-browser entropy for writing scripts (images) is 3.13 larger than the one for writing scripts (support). That is, the images do contain more information than whether the writing scripts are supported. The cross-browser stability for writing scripts (support) is calculated based on the results after applying our mask, because some writing scripts are shipped with the browser and not cross-browser stable. Correspondingly, the cross-browser entropy for writing scripts (support) is lower than the single-browser one.

15) CPU Virtual Cores: The number of CPU virtual cores, calculated from the HardwareConcurrency value only (if not supported, the value is "undefined"), has an entropy of 1.92 for single-browser fingerprinting. We expect that the entropy will increase in the future, because just before our submission, Firefox 48 starts to support the new feature. The cross-browser stability is 100%, because we can detect whether a browser supports HardwareConcurrency and applies a customized mask. The cross-browser entropy is different from the single-browser one due to the size of data, and the normalized entropies for both are very similar.

16) Normalized WebGL Renderer: The WebGL renderer is not cross-browser fingerprintable, partly because different browsers provide different levels of information. We extract the common information from different browsers, and align the information in a standard format. Compared with the original WebGL renderer with 5.70 entropy, the entropy for the normalized one is 4.98. The reason for the drop is that the extraction will discard some information, e.g., for Chrome, to align with other browsers, e.g., Edge browser. Correspondingly, the cross-browser stability increases from 15.39% for the original WebGL renderer to 37.39% for the normalized one.

There are two things worth noting here. First, the WebGL vendor does not provide more information than the WebGL renderer. That is, when we combine both values together, the entropy is the one for WebGL renderer. Second, our GPU tasks have much more information than the one provided by WebGL vendor and renderer. Some browsers, namely Firefox, do not provide WebGL vendor and renderer information, which gives us much room to fill the gap. Furthermore, even when a browser provide such information, the entropy for our GPU tasks when combined together is 7.10, much larger than the 5.70 entropy provided by WebGL render. The reason is that the rendering is a combination of software and hardware, and WebGL renderer only provides the hardware information for hardware rendering.

D. Observations

During our experiments and implementations, we have observed several interesting facts and shown them below in this subsection:

Observation 1: Our fingerprintable features are highly reliable, i.e., the removal of one single feature has little impact on the fingerprinting results.

In this part, we show the impact of removing a single feature from both AmIUnique and our approach, and then measure the uniqueness of both. The results show that the uniqueness of our fingerprinting is still above 99% when removing any single features in Table IV including all the old ones from AmIUnique and our new ones. At contrast, the uniqueness for AmIUnique drops below 84% if removing any single one of the following six attributes, namely user agent, timezone, list of plugins, content language, list of HTTP headers, and screen resolution and color depth. In sum, our approach is more reliable than AmIUnique in terms of used features.

Observation 2: Software rendering can also be used for fingerprinting.

One common understanding for WebGL is that software rendering may diminish all the differences caused by the graphic cards. However, our experiment shows that even software rendering can be used for fingerprinting. Specifically, we select all the data where WebGL is rendered by SwiftShader, an open source software renderer invented by Google and used by Chrome when hardware rendering is unavailable. We calculate a special fingerprint only containing all our GPU rendering tasks, i.e., Task (a)–(p) excluding writing scripts and video.

Due to the high adoption of hardware rendering, we only collect 88 cases using SwiftShader and find 11 distinct GPU fingerprints with 7 unique ones. The uniqueness of software rendering is definitely much lower than the one of hardware rendering but still not zero. That is, we need to be careful when adopting software rendering to mitigate WebGL-based fingerprinting.

Observation 3: WebGL rendering is a combination of software and hardware in which the hardware contributes more than the software.

In this observation, we look at another extreme compared to software rendering, which is Microsoft Basic Rendering. Microsoft Basic Rendering provides a universal driver for all kinds of graphic cards, i.e., the use of Microsoft Basic Rendering will minimize the effects of software driver and show the ones brought by the hardware. Similar to the experiment for software rendering, we select these that use Microsoft Basic Rendering and calculate the fingerprints.

For similar reasons in software rendering, we only collect 32 cases using Microsoft Basic Rendering and find 18 distinct GPU fingerprints with 15 unique values. The uniqueness of Microsoft Basic Rendering is lower than the one using normal graphic card drivers, meaning that WebGL is rendered by both software and hardware. Meanwhile, we consider hardware makes more contributions, because the uniqueness for Microsoft Basic Rendering is higher than the one for the software renderer.

Observation 4: DataURL is implemented differently across browsers.

In this observation, we look at DataURL, a common format used in prior fingerprinting to represent images. Surprisingly, we find that DataURL is implemented very differently in browsers, i.e., if we convert an image into DataURL, the representation varies a lot across browsers. This is a good news for single-browser fingerprinting but bad for cross-browser. As shown in Table IV, the cross-browser rate for Canvas is very low (8.17%), because we adopt the code from AmIUnique where DataURL is used to store images.

Observation 5: Some differences between rendering results are very subtle, i.e., with one or two pixel variance.

In this last observation, we manually compare the differences between rendering results, and find that while some of them are large, especially between software and hardware rendering, some are very subtle, especially when two graphic cards are similar to each other. For example, the Suzanne model rendered by an iMac and another Mac Pro only differs one pixel on the texture, and if we rotate the model, the difference will be gone.

VII. Defense of the Proposed Fingerprinting

In this section, we discuss how to defend our proposed browser fingerprinting. We will first start from existing defense, the famous Tor browser, and then come to some visions of our defense.

Tor Browser normalizes many browser outputs to mitigate existing browser fingerprinting. That is, many features are unavailable in Tor Browsers—based on our test, only the following features, notably our newly proposed, still exist, which include the screen width and height ratio, and audio context information (e.g., sample rate and max channel count). We believe that it is easy for Tor Browser to normalize these remaining outputs.

Another thing worth mentioning is that Tor Browser disables canvas by default, and will ask users to allow the usage of canvas. If the user does allow canvas, she can still be fingerprinted. The Tor Browser document also mentions a unimplemented software rendering solution, however as noted in Section VI-D, the outputs of software rendering also differ significantly in the same browser. We still believe that this is the way to pursue, but more careful analysis is needed to include all the libraries of software rendering.

Overall, the idea of defending browser fingerprinting can be generalized as virtualization, and we need to find a correct virtualization layer. Think about one extreme solution, which is a browser running inside a virtual machine—everything is normalized in the virtual machine, and the browser outputs are the same across different physical machines. However, the drawback is that machine virtualization is heavyweight. Tor browser is another extreme—everything is virtualized as part of a browser. This approach is lightweight, but we need to find all possible fingerprintable places, such as canvas and audio context: If one place is missing, the browser can still be somehow fingerprinted. We leave it as our future work to explore the correct virtualization layer.

VIII. Discussions on Ethics Issues

We have discussed ethics issues with the institutional review board (IRB) of our organization, and obtained the IRB approval. Specifically, although web tracking can be used to acquire private information, the identifiers that we obtain from crowdsourcing workers, e.g., the behaviors of computer graphics cards, are not private themselves. Only when the identifiers are associated with private information, such as browsing history, the combination is considered as private—however, this step is out of scope of the research. Our survey part, i.e., the study about the statistics of multiple browser usage in the Appendix A, contains users' browsing habits. In order to ensure privacy, the survey is anonymized and we do not store user ID from MicroWorkers.

IX. Related Work

In this section, we discuss related work on existing web tracking and anti-tracking techniques.

A. Web Tracking Techniques and Measurement

We first talk about the first generation tracking, i.e., cookie or super-cookie based, and then the second generation, browser fingerprinting.

1) Cookie or Super-cookie based Tracking: There is much existing work focusing on the measurement or study of cookie or super-cookie based web tracking techniques. Mayer et al. [28] and Sanchez et al. [40] conduct comprehensive discussions about third-party tracking, including tracking techniques, business models, defense choices and policy debates. Another important measurement work from Roesner et al. proposes a comprehensive classification framework for different web tracking deployed in real-world websites [39]. Lerner et al. conduct an archaeological study of web tracking, including cookie and super-cookie based as well as browser fingerprinting, from 1996 to 2016 [27]. Soltani et al. and Ayenson et al. measure the prevalence of non-cookie based stateful tracking and show how tracking companies use multiple client-side states to regenerate deleted identifiers [11, 41]. Metwalley et al. [30] propose an unsupervised measurement of web tracking. In addition to tracking behaviors and techniques, Krishnamurthy et al. [22–25] focus on the risk of harm resulted from web tracking, showing that not only user's browsing history, but also other sensitive personal information, such as name and email, can be leaked out.

2) Browser Fingerprinting: Now let us discuss browser fingerprinting, the second-generation web tracking. We first talk about existing measurement studies. Yen et al. and Nikiforakis et al. discuss different second-generation tracking techniques used in existing fingerprinting tools and their effectiveness in their works [36, 46]. Acar et al. [9] perform a large-scale study of three advanced web tracking mechanisms, one on second-generation web tracking, i.e., canvas fingerprinting, and the other two staying on the first-generation web tracking, i.e., evercookies and use of "cookie syncing" in conjunction with evercookies. Fifield el al. [20] focus on a specific metric, i.e., the font, of second-generation web tracking. FPDetective [10] conducts a large-scale study of millions of most popular websites by focusing on the font detection with their framework. Englehardt et al. [18] also conduct a large-scale study on 1

million websites and find many new fingerprinting features, such as AudioContext. We have used their newly discovered fingerprinting features as part of prior ones in Section II of our paper as well.

Now let us talk about browser fingerprinting works. Mowery et al. [32] are probably one of the very early works in proposing canvas-based fingerprinting. Some other works [31, 33] focus on fingerprinting browser JavaScript engine. Nakibly et al. [34], a position paper, propose several hardware-based tracking including microphone, motion sensor and GPU. Their GPU tracking only includes timing-based features, less reliable than the technique in the paper. Laperdrix et al. [26], i.e., AmIUnique, perform a most extensive study on browser fingerprinting with 17 attributes and we have compared with them throughout our paper. Boda et al. [14] attempts to achieve cross-browser tracking, but their features are old ones from single-browser tracking including IP address. As discussed, IP addresses are unreliable when a machine is using a DHCP, behind a NAT, or moved to a new location like a laptop.

As a general comparison with existing works, our approach introduces many new features on the OS and hardware levels. For example, we introduce many GPU features such as textures, varyings, lights and models. For another example, we also introduce a side channel to detect installed writing scripts and some new information in AudioContext. All these new features contribute to our high fingerprinting uniqueness and cross-browser stability.

B. Existing Anti-tracking Mechanisms

We first talk about existing anti-tracking for the first-generation tracking, and then for the second.

1) Anti-tracking against Cookie or Super-cookie based Techniques: Roesner et al. [39] proposed a tool called ShareMeNot, defending social media button tracking, such as Facebook Like button. Private browsing mode [44, 45] isolates normal browsing from private ones with a separate user profile. Similarly, TrackingFree [37] adopts the profile-based isolation and proposes an indegree-bounded graph for the profile creation. The Do Not Track (DNT) [43] header is a opt-out approach, which requires tracker compliance. As shown by prior works [28, 39], DNT cannot effectively protect users from tracking in real world. Users can also disable third-party cookie, which is supported by most browsers to avoid cookie-based tracking. Meng et al. [29] design a policy and empower users to control whether to be tracked, but they have to rely on an existing anti-tracking technique.

All the aforementioned works focus on cookies or super-cookie based web tracking, and can either fully or partially prevent such tracking. None of them can prevent the proposed fingerprinting in this paper, because the proposed belongs to the second generation, which does not require a server-side, stateful identifier.

2) Anti-tracking against Browser Fingerprinting: Tor Browser [38] can successfully defend many browser fingerprinting techniques, including features proposed in our paper. Please refer to Section VII for more details. Other than the normalization technique proposed in Tor Browser, PriVaricator [35] adds randomized noise to fingerprint-able outputs. Because PriVaricator is not open source, we could not test our fingerprinting against their defense.

X. Conclusion

In conclusion, we have proposed a novel browser fingerprinting that can identify not only users behind one browser but also these that use different browsers on the same machine. Our approach adopts OS and hardware levels features including graphic cards exposed by WebGL, audio stack by AudioContext, and CPU by hardwareConcurrency. Our evaluation shows that our approach can uniquely identify more users than AmIUnique for single-browser fingerprinting, and than Boda et al. for cross-browser fingerprinting. Our approach is highly reliable, i.e., the removal of any single feature only decreases the accuracy by at most 0.3%.

Acknowledgement

The authors would like to thank anonymous reviewers for their thoughtful comments. This work is supported in part by U.S. National Science Foundation (NSF) under Grants CNS-1646662 and CNS-1563843. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSF.

References

[1] Core estimator. https://github.com/oftn-oswg/core-estimator.

[2] [email threads] proposal: navigator.cores. https://lists.w3.org/Archives/Public/public-whatwg-archive/2014May/0062.html.

[3] [github] Am I Unique? https://github.com/DIVERSIFY-project/amiunique.

[4] [graphics wikia] anti-aliasing. http://graphics.wikia.com/wiki/Anti-Aliasing.

[5] Panopticlick: Is your browser safe against tracking? https://panopticlick.eff.org/.

[6] Watched: A wall street journal privacy report. http://www.wsj.com/public/page/what-they-know-digital-privacy.html.

[7] [wikipedia] cube mapping. https://en.wikipedia.org/wiki/Cube_mapping.

[8] [wikipedia] list of writing systems. https://en.wikipedia.org/wiki/List_of_writing_systems.

[9] G. Acar, C. Eubank, S. Englehardt, M. Juarez, A. Narayanan, and C. Diaz, "The web never forgets: Persistent tracking mechanisms in the wild," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14, 2014, pp. 674–689.

[10] G. Acar, M. Juarez, N. Nikiforakis, C. Diaz, S. Gürses, F. Piessens, and B. Preneel, "FPDetective: Dusting the web for fingerprinters," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '13, 2013, pp. 1129–1140.

[11] M. Ayenson, D. Wambach, A. Soltani, N. Good, and C. Hoofnagle, "Flash cookies and privacy ii: Now with html5 and etag respawning," *Available at SSRN 1898390*, 2011.

[12] S. Berger. You should install two browsers. http://www.compukiss.com/internet-and-security/you-should-install-two-browsers.html.

[13] T. Bigelajzen. Cross browser zoom and pixel ratio detector. https://github.com/tombigel/detect-zoom.

[14] K. Boda, A. M. Földes, G. G. Gulyás, and S. Imre, "User tracking on the web via cross-browser fingerprinting," in *Proceedings of the 16th Nordic Conference on Information Security Technology for Applications*, ser. NordSec'11, 2012, pp. 31–46.

[15] F. Boesch. Soft shadow mapping. http://codeflow.org/entries/2013/feb/15/soft-shadow-mapping/.

TABLE V: Statistics of Browser Usage

Single browser	>2 browsers	>3 browsers	Chrome& Firefox	Chrome& Microsoft IE/Edge
30%	70%	13%	33%	20%

[16] F. T. Commission. Cross-device tracking. https://www.ftc.gov/news-events/events-calendar/2015/11/cross-device-tracking.

[17] P. Eckersley, "How unique is your web browser?" in *Proceedings of the 10th International Conference on Privacy Enhancing Technologies*, ser. PETS'10, 2010.

[18] S. Englehardt and A. Narayanan, "Online tracking: A 1-million-site measurement and analysis," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16, 2016.

[19] A. Etienne and J. Etienne. Classical suzanne monkey from blender to get your game started with threex.suzanne. http://learningthreejs.com/blog/2014/05/09/classical-suzanne-monkey-from-blender-to-get-your-game-started-with-threex-dot-suzanne/.

[20] D. Fifield and S. Egelman, "Fingerprinting web users through font metrics," in *Financial Cryptography and Data Security*. Springer, 2015, pp. 107–124.

[21] S. Kamkar. Evercookie. http://samy.pl/evercookie/.

[22] B. Krishnamurthy, K. Naryshkin, and C. Wills, "Privacy leakage vs. protection measures: the growing disconnect," in *Web 2.0 Security and Privacy Workshop*, 2011.

[23] B. Krishnamurthy and C. Wills, "Privacy diffusion on the web: a longitudinal perspective," in *Proceedings of the 18th international conference on World wide web*. ACM, 2009, pp. 541–550.

[24] B. Krishnamurthy and C. E. Wills, "Generating a privacy footprint on the internet," in *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*. ACM, 2006, pp. 65–70.

[25] ——, "Characterizing privacy in online social networks," in *Proceedings of the first workshop on Online social networks*. ACM, 2008, pp. 37–42.

[26] P. Laperdrix, W. Rudametkin, and B. Baudry, "Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints," in *37th IEEE Symposium on Security and Privacy (S&P 2016)*, 2016.

[27] A. Lerner, A. K. Simpson, T. Kohno, and F. Roesner, "Internet jones and the raiders of the lost trackers: An archaeological study of web tracking from 1996 to 2016," in *25th USENIX Security Symposium (USENIX Security 16)*, Austin, TX, 2016.

[28] J. R. Mayer and J. C. Mitchell, "Third-party web tracking: Policy and technology," in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 413–427.

[29] W. Meng, B. Lee, X. Xing, and W. Lee, "Trackmeornot: Enabling flexible control on web tracking," in *Proceedings of the 25th International Conference on World Wide Web*, ser. WWW '16, 2016, pp. 99–109.

[30] H. Metwalley and S. Traverso, "Unsupervised detection of web trackers," in *Globecom*, 2015.

[31] K. Mowery, D. Bogenreif, S. Yilek, and H. Shacham, "Fingerprinting information in javascript implementations," 2011.

[32] K. Mowery and H. Shacham, "Pixel perfect: Fingerprinting canvas in html5," 2012.

[33] M. Mulazzani, P. Reschl, M. Huber, M. Leithner, S. Schrittwieser, E. Weippl, and F. Wien, "Fast and reliable browser identification with javascript engine fingerprinting," in *W2SP*, 2013.

[34] G. Nakibly, G. Shelef, and S. Yudilevich, "Hardware fingerprinting using html5," *arXiv preprint arXiv:1503.01408*, 2015.

[35] N. Nikiforakis, W. Joosen, and B. Livshits, "Privaricator: Deceiving fingerprinters with little white lies," in *Proceedings of the 24th International Conference on World Wide Web*, ser. WWW '15, 2015, pp. 820–830.

[36] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, "Cookieless monster: Exploring the ecosystem of web-based device fingerprinting," in *IEEE Symposium on Security and Privacy*, 2013.

[37] X. Pan, Y. Cao, and Y. Chen, "I do not know what you visited last summer - protecting users from third-party web tracking with trackingfree browser," in *NDSS*, 2015.

[38] M. Perry, E. Clark, and S. Murdoch, "The design and implementation of the tor browser [draft][online], united states," 2015.

[39] F. Roesner, T. Kohno, and D. Wetherall, "Detecting and defending against third-party tracking on the web," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'12, 2012, pp. 12–12.

[40] I. Sánchez-Rola, X. Ugarte-Pedrero, I. Santos, and P. G. Bringas, "Tracking users like there is no tomorrow: Privacy on the current internet," in *International Joint Conference*. Springer, 2015, pp. 473–483.

[41] A. Soltani, S. Canty, Q. Mayo, L. Thomas, and C. J. Hoofnagle, "Flash cookies and privacy," in *AAAI Spring Symposium: Intelligent Information Privacy Management*, 2010.

[42] US-CERT. Securing your web browser. https://www.us-cert.gov/publications/securing-your-web-browser.

[43] Wikipedia. Do Not Track Policy. http://en.wikipedia.org/wiki/Do_Not_Track_Policy.

[44] ——. Privacy Mode. http://en.wikipedia.org/wiki/Privacy_mode.

[45] M. Xu, Y. Jang, X. Xing, T. Kim, and W. Lee, "Ucognito: Private browsing without tears," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15, 2015, pp. 438–449.

[46] T.-F. Yen, Y. Xie, F. Yu, R. P. Yu, and M. Abadi, "Host fingerprinting and tracking on the web: Privacy and security implications," in *Proceedings of NDSS*, 2012.

APPENDIX A
SURVEY OF PEOPLE'S USAGE OF MULTIPLE BROWSERS

In this appendix, we study the statistics of people who use multiple browsers on the same machine. Note that this is a small-scale, separate study from all other designs and experiments of the paper. We perform the study to strengthen the motivation of the paper. Our results show that people do use more than one browser on the same machine with a considerable amount of time.

Now let us introduce our experiment setup on MicroWorkers, a crowdsourcing website. We conduct a survey with an open question that ask survey takers which browser(s) they have and normally use as well as how much time in terms of percentage they spend on each browser. They are free to write anything into a multiple-line text box.

Here are our experiment results. We have collected 102 answers with one answer just copying our survey link and another mentioning a browser that does not exist. After excluding these two invalid answers, we have exactly 100 in total. 95% of the surveyed users have installed more than two browsers because IE or Edge are installed by default. We further count the percentage of them using two or more browser regularly, i.e., they spend at least more than 5% time on one of the browser.

The results of people using browsers are shown in Table V. 70% of the surveyed takers use two or more browsers regularly, and only 30% use a single browser. Browser types in the survey answers include Chrome, Firefox, IE, Edge, Safari, Coconut Browser, and Maxthon. The results show that people do use multiple browsers, and cross-browser fingerprinting is important and necessary.