

Rendered Private: Making GLSL Execution Uniform to Prevent WebGL-based Browser Fingerprinting

Shujiang Wu, Song Li, Yinzhi Cao, and Ningfei Wang^{†*}
Johns Hopkins University, [†]Lehigh University
{swu68, lsong18, yinzhi.cao}@jhu.edu, wangningfei7@gmail.com

Abstract

Browser fingerprinting, a substitute of cookies-based tracking, extracts a list of client-side features and combines them as a unique identifier for the target browser. Among all these features, one that has the highest entropy and the ability for an even sneakier purpose, i.e., cross-browser fingerprinting, is the rendering of WebGL tasks, which produce different results across different installations of the same browser on different computers, thus being considered as fingerprintable.

Such WebGL-based fingerprinting is hard to defend against, because the client browser executes a program written in OpenGL Shading Language (GLSL). To date, it remains unclear, in either the industry or the research community, about how and why the rendering of GLSL programs could lead to result discrepancies. Therefore, all the existing defenses, such as these adopted by Tor Browser, can only disable WebGL, i.e., a sacrifice of functionality over privacy, to prevent WebGL-based fingerprinting.

In this paper, we propose a novel system, called UNIGL, to rewrite GLSL programs and make uniform WebGL rendering procedure with the support of existing WebGL functionalities. Particularly, we, being the first in the community, point out that such rendering discrepancies in state-of-the-art WebGL-based fingerprinting are caused by floating-point operations. After realizing the cause, we design UNIGL so that it redefines all the floating-point operations, either explicitly written in GLSL programs or implicitly invoked by WebGL, to mitigate the fingerprinting factors.

We implemented a prototype of UNIGL as an open-source browser add-on (<https://www.github.com/unigl/>). We also created a demo website (<http://test.unigl.org/>), i.e., a modified version of an existing fingerprinting website, which directly integrates our add-on at the server-side to demonstrate the effectiveness of UNIGL. Our evaluation using crowdsourcing workers shows that UNIGL can prevent state-of-the-art WebGL-based fingerprinting with reasonable FPSes.

^{*}The last author, Ningfei Wang, contributed to the paper when he was a master student financially supported and mentored by Dr. Yinzhi Cao.

1 Introduction

Browser fingerprinting [12, 13, 20, 23, 34, 45, 63], a substitute of traditional cookie-based approaches, is recently widely adopted by many real-world websites to track users' browsing behaviors potentially without their knowledge, leading to a violation of user privacy. In particular, a website performing browser fingerprinting collects a vector of browser-specific information called browser fingerprint, such as user agent, a list of browser plugins, and installed browser fonts, to uniquely identify the target browser.

Among all the possible fingerprintable vectors, the rendering behavior of WebGL, i.e., a Web-level standard that follows OpenGL ES 2.0 to introduce complex graphics functionalities to the browser, is an important factor that contributes the most, in terms of entropy, to the overall distinguishability of browser fingerprints [19]. Specifically, WebGL-based fingerprinting is first discovered by Mowery et al. [41], and then further explored by Cao et al. [19], who not only show that WebGL-based fingerprinting has the highest entropy among all fingerprinting factors, but also demonstrate the ability of WebGL-based fingerprinting for an even sneakier purpose, i.e., cross-browser fingerprinting, compared to traditional fingerprinting vectors like user agents.

In order to prevent WebGL from being used as a vector of browser fingerprinting, Tor Browser, the pioneer private browser for the Web makes WebGL click-to-play, i.e., disabling it by default, so that a website cannot use it for the tracking purpose. However, there exists a tradeoff between privacy and functionality: Tor Browser sacrifices an important functionality—i.e., all the computer graphics features brought by WebGL, which are particularly useful for modern web applications like games [10] and virtual reality [50]—for privacy. Specifically, according to a 2016 study [55], about 10% of Top 10K Alexa websites, including famous ones visited by billions of users such as Google Map and Earth [2], adopt WebGL to augment user experience—and the number keeps increasing as the WebGL community grows. Therefore, the research question that we want to ask in the paper is whether

a browser can allow Web applications to use WebGL and its abundant functionalities without violating users' privacy.

Before answering this question, we first take a look at how existing works prevent browser fingerprinting that does not use WebGL. There are two categories of approaches in defending against browser fingerprinting in general (e.g., these based on fonts, plugins, and user agent), which are randomization and uniformity. The former, adopted by PriVaricator [44] and some browser add-ons [1, 9], adds noise to the fingerprinting results so that an adversary cannot obtain an accurate fingerprint each time. However, according to prior work [18, 49], such randomization-based defense can be defeated if the adversary fingerprints the browser multiple times and averages the results. In addition, according to a recent work [59], inconsistencies in browser fingerprints may cause further privacy violations. Because of these concerns, Tor Browser also explicitly prefers the latter, i.e., uniformity, over randomization in its design document [49].

Therefore, our detailed research question becomes how to make uniform WebGL rendering results and prevent WebGL-based browser fingerprinting. The answer to this question is unknown in the community as indicated in Tor Browser's practice of disabling WebGL. The reason is that unlike other forms of fingerprinting (e.g., user agent and fonts), which rely on the outputs of a browser API, WebGL-based fingerprinting runs a program, i.e., a rendering task, in OpenGL Shading Language (GLSL). One possible solution, i.e., an idea floated in the design document [49] of Tor Browser without any implementation, is to adopt software rendering and make uniform WebGL rendering. However, Cao et al. [19] show that even if software rendering is enabled, WebGL rendering results are still fingerprintable.

Now, to answer the specific question of making WebGL uniform in the paper, we need to understand why a single WebGL rendering task differs much from one browser to another. From a high level, the reason is that computer graphic tasks pursue visual rather than computational uniformity. One single WebGL task on different browsers is rendered by a different combination of a variety of computer graphics rendering layers, such as browsers, graphics libraries (e.g., DirectX and OpenGL) including conversion interfaces (e.g., Almost Native Graphics Layer Engine, i.e., ANGLE), rendering mechanisms, device drivers and graphics cards. Therefore, different implementations and even versions of these various layers will lead to a computationally different rendering result. This high-level answer also partially explains the reason that software rendering cannot prevent fingerprinting: Software rendering, belonging to rendering mechanisms, is just one of the many layers that could lead to the rendering discrepancies, and it may also have different versions and implementations.

While this problem appears hard to solve unless we make uniform all the graphics layers, the root reason, after our intensive manual study and experiment, can be summarized as one surprisingly concise and abstract sentence—i.e., the

results of floating-point operations on different machines are different inside and across various graphics layers, leading to rendering differences. This one-sentence, intuitive reason can be further broken down into many sub-reasons when the WebGL rendering performs various operations in different graphics layers. Let us illustrate two examples.

First, we consider the color value, i.e., RGB, in WebGL, which semantically ranges from 0 to 255 but is represented as a floating-point from 0 to 1. Therefore, a conversion is required when WebGL renders a 0–1 color value on the screen to be a 0–255 RGB value—and most importantly the conversion, i.e., a floating-point operation, will lead to rendering difference. Say one WebGL implementation, i.e., a combination of different graphics layer, multiplies the color value with 255 and applies *floor* to convert it to the RGB value, and the other applies *round*. Then, a color value of $\langle 0.5, 0.5, 0.5 \rangle$ will be rendered as $\langle 127, 127, 127 \rangle$ in the former, but as $\langle 128, 128, 128 \rangle$ in the latter. This float-to-int conversion issue can be generalized in many other representation, such as alpha value, texture size, and canvas size—and also other conversion algorithms beyond *floor* and *round*, such as linear interpolation in texture mapping.

Second, let us consider another common graphics operation involving several float multiplications and then a subtraction, i.e., we need to decide whether a given point, very close to one triangle edge, is inside the given triangle. Say, there are two WebGL implementations, one adopting 10-bit float numbers and the other a higher precision, i.e., 16-bit. The multiplication results on these two implementations differ slightly, because the former has fewer decimals than the latter. Because the given point is very close to the triangle, such slight difference will propagate to the float subtraction, leading to a positive in the former implementation but a negative the latter. Therefore, the point will be judged as either inside or outside the triangle in these two implementations, causing a rendering difference.

That said, the *key* insight of the paper is that we need to make uniform all the floating-point operations across various computer graphics layers. Specifically, we adopt two integers, one as the numerator and the other as the denominator, to simulate floating-point operations in GLSL programs so that the underlying layers, regardless of their implementation or approximation for floating-point operations, always produce the same results. When we need to feed simulated values into WebGL, we convert the value to a float based on its semantics. In the aforementioned color value example, we can use $127/255$ to represent 127 and $128/255$ for 128, leading to no confusions under different implementations.

While the idea is intuitively simple, the major challenge is that WebGL rendering process involves implicit floating-point operations. In order to understand this challenge, let us briefly describe the three-stage WebGL rendering process—i.e., (i) vertex rendering, (ii) rasterization and interpolation, and (iii) fragment rendering—and corresponding floating-point operations in each stage. The first stage, controlled by a

GLSL program, i.e., vertex shader, generates vertices information using graphics operations, e.g., transformation and rotation, and also associates attribute values with each vertex. These aforementioned graphics operations are all related to floating-point operations. Then, the second stage, an implicit one implemented by WebGL and not controlled by any GLSL program, generates fragments, called rasterization, and then interpolates values based on fragments using floating-point operations. Lastly, the third stage, controlled by another GLSL program, called fragment shader, colors each fragment, which also involves floating-point operations, such as texture mapping.

In this paper, we propose UNIGL, a novel system that rewrites GLSL programs and redefines all the floating-point operations in the aforementioned three stages of WebGL rendering. Specifically, UNIGL hooks JavaScript APIs—which accept GLSL programs and the corresponding parameters such as vertex and index arrays—and then rewrites both vertex and fragment shaders via three phases mapping to the three stages of WebGL rendering. First, UNIGL converts the vertex shader to a JavaScript program and executes it. During the execution, floating-point operations in vertex shader, e.g., matrix multiplication, are executed as JavaScript, thus kept with uniformity. Second, UNIGL takes the execution results of JavaScript vertex shader and feeds them into a customized rasterization and interpolation engine written as a fragment shader. In this phase, UNIGL preserves uniformity for floating-point operation implemented natively in WebGL’s rasterization and interpolation module via integer simulation. Lastly, UNIGL rewrites the original fragment shader and redefines floating-point operations, such as texture mapping, in the fragment shader via integer simulation.

In designing UNIGL, we realize the following additional challenges from the viewpoint of system building.

- **Backward Compatibility.** We want UNIGL to be backward compatible with existing commercial Web Browsers. Specifically, we deploy UNIGL as a browser add-on, easily installable, to protect Web users’ privacy.
- **Performance.** We want to keep the high-performance benefits brought by WebGL, especially when it runs on GPU. Therefore, we design UNIGL so that the rendering bottleneck, i.e., rasterization, interpolation and coloring, runs as a GLSL program on fragment shader possibly via GPU (depending on whether the underlying rendering mechanism is software or hardware rendering). Note that the vertex shader has to run as a JavaScript program because otherwise we do not have access to intermediate results between two shaders without modifying the browser. Because of this, we adopt multiple optimization techniques, such as caching, typed array, and code-data separation, to speed up UNIGL’s vertex shader.
- **Variable Number Limits.** Since we choose backward compatibility, we are constrained by the limit that is enforced by the current version of WebGL. Particularly, all current

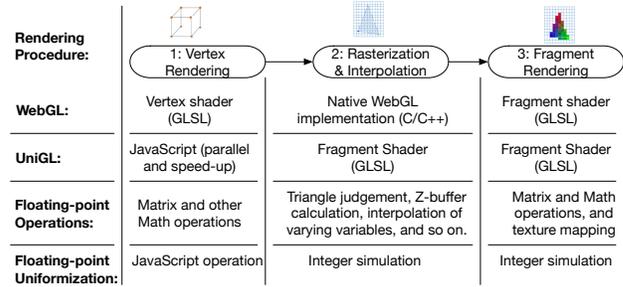


Figure 1: A High-level Overview of Rendering Procedure (i.e., the execution of GLSL programs) in both WebGL and UNIGL as well as a Description of Corresponding Floating-point Operations

implementations of WebGL have enforced a limit [26] for “uniform” variables, sometimes 256 or 1024 depending on the graphics card and operating system. Therefore, in designing UNIGL, we have to divide one rendering task iteratively until each small one can fit into and run as a fragment shader.

We implemented a prototype of UNIGL as an open-source Google Chrome add-on, which is available at the repositories of this GitHub user (<https://www.github.com/unigl/>). We also created a demo website, i.e., <http://test.unigl.org/>, which works on modern web browsers including Chrome, Firefox, and Safari. Specifically, the demo is a modified version of Cao et al.’s fingerprinting website [19], which directly integrates our add-on at the server-side, to show that UNIGL can prevent WebGL-based browser fingerprinting.

We make the following contributions in the paper:

- We are the first to point out that WebGL-based browser fingerprinting is caused by floating-point operations.
- We design UNIGL to rewrite GLSL programs and redefine all, i.e., explicit and implicit, floating-point operations embedded deeply inside WebGL.
- We show that our prototype of UNIGL can defend against WebGL-based browser fingerprinting with reasonable FPS.

2 Overview

We give an overview of WebGL and UNIGL rendering procedure, and then present a running example.

2.1 WebGL’s Rendering and Floating-point Operations

WebGL’s rendering procedure can be roughly divided into three stages as shown in Figure 1. First, the vertex shader in WebGL performs operations, e.g., rotation via a matrix multiplication, related to the vertices of a computer graphics model. Specifically, the shader accepts two types of variables, i.e., attributes and uniforms, binds attributes, such as texture coordinates, to vertices, and then outputs transformed vertices, i.e., $gl_Position$, and varyings. Many operations in the vertex shader, such as matrix multiplication and Math functions like \sqrt{x} , have floating-point values involved.

Second, the outputs of the vertex shader are fed into the rasterization and varying interpolation module implemented

natively by WebGL. The module maps a computer graphics model to each pixel on the canvas and interpolates each varying variable based on the attribute values on each vertex. Let us illustrate three major floating-point related operations in this module. (i) The module decides whether a given pixel is inside a triangle. (ii) The module calculates the z-buffer of each point, i.e., determining whether a point is in front of or behind another on the canvas. (iii) The module interpolates a varying variable based on attributes.

Lastly, the outputs of the rasterization and varying interpolation module are fed into the fragment shader in WebGL, which paints all pixels on the canvas by assigning a value to `gl_FragColor`. All the floating-point related operations in the vertex shader, such as matrix operations, also exist in the fragment shader. Additionally, color lookup operations, such as texture mapping, need to fetch a color from a texture, which involve floating-point operations as well.

2.1.1 An Explanation of Floating-point Operation and Rendering Discrepancies

We now explain that floating-point operations will cause rendering discrepancies. Figure 2 shows a classic computer graphics model, i.e., a 3D monkey head, covered with a random texture. We render this WebGL task in Google Chrome browser on two different machines, one iMac and the other Dell with Windows system—the rendering results, though being visually the same (Left and Middle of Figure 2), are quite different if we compare them pixel by pixel (Right of Figure 2).

This model is complex with many floating-point operations and thus hard to explain the discrepancies. We now decompose the complex model into several small experiments with only one or a few types of floating-point operations for explanation.

- A Varying Experiment. We setup a thin 3×100 rectangle and then specify a varying variable spanning along the long edge from 0 to 100. The variable starts from a color $\langle 0, 0, 0 \rangle$ at position 0 and ends to a color $\langle 0, 0, 255 \rangle$ at position 100. That is, we are rendering a simple spectrum on a canvas with only one type of floating-point operation, i.e., the interpolation of a varying variable between 0 and 100.

When we perform this varying experiment on different machines and compare the rendering results, we find that the result differences are several single-pixel lines orthogonal to the long edge. This explains that floating-point operations for interpolating varying variable will lead to rendering discrepancies.

- A Triangle Experiment. We setup a triangle on a 255×255 canvas: two vertices at $\langle 127, 0 \rangle$ and $\langle 0, 128 \rangle$, and the third movable along the line from $\langle 255, 0 \rangle$ to $\langle 255, 255 \rangle$. We then color the triangle with only a single color, such as black. By doing so, we create a rendering task with another

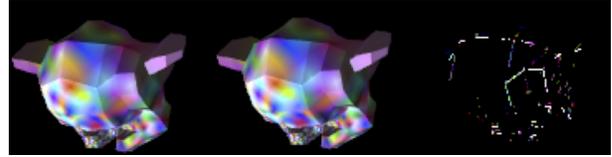


Figure 2: An Illustration of Rendering Task Difference for Browser Fingerprinting Purpose (Left: A classic monkey head model rendered in Google Chrome on an iMac, Middle: the same model rendered in Google Chrome on a Windows machine, Right: The pixel difference between these two rendering results.)

type of floating-point operation, i.e., determining whether a pixel is inside a triangle.

When we perform this triangle experiment on different machines and move the point along the edge, we find that the result difference is a single pixel close to the triangle edge for a special third vertex position. This explains that floating-point operations for triangle judgement will lead to rendering discrepancies.

- A Texture Experiment. We setup a triangle and then map a random texture onto the triangle. That is, we create a rendering task with a texture mapping operation, which has floating-point operations in color interpolation. When we perform this texture experiment on different machines, we find that the result differences are several pixels within the triangle. This explains that floating-point operations for texture mapping will lead to rendering discrepancies.

2.2 UNIGL’s Rendering and Floating-point Operations

Now let us go over the three-stage rendering procedure (Figure 1) again in UNIGL and show how to make uniform floating-point operations in aforementioned steps. First, UNIGL moves the vertex shader from GLSL to JavaScript, i.e., all the floating-point operations are executed on JavaScript interpreter and thus handled by CPU without any discrepancies. Note that we need to adopt parallel workers and many other speed-up techniques to run the JavaScript version of vertex shader fast.

Second, UNIGL implements a customized rasterization and varying interpolation module via GLSL in which all the floating-point values are represented via integer simulation, and therefore, all the aforementioned floating-point operations in this module are made uniform. It is worth noting that theoretically we can also implement the module via JavaScript for uniformization, but the performance is unacceptable.

Lastly, UNIGL executes the fragment shader in GLSL, but rewrites all the floating-point operations via integer simulation. Additionally, UNIGL also implements a customized texture mapping algorithm using integers so that the color lookup operation in texture mapping is also made uniform.

2.3 A Running Example

After a high-level overview of UNIGL rendering, we now present a detailed running example to illustrate our target problem and how UNIGL solves the problem via uniformity.

JavaScript:

```
1 vers=[...]; // vertices information
2 inds=[...]; // indices information
3 ...
4 var versBufferObject = gl.createBuffer();
5 gl.bindBuffer(gl.ARRAY_BUFFER, versBufferObject);
6 gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vers),
7   gl.STATIC_DRAW);
8 var indexBufferObject = gl.createBuffer();
9 gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBufferObject);
10 gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(
11   inds), gl.STATIC_DRAW);
12 ...
13 gl.bindBuffer(gl.ARRAY_BUFFER, versBufferObject);
14 var posAttr=gl.getAttribLocation(program, 'aVersPos');
15 ...
16 var uMVMLoc=gl.getUniformLocation(program, "uMVMMatrix");
17 gl.uniformMatrix4fv(uMVMLoc, [...]);
18 ...
19 // bind other attributes and uniforms
20 gl.drawElements(gl.TRIANGLES, length, gl.UNSIGNED_SHORT,
21   0);
```

Vertex Shader:

```
1 attribute vec3 aVersPos, aVersNormal;
2 attribute vec2 aTextureCoord;
3 uniform mat4 uMVMMatrix, uPMatrix;
4 uniform mat3 uNMatrix;
5 uniform vec3 uAmbColor, uLtDir, uDirColor;
6 varying vec2 vTextureCoord;
7 varying vec3 vLightWeighting;
8 void main(void) {
9   gl_Position=uPMatrix * uMVMMatrix * vec4(aVersPos.xyz,
10     1.0);
11   vTextureCoord=aTextureCoord;
12   vLightWeighting=uAmbColor + uDirColor * max(dot(
13     uNMatrix * aVersNormal, uLtDir), 0.0);
14 }
```

Fragment Shader:

```
1 varying vec2 vTextureCoord;
2 varying vec3 vLightWeighting;
3 uniform sampler2D uSampler;
4 void main(void) {
5   vec4 texColor=texture2D(uSampler, vTextureCoord);
6   gl_FragColor=vec4(texColor.rgb * vLightWeighting,
7     texColor.a);
8 }
```

JavaScript:

```
1 vers = [...]; // vertices information
2 inds = [...]; // indices information
3 ... // hook JS APIs to obtain arguments
4 var UniGL_aVersPos=UniGLAttr('aVersPos');
5 var UniGL_uMVMMatrix=UniGLUniform('uMVMMatrix');
6 ...
7 for(UniGL_I=0;UniGL_I<AttrLen;UniGL_I++) {
8   // Transformed JS vertex shader
9   aVersPos=UniGL_aVersPos[UniGL_I];
10  uMVMMatrix=UniGL_uMVMMatrix;
11  ...
12  UniGL_Position=UniGLMultiply(UniGLMultiply(uPMatrix,
13    uMVMMatrix), UniGLVec4(UniGLExtract(aVersPos,
14    [1,1,1]), 1.0));
15  vTextureCoord=aTextureCoord;
16  vLightWeighting=UniGLAdd(uAmbColor, UniGLMultiply(
17    uDirColor, UniGLMax(UniGLDot(UniGLMultiply(uNMatrix,
18    aVersNormal), uLtDir), 0.0));
19  // end of transformation
20  UniGL_vTextureCoord.push(vTextureCoord);
21  UniGL_vLightWeighting.push(vLightWeighting);
22 }
23 UniGL_CallFragmentShader(UniGL_Position,
24   UniGL_vTextureCoord, UniGL_vLightWeighting);
```

Fragment Shader:

```
1 #define N AttributeNumber
2 uniform ivec3 UniGL_Position[N];
3 uniform ivec2 UniGL_vTextureCoord[N];
4 uniform ivec3 UniGL_vLightWeighting[N];
5 uniform sampler2D uSampler;
6 void main(void) {
7   for (UniGL_I=0;UniGL_I<N;UniGL_I+=3) {
8     if (UniGL_InTriangleZBuffer(gl_FragCoord, UniGL_I)) {
9       vTextureCoord = UniGL_Interpolate(gl_FragCoord,
10         UniGL_I, UniGL_vTextureCoord[UniGL_I],
11         UniGL_vTextureCoord[UniGL_I+1], UniGL_vTextureCoord[
12         UniGL_I+2]);
13       vLightWeighting = ...;
14       // Transformed fragment shader
15       ivec4 texColor=UniGL_texture2D(uSampler,
16         vTextureCoord);
17       gl_FragColor=UniGL_i2f(ivec4(UniGL_Multiply(texColor
18         .rgb, vLightWeighting), texColor.a));
19       // end of transformation
20     }
21   }
22 }
```

Figure 3: A Running Example (Left: The original code, Right: The code rewritten by UNIGL—JavaScript WebGL APIs are hooked, the vertex shader is rewritten as JavaScript code, and the fragment shader is still as fragment shader with floating-point operations redefined.)

We now show the source code of this rendering task of Figure 2 in Figure 3 (Left). The source code contains three parts: JavaScript, Vertex Shader and Fragment Shader. First, the JavaScript code prepares data, such as attributes, uniforms, and texture, for both vertex and fragment shaders. Lines 4–13 (Left, JavaScript) show an example of passing attributes to the vertex shader. Next, Lines 14–16 (Left, JavaScript) show another example of passing uniforms to the vertex shader. Second, the vertex shader code accepts attribute and uniform values from the JavaScript and then performs operations, i.e., Lines 9–11 (Left, Vertex Shader), on each attribute in a parallel manner. The outputs of the vertex shader to the fragment shader are a special variable, *gl_position*, which indicates the transformed vertices, and multiple varyings. Third, the fragment shader accepts outputs, i.e., vertices and varying, from the vertex shader, performs rasterization and interpolation, and then runs the code (i.e., Line 5–6, Left, Fragment Shader).

Now, let us describe the floating-point related operations that cause the rendering result difference. First, the vertices, i.e., *gl_position*, and the varyings, i.e., *vTextureCoord* and *vLightWeighting*, are passed and interpolated between the vertex and the fragment shader. Such interpolation and accompanied rasterization involve floating-point operations and may cause difference. Second, WebGL functions, such as *dot* at Line 11 (Left, Vertex Shader) and *texture2D* at Line 5 (Left, Fragment Shader), include floating-point operations and may cause difference. Lastly, floating-point operations, such as Lines 9–10 (Left, Vertex Shader) and Line 6 (Left, Fragment Shader), may cause difference.

Next, we use Figure 3 (Right) to illustrate how UNIGL rewrites the original code and prevents such differences caused by floating-point operations. First, UNIGL will hook all the JavaScript APIs, such as *bindBuffer* and *bufferData*, to obtain vertices and indices information and associate them with attributes and uniforms in the vertex shader. Then, UNIGL rewrites the original vertex shader by replacing

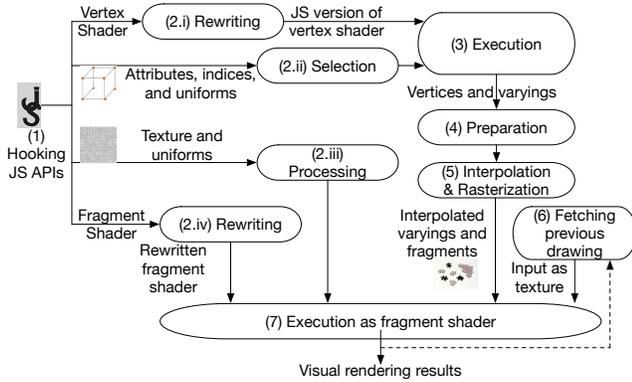


Figure 4: Overall Architecture of UNIGL (Steps 1–4 are executed as JavaScript, and Steps 5–7 as a fragment shader in a GLSL program). floating-point operations like *dot* with JavaScript functions like *UniGLDot*, and executes it as JavaScript at Line 9–15 (Right, JavaScript). Second, UNIGL collects the execution results from JavaScript-based vertex shader (Lines 16–17, Right, JavaScript) and then feeds them into a rewritten fragment shader as uniforms (Line 2–4, Right, Fragment Shader). Third, UNIGL rewrites the original fragment shader and execute it with customized rasterization and interpolation. The rasterization process at Lines 7–8 (Right, Fragment Shader) goes through all the triangles for each pixel on the canvas and determines whether a given pixel lies inside a triangle. If yes, UNIGL calculates the z-buffer of the given pixel for the triangle, decides whether the triangle is in the front and needs to be rendered, and also calculates pixel values based on alpha. Then, the interpolation process (Lines 9–10, Right, Fragment Shader) interpolates varyings based on vertices and attributes passed from JavaScript as uniforms. Lastly, UNIGL executes a rewritten fragment shader at Lines 12–13.

3 Design

We present the design of UNIGL in this section.

3.1 System Architecture

In this subsection, we show and describe the overall architecture of UNIGL. The main function of UNIGL is encapsulated as JavaScript files executed directly in the website runtime context. The add-on part is a thin layer used to inject UNIGL into the runtime context of the target website with WebGL tasks. That is, the purpose of the add-on is to ensure the injected UNIGL JavaScript code is executed before any website code. Once the injected scripts are running, the add-on code is disposable. We make this design choice so that UNIGL can be easily transferred between browsers.

Once the main part of UNIGL is injected into the target website, UNIGL performs seven detailed steps to render a WebGL task on a canvas as shown in Figure 4. First, UNIGL needs to perform some preparation tasks outside the three stages in Figure 1. In Step (1), UNIGL hooks WebGL related JavaScript APIs to obtain four types of information: (i) vertex shader (via *shaderSource* API), (ii) inputs to vertex

shader, such as attributes, indices and uniforms (via APIs, such as *bufferData* and *bindBuffer*), (iii) fragment shader (via *shaderSource* API), and (iv) inputs to fragment shader, such as texture and uniforms (via APIs, such as *texImage2D* and *texParameteri*). Then, in Step (2), UNIGL processes all the information obtained in Step (1). Particularly, UNIGL rewrites both vertex and fragment shaders in Step (2.i) and (2.iv), and extracts and prepares inputs in Step (2.ii) and (2.iii). For example, UNIGL reads indices and attributes, associates each attribute buffer with corresponding attribute variable in the vertex shader, and then prepares data based on the drawing mode (e.g., *drawElements* vs. *drawArray*, and *gl.POINTS* vs. *gl.LINES* vs. *gl.TRIANGLES*). Similarly, UNIGL also extracts texture information, such as image width and height, and texture mapping algorithm (e.g., *GL_LINEAR* vs. *GL_NEAREST*).

After that, in Step (3), UNIGL executes the rewritten vertex shader and generates outputs, i.e., vertices and varyings, which also belongs to Stage 1 in Figure 1. Next, in Step (4), UNIGL prepares inputs to the fragment shader by processing the outputs in Step (3), i.e., Stage 2 in Figure 1. Specifically, UNIGL performs backface culling on the triangles and iteratively divides the visible triangles by half so that each rendering task only contains uniforms within the limit enforced by WebGL. Then, in Step (5), UNIGL loops through all the pixels on canvas and determines whether each pixel falls inside the triangles or on the lines depending on the drawing mode. If yes, UNIGL interpolates all the varyings based on the given pixel and vertices. If no, UNIGL fetches previous drawing results from a special texture and uses the pixel color in Step (6). Lastly, UNIGL executes the rewritten fragment shader to calculate the pixel color in Step (7), i.e., performing Stage 3 in Figure 1.

3.2 Floating-point Operation Simulation

In this subsection, we present how to simulate floating-point operations using integers, i.e., the integer simulation method in Figure 1 for fragment shader. Such method is used in both Stage 2: Rasterization&Interpolation and Stage 3: Fragment Rendering, i.e., Step (5) and (7) in Figure 4.

3.2.1 Floating-point Representation and Operation

UNIGL adopts two integers, i.e., one numerator (p) and the other denominator (q), to represent an arbitrary floating-point value in the fragment shader. In this paper, we also refer the denominator as a *base*, because UNIGL can easily perform operations, such as addition and subtraction, on two values with the same base. Note that such representation also aligns well with the physical meaning of WebGL floating-point values. Take coordinates for example. The original vertices or texture coordinates are specified as an integer in terms of the canvas or texture size, which can serve as the numerator, and then the canvas or texture size can serve as the base. For

another example, 255 will be the base for all the color values, because all RGB colors are within the range of 255.

One important operation for UNIGL’s floating-point numbers is to change base for a given number. Such operation will be used when UNIGL converts simulated floating-point values to real ones represented by WebGL, such as the specification of texture coordinates at Line 12 and color values at Line 13 of Figure 3 (Right, Fragment Shader). The choice of a base depends on the underlying physical meaning of the WebGL functions, e.g., UNIGL adopts 255 as the base for *gl_FragColor*.

We now explain why the base representation can make uniform rendering results across browsers. Specifically, although a color value is represented as a float value internally in WebGL, WebGL has to convert it back to an integer when rendering the color on canvas. That is, if WebGL accepts a value p in between $1/255$ and $2/255$, it may render p as 1 or 2 depending on the underlying conversion algorithm. At contrast, if WebGL accepts a value as either $1/255$ or $2/255$, the ambiguity disappears and the results are uniform across browsers. In sum, UNIGL adopts different bases according to the physical meaning when UNIGL passes the floating-point value back to WebGL.

Next, we describe how to change base for a given number especially when it does not have the required base. Intuitively, we can multiply the value with the new base and divide the product with the old base. However, such intuitive approach does not work, because the division of one integer over another involves floating-point operations in some WebGL implementations. That is, the division result differs from one browser to another. Therefore, after obtaining the quotient, UNIGL needs to search within a range (i.e., ± 1) of the quotient for the real quotient. Other than the base change operation, UNIGL also supports basic arithmetic operations, which follow fraction operations. Due to simplicity, we skip details here. One thing worth noting is that UNIGL needs to avoid result overflows—if so, UNIGL needs to increase the base to accommodate a larger value.

In the next two subsections, we show how to use such base representation of floating-point values and replace existing ones in two important types of WebGL functions.

3.2.2 Floating-point Operations in Rasterization and Interpolation

Rasterization and interpolation are automatically performed in between the vertex and fragment shaders. Specifically, for all pixels on the canvas and all triangles, rasterization needs to decide whether the given pixel is inside the triangle and, if so, calculate the z-buffer. Note that both procedures involve floating-point values and operation, which need to adopt the base representation during calculation. Then, the interpolation calculates weights for three vertices at a given triangle and then outputs the interpolated varying, i.e., a weighted sum

of the attribute values at three vertices. All the weights and calculations are in the aforementioned base representation.

Another thing here, being different from normal rasterization and interpolation, is that UNIGL divides the entire, rectangle canvas into two triangles with four vertices as shown in Figure 5a instead of using the original vertices. The usage of such vertices is necessary to design a GLSL-version of rasterization, because the fragment shader will only expose a pixel to the GLSL program when the pixel is inside one triangle. Therefore, if we adopt the original vertices as the inputs to the fragment shader, some pixels, especially when they are on the edge, may be considered as inside a triangle by one browser but outside by another. The division shown in Figure 5a will consider all pixels on the canvas as being within either of these two triangles on any browser. This will give UNIGL the capability to go through all the pixels and decide whether a pixel, such as (x, y) in Figure 5a, is inside triangles consisted of the original vertices, i.e., $(x_1, y_1) \dots (x_4, y_4)$.

3.2.3 Floating-point Operations in Fragment Shader

The fragment shader uses many float-point related functions, such as “texture2D”, “normalize” and “sqrt”. In this subsection, we use texture mapping, i.e., “texture2D”, as an example to show the procedure of adopting integer simulation and replacing floating-point operations.

Texture mapping, in its normal definition, is a method of applying a two-dimensional surface upon a three-dimensional graphics model. There are many variations of texture mapping algorithms, such as linear interpolation (i.e., `GL_LINEAR`) and nearest neighbor (`GL_NEAREST`). Sometimes, mipmaps are also generated to process the texture before mapping. In this paper, we use linear interpolation as a proof of concept algorithm to show how to redefine texture mapping in UNIGL.

One of the major tasks in redefining texture mapping is to pass texture data from JavaScript to the fragment shader. The naïve method is to utilize “uniforms” just as vertices. However, because there exists a limit for the number of “uniform” variables and texture cannot be divided in multiple draws, we have to rely on existing texture information stored in WebGL.

Here is how UNIGL redefines a linear interpolation algorithm for texture mapping. UNIGL stores texture information using the default WebGL method with a nearest neighbor algorithm—therefore, WebGL will just directly fetch color values from the texture instead of performing any computation. When UNIGL has a texture coordinate, say, for example, $1/base$ in Figure 5c, UNIGL will first change the base to the size of the texture. Note that we use a square-shape texture as an example and a rectangle-shape will be similar.

Then, UNIGL fetches the colors, i.e., $c_1 \dots c_4$, of four texture points in Figure 5c, which locate around the target texture coordinate. Because UNIGL uses the texture size as the new base, the ambiguity among browsers will disappear. Next, UNIGL calculates two weights, i.e., w_1 and w_2 , based on the distance between the target texture coordinate and four texture

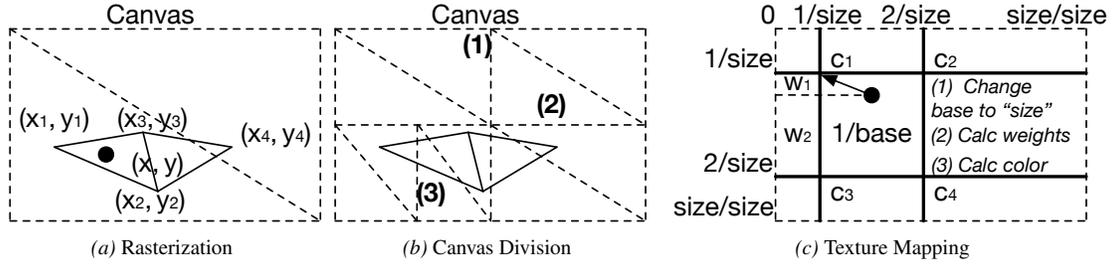


Figure 5: Explanation of Different UNIGL Operations. In subfigure (a), during rasterization, UNIGL needs to determine the triangle that each point belongs to. Specifically, Point (x, y) is inside the triangle consisting of vertices (x_1, y_1) , (x_2, y_2) , and (x_3, y_3) , but outside the other triangle consisting of vertices (x_2, y_2) , (x_3, y_3) , and (x_4, y_4) . Therefore, UNIGL will interpolate the values of all the varyings at (x, y) based on (x_1, y_1) , (x_2, y_2) , and (x_3, y_3) . In subfigure (b), UNIGL first divides the entire canvas into two parts, i.e. left and right, and then top and bottom. After that, because the number of vertices in some divided parts is small enough to be handled by WebGL, UNIGL will not further divide such parts, like top right and top left. If the number of vertices in a part is still larger than the minimum number of allowed uniform variables, UNIGL will further divide the canvas, like cut (3). Subfigure (c) adopts a square texture as an example to show how the linear interpolation algorithm of texture mapping works in UNIGL. UNIGL calculates an interpolated color value for a given point based on four color values around this point.

Table 1: WebGL Data and Corresponding JavaScript APIs that Accept such Data

Intercepted WebGL Data	JavaScript APIs
GLSL Program	createProgram, attachShader
Shader	attachShader, getShaderSource
Buffer	bufferData
Attribute	vertexAttribPointer
Attribute Location	getAttribLocation
Uniform	uniform*** (e.g., uniform1f)
Uniform Location	getUniformLocation

fetched points—these two weights can be in any base. Lastly, UNIGL calculates the color for the target texture coordinate using these two weights, i.e., $color = w_2w_2c_1 + w_1w_2c_2 + w_2w_1c_3 + w_1w_1c_4$.

3.3 Rendering Preparation

In this subsection, we introduce how to prepare inputs to both rewritten vertex and fragment shaders, i.e., Steps (1), (4), and (6). These steps do not have direct involvement with floating-point values, but are essential in preparing the vertex and fragment shaders in UNIGL.

JavaScript Hooking and Data Extraction. UNIGL hooks WebGL-related JavaScript APIs in Step (1) of Figure 4. Specifically, UNIGL utilizes the dynamic feature of JavaScript to redefine such JavaScript APIs, intercepts all arguments, i.e., parameters to WebGL, processes the arguments, and stores them in an internal data structure of UNIGL. Table 1 shows a list of WebGL data and corresponding JavaScript APIs. Such data can be roughly divided into two major categories: programs and inputs. “GLSL program” and “shader data” in Table 1 are intercepted by UNIGL for rewriting purpose. All others, such as “attribute” and “uniform”, are inputs to the program—UNIGL intercepts them and then feeds them to the rewritten programs. Both “attribute” and “uniform” refer to the data, e.g., colors and vertices, and “at-

tribute location” and “uniform location” are the corresponding variables defined in the shaders.

Backface Culling. UNIGL adopts backface culling [11] in Step (4) of Figure 4 to determine whether a polygon, such as a triangle, of a graphical object is visible. Specifically, UNIGL calculates the normal vector of all the triangles and filters these triangles of which the normal vector does not face the camera.

Rendering Task Division. Rendering task division, part of Step (4), is designed specifically in UNIGL to overcome the limit of the uniform variable numbers in fragment shader. Figure 5b shows an illustration of such division. UNIGL first divides the canvas vertically by half, e.g., division (1) in Figure 5b, and then horizontally, e.g., division (2). Such vertical or horizontal division will be performed alternatively in each iteration on a small region until the number of vertices in that region is smaller than the limit enforced by WebGL implemented in a specific browser. Note that if a triangle is partially inside a region, e.g., the right corner region in Figure 5b, UNIGL will count all three vertices towards the limit. The reason is that all three vertices are required to determine whether a given pixel lies inside a triangle.

Reading Previous Draw Results. In this part, we describe how UNIGL handles multiple draws in Step (6). For example, a WebGL program may first draw a triangle by calling one GLSL program and then a rectangle by calling another. In such multiple draws, the rendering results are treated as a background in the latest draw. Because UNIGL goes through all the pixels in the fragment shader each time, UNIGL needs to read previous draw results. Specifically, UNIGL relies on the readPixels API to obtain the canvas contents, constructs a texture based on the contents and then passes the texture to the fragment shader. Then, in the fragment shader, UNIGL first reads the color value from the texture, and assigns the color to $gl_FragColor$. Later on, if the current drawing renders a

```

1  attribute vec2  vertPosition;
2  void main() {
3    gl_Position =  vec4(vertPosition, 0.0, 1.0);
4    gl_PointSize = 1.0;
5  }

```

Figure 6: Dummy Vertex Shader (The vertex shader performs a self-mapping with a z-value as 0.0 and a w-value as 1.0. The point size is also set to be 1.0.)

color on this pixel, the assigned color will be overwritten; otherwise, the assigned color is treated as the background.

3.4 Rewriting and Rendering

In this subsection, we describe UNIGL’s rewriting and rendering process, i.e., Step (2). In both Steps (2.i) and (2.iv), UNIGL preprocesses the GLSL program, i.e., replacing pre-processor directives with a hash symbol at the beginning, and then parses the processed program into Abstract Syntax Tree (AST). We then discuss how to rewrite vertex and fragment shaders separately.

- **Vertex Shader.** UNIGL traverses through the AST, redefines corresponding node, e.g., replacing the operator plus with a function `UniGL_Plus`, and then converts the AST back to either JavaScript or GLSL code. All the type information is kept the same because UNIGL has redefined all the types in JavaScript to be the same as in GLSL. Note that one additional step is that UNIGL needs to divide the w value of `gl_Position` from the x , y and z values so that UNIGL can switch the rendering results from an orthographic view to a perspective view. This step was performed implicitly in the original WebGL, and UNIGL needs to do so as well.
- **Fragment Shader.** UNIGL traverses through the AST and redefines the following three types of nodes: (i) operators, (ii) constant float number, and (iii) type information related to floating point values. First, UNIGL redefines all the existing operations, such as multiply, with the corresponding GLSL function, such as `UniGL_Multiply`. Second, UNIGL converts all the constant floating point values, e.g., 0.32 as an alpha value, to our base representation, e.g., 32 as the value and 100 as the base for all alphas. Lastly, UNIGL need to convert all the types related to floating point values, such as float and `vec3`, to the corresponding integer type, such as int and `ivec3`.

Note that WebGL requires that all GLSL programs have both vertex and fragment shaders. Therefore, UNIGL executes a dummy vertex shader as shown in Figure 6. The dummy vertex shader needs to set `gl_PointSize`, because the default value also differs on different OSes, e.g., Mac vs. Windows.

3.5 Execution of JavaScript Vertex Shader and Corresponding Floating-point Operations

In this subsection, we describe the execution of JavaScript vertex shader in Step (3). All the floating-point operations in

the vertex shader are thus executed on CPUs. We adopt five runtime optimizations to speed up the execution as shown below.

- **Multiple Web Workers.** Once one frame comes in, UNIGL puts the vertex rendering tasks of the frame into a queue. Then, multiple workers keep fetching the tasks from the queue, and run them in parallel.
- **Caching.** UNIGL adopts a caching mechanism for matrix operations, e.g., `UniGLAdd` and `UniGLMultiply` at Lines 12 and 14 of Figure 3 (Right, JavaScript), in Step (3). That is, UNIGL will cache the calculation results of a matrix operation, e.g., the multiplication of two matrices. If the rewritten vertex shader asks for the results of an operation upon the same matrices, UNIGL will directly return the result directly instead of calculating it again.
- **Typed, fixed-size Array.** UNIGL adopts typed arrays, such as `Float32Array`, instead of the normal JavaScript array to store data. The reason is that typed array are stored in a contiguous memory region can fast accessed by the browser. In addition, UNIGL needs to specify the length to avoid array resizing. The reason is that array resizing may involve additional memory allocation and data copy. Note that we also need to avoid using some heavy JavaScript array operations, such as `Array.map()`.
- **Code and Data Separation.** UNIGL separates the code and the data for the rewritten shader. That is, the code will be prepared in the hooked `useProgram` API, which does not have an influence on the runtime performance, and JIT-ed for performance speed-up. Note that UNIGL triggers the JIT engine by executing the code once with initial data. Then, further data will be prepared in the draw stage.
- **WebAssembly.** UNIGL executes some heavyweight operations, such as matrix multiplication, using native code like WebAssembly.

4 Implementation

We implemented a prototype of the core function of UNIGL with around 8,500 lines of JavaScript code, around 650 lines of GLSL code, and around 3,000 lines of code for auxiliary components such as WebAssembly and add-on. The rewriting component of UNIGL is modified from one open-source GitHub repository (namely, <https://github.com/stackgl/glsl-transpiler>). Other than this repository, we also use several other libraries, such as `glMatrix`.

UNIGL is open-source, available at repositories under this GitHub user (<https://www.github.com/unigl/>). We also provide a demo at this url (<http://test.unigl.org/>), a modified version of Cao et al. [19]’s fingerprinting websites, to demonstrate that UNIGL can prevent state-of-the-art WebGL-based browser fingerprinting. All rendering results are the same in this new, UNIGL rewritten version.

5 Evaluation

We evaluate UNIGL prototype on three metrics: anti-fingerprinting capability, performance, compatibility, and CPU energy consumption.

5.1 Anti-fingerprinting Capability

We adopt a state-of-the-art WebGL-based browser fingerprinting work [19] as our benchmark to evaluate the anti-fingerprinting capability and performance of UNIGL. Specifically, the benchmark contains 17 different WebGL rendering tasks including plain WebGL tasks and these relying on three.js, a WebGL library, to explore various WebGL features, such as varyings, light and texture. The first column of Table 2 shows the names of all the rendering tasks and the second column a rendering result example on a dell desktop installed with Windows 10.

We evaluate UNIGL by asking Amazon Mechanical Turks to visit our website—including a demo site of UNIGL together with the original fingerprinting site from Cao et al.—using Firefox, Chrome and Safari. In total, we have collected 656 fingerprints from these three types of browsers. Among all the 656 fingerprints, UNIGL only renders one unique fingerprint for each rendering task across different browsers. This unique fingerprint, being visually the same to the original rendering result, is shown in the “Example” column under UNIGL of Table 2. As a comparison, we also show the number of unique fingerprints of Cao et al. in the “# Unique Results” column under “Original” of Table 2—this confirms Cao et al.’s findings that WebGL is a high-entropy vector for browser fingerprinting. We also list some more statistics in Appendix A.

In addition to the Amazon Mechanical Turk experiment, we also perform a local experiment that enumerates a large varieties of factors across the graphics layers. Specifically, we test UNIGL with the following different settings: OS (Windows 7, 8, 10, iOS 10.14.1, and Ubuntu 18.04), graphics card (Nvidia Geforce GTX 1070, Intel Iris plus Graphics 640, AMD Radeon R9 M390, and AMD Radeon HD 6770m), drivers (Nvidia, Intel, and AMD drivers), screen resolution (all these provided by the OS, such as 2560x1440 and 1920x1080), and DPI scaling (100%, 125%, 150%, 175%, 200%, and 225%). UNIGL only produces one unique result, which is the same as the Amazon Mechanical Turk experiment.

5.2 Performance

We test the performance of UNIGL by using both micro- and macro-benchmarks. All the experiments, except for the crowdsourced results in Table 2, are performed on an iMac – the machine has an Intel Core i5, 3.2 Hz, 4-core CPU, 24 GB memory, and an AMD Radeon R9 M390 GPU with 2048 MB VRAM.

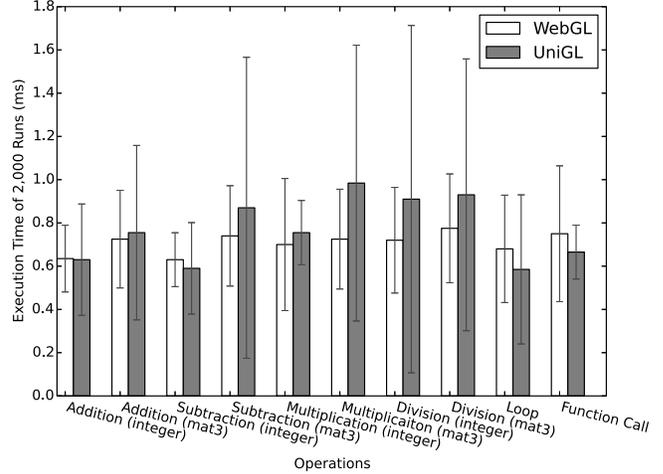


Figure 7: Micro-benchmark of Vertex Shader.

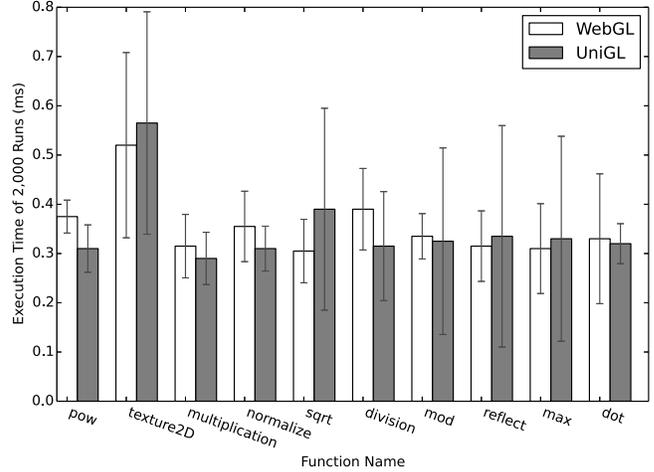


Figure 8: Micro-benchmark of Fragment Shader (all the operations in WebGL are using floating-point values; these in UNIGL adopts integer simulation).

5.2.1 Micro-benchmark

In the micro-benchmark, we test several atomic WebGL operations and compare the original version with the one defined in UNIGL. Specifically, we run each operation in either fragment or vertex shader for 2,000 times and calculate the interval between two draws. Each experiment is performed 20 times to obtain a standard deviation. Note that we adopt a simple model, i.e., a cube, in the micro-benchmark experiment. When we are testing one shader, the other shader will contain a one-line, dummy statement, i.e., the assignment of either `gl_Position` or `gl_FragColor`.

Figure 7 shows the micro-benchmark performance of the vertex shader. The vertex shader of UNIGL outperforms the one written in GLSL in some aspects, such as these operations that have integers involved. The reason is that CPU is well designed for integer operation when compared with GPU. On the contrary, the original shader written in GLSL is better at matrix operations, because such operations can be performed in parallel using shading languages.

Table 2: Macro-benchmark WebGL Tasks [19] and Corresponding Rendering Results with UNIGL (“# Vertices” means the number of vertices in the model, which are two per line segment in a 2D model and three per triangle in a 3D model. “Example” is one rendering example collected from users. In “# Unique Results” columns, X/Y means the number of unique fingerprints collected from all the users out of the total number of fingerprints. “FPS” means frames per second—which is around 60 Hz due to the screen refresh rate.)

WebGL Task	# Vertices	Example	Original				UNIGL		
			# Unique Results			FPS	Example	# Unique Results	FPS
			Chrome	Firefox	Safari				
Curve and Line	262		74/496	23/108	18/52	60.32±0.38		1/656	61.77±0.54
Curve and Line (AA)	262		83/496	32/108	21/52	60.78±0.54		1/656	61.83±0.97
Cube	36		4/496	2/108	5/52	60.67±0.49		1/656	62.50±0.80
Cube (AA)	36		55/496	20/108	23/52	60.46±0.15		1/656	62.39±1.36
Cube (Camera)	36		56/496	18/108	7/52	60.02±0.22		1/656	61.75±1.32
Monkey head (Texture)	2,904		5/496	18/108	2/52	60.14±1.17		1/656	61.88±1.61
Monkey head (Light)	2,904		36/496	11/108	15/52	59.95±0.60		1/656	61.07±1.02
Two models (Light)	2,988		44/496	18/108	14/52	60.02±1.19		1/656	61.90±0.97
Two models (Complex light)	2,988		52/496	6/108	20/52	60.18±0.40		1/656	60.02±1.13
Two models (Texture)	2,988		79/496	31/108	21/52	60.31±0.54		1/656	62.33±1.22
Two models (Transparency)	2,988		87/496	25/108	22/52	59.97±1.13		1/656	60.13±1.76
Two models (Tex&Light)	2,988		38/496	14/108	11/52	60.04±0.31		1/656	59.74±0.75
Thousands of rings (three.js)	5,376		53/496	25/108	15/52	60.52±0.53		1/656	57.47±1.87
Clipping plane (three.js)	36		44/496	14/108	19/52	59.98±0.44		1/656	59.67±1.29
Bubble (three.js)	974		49/496	17/108	22/52	60.20±1.52		1/656	60.07±1.43
Compressed Texture (three.js)	98		72/496	21/108	19/52	60.04±0.56		1/656	59.59±0.73
Shadow (three.js)	156		53/496	17/108	19/52	59.84±0.35		1/656	60.12±1.02
Combined fingerprint			123/496	41/108	26/52			1/656	

Figure 8 shows the micro-benchmark performance of the fragment shader. Similar to the vertex shader, while UNIGL is slower than the original WebGL in some cases, such as “texture2D” as UNIGL redefines the function, it is worth noting that UNIGL is faster than the original WebGL in many other cases, such as “pow” and “multiplication”. The reason is that an integer operation is indeed sometimes cheaper than a floating-point one. For example, it takes less time to multiply two integers than two floating-point values. Note that we are referring to integer operations that exist in the original vertex shader during this discussion. Floating-point values are still represented as floats in the vertex shader of UNIGL.

5.2.2 Macro-benchmark

In this subsection, we use the WebGL tasks provided by Cao et al. [19] as our macro-benchmark to measure the FPS of

these rendered by UNIGL. The column “FPS” under UNIGL of Table 2 shows the FPS of each rendering task and we also show the FPS without UNIGL, i.e., these rendered directly by WebGL in the same table. The performance of UNIGL can satisfy the required screen refresh rate, i.e., 60 Hz. The FPS of UNIGL for all the tasks are similar to the original one rendered by WebGL alone.

There are two things worth noting. First, the FPS of UNIGL is even sometimes a little bit higher than the one of WebGL. The reason is that when the model is simple, our highly optimized vertex shader with the help of WebAssembly is faster than the original one. Second, the FPSes of both UNIGL and WebGL are a little bit higher than 60 Hz in some tasks, because modern browsers reduce the precision of *performance.now* to prevent timing attacks [7], which may

Table 3: Overhead Breakdown for the “Two models (Complex light)” Task

Procedure	Overhead
Data Preparation	0.08±0.07ms
Backface Culling	2.16±0.06ms
Vertex Shader	2.26±0.18ms
Rendering Task Division	5.34±0.73ms
Fragment Shader	6.51±0.85ms
Total	16.35±0.74ms

Table 4: Vertex Shader Optimization (all numbers are averaged from 10 experiments and rounded to ms)

Unoptimized Vertex Shader	110 ms
Result Caching of Matrix Operation	-24 ms
Typed, Fixed-size Array for Data	-22 ms
Code and Data Separation	-39 ms
Parallelization	-19 ms
WebAssembly	-4 ms
Optimized Vertex Shader	2 ms

lead to a small measurement error. Such measurement errors are consistent across UNIGL and WebGL.

We further look at one specific task, i.e., “Two models (Complex light)”, and analyze the overhead brought by UNIGL. Table 3 shows the overhead breakdown by different procedures of UNIGL. The rendering task division and fragment shader are the most time-consuming procedures, i.e., each taking one third of the entire overhead. Both data preparation and backface culling are lightweight, taking up a small portion of the overhead.

We then look at how our optimization reduces overhead of UNIGL, especially the vertex shader, using the same task. Specifically, we evaluate five optimizations and their impact on the performance in Table 4. The unoptimized vertex shader in JavaScript takes 110ms and each optimization reduces the overhead to some degree. Code and data separation is the most effective one, i.e., about 40ms reduction, because JIT engine will execute code natively rather than on an interpreter. Then, both caching and typed, fixed-size array speed up the shader by reducing around 20ms, and parallelization also reduces the overhead by around 19ms. Lastly, if we apply WebAssembly optimization, the overhead can also be reduced by 4 ms.

5.3 Compatibility

In this section, we evaluate the compatibility of UNIGL with existing WebGL applications. Specifically, in addition to the WebGL tasks from Cao et al. [19], we run UNIGL using two other real-world WebGL applications shown below:

- **Zygote Body.** Zygote Body (<https://www.zygotebody.com/>), formerly known as Google Body, is created by Zygote Media Group to renders a manipulable 3D model of human body from outside, such as skins, muscle tissues and hairs, to inside, such as blood vessels and skeletons.

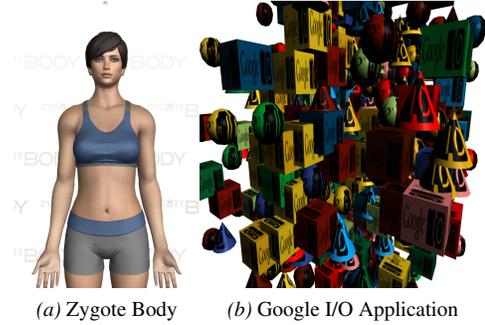


Figure 9: Two Screenshots of Zygote (Google) Body and Google I/O Application Rendered by UNIGL

- **Google I/O 2011 Applications.** Google has presented WebGL applications (<https://webglsamples.org/google-io/2011/index.html>) at its I/O event in 2011 to show the new technique and performance.

Our evaluation result shows that UNIGL is compatible with both applications. First, we run UNIGL with Zygote Body—the human body is rendered correctly with no visual difference. We can also manipulate it by looking at different layers, such as skeleton and muscle. Second, we run UNIGL with Google I/O applications—all the objects are shown and displayed correctly with the right texture, moving on the canvas the same as ones with WebGL directly. Two screenshots of both applications are also shown in Figure 9: Figure 9a shows the front page of Zygote Body, a default rendering of a human, and Figure 9b a screenshot of one Google I/O application after it runs for two seconds.

5.4 CPU Energy Consumption

In this section, we evaluate the CPU package power of our macro-benchmark. Specifically, we use CPUID’s HWMonitor [3], a program that monitors PC systems’ main health sensors, to calculate the CPU package power consumption for each macro-benchmark task. Note that our experiment is performed on a Dell Desktop because HWMonitor is only available on PC systems.

Figure 10 shows our evaluation results, i.e., CPU package power consumption when each macro-benchmark task is rendered by WebGL (hardware rendering), WebGL (software rendering), and UNIGL. The power consumption for WebGL (hardware rendering) is the smallest for all the tasks because hardware rendering relies on GPU to perform computation. UNIGL is the second, because UNIGL relies on CPU for rendering vertex shader, but GPU for fragment shader. WebGL (software rendering) is the highest as all the rendering tasks are performed on CPU.

It is also worth noting that CPU package powers for “thousands of rings” and “clipping plane” are the highest compared with other tasks. The reason is that the vertex shader for both tasks are computationally heavy. Take “thousands of rings” for example. The position and shapes for all the rings are calculated in the vertex shader.

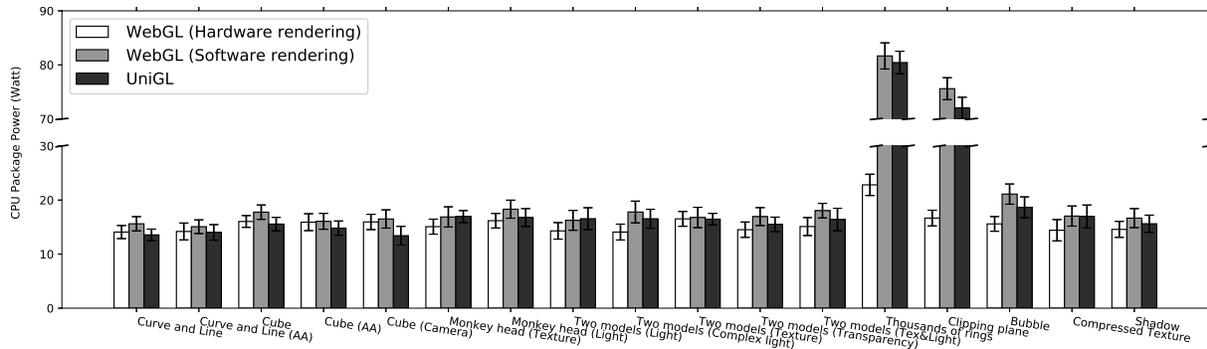


Figure 10: CPU Package Power Comparison among WebGL (hardware rendering), WebGL (software rendering), and UNIGL. (Note that all other WebGLs mentioned in the paper except for this figure and corresponding texts refer to the default hardware rendering.)

6 Discussion

We discuss several issues regarding UNIGL in this section.

- **Timing side-channel attacks.** Timing information, such as the rendering speed of WebGL tasks, may also be used for fingerprinting. For example, Naghibijouybari et al. [43] shows that timing side channels exist in GPU, e.g., WebGL rendering tasks. Similarly, the execution time of floating-point operations can also be used as a timing side channels [29, 51]. Many existing works, such as Deterministic Browser [18], JavaScript Zero [54], and CTFP [14], are proposed to defend against such timing channels, and therefore we would consider such timing-based fingerprinting out of scope of the paper.
- **Self-modifying code (i.e., an strong adversary aware of UNIGL).** There are three techniques used to prevent self-modifying code that is aware of the existence of UNIGL from tampering the UNIGL code and logics. First, UNIGL adopts anonymous closure to encapsulate all the core code of UNIGL from access by any potentially malicious website JavaScript. Specifically, anonymous closure makes sure that all the private variables and original WebGL functions, such as drawElements, are securely protected. Second, UNIGL obtains all the system object, such as “undefined”, to avoid tampering from an adversary. Lastly, the add-on code injects the main function of UNIGL as the first script to execute before any other website JavaScript. It is worth noting that we are aware that there exists an active Chrome bug [4] at the time when we write the paper, which is about “document_start” hook on child frames. We believe that this bug should be fixed to follow the specification of Chrome extension.
- **WebGL Vulnerability.** WebGL may expose some low-level vulnerabilities in device drivers to web applications [62]. As discussed in Milkomeda [62], WebGL has already imported security checks to prevent an attacker exploiting such vulnerabilities.
- **Fingerprinting via WebGL meta-information.** We realize that not only the rendering behaviors of WebGL tasks but also the meta-information of WebGL engine and implementation can be used for fingerprinting. For example, as shown

by this website (<https://browserleaks.com/webgl>), different WebGL meta-information, such as vendor, renderer, and shader parameters, can all be used as part of browser fingerprinting.

We would like to point out that such fingerprinting relying on WebGL meta-information is relatively easy to prevent as shown by Tor Browser’s in uniformization of the reported values of such meta-information. Specifically, Tor Browser changes and makes uniform the return values of WebGL meta-information functions, such as `getParameter()`, `getSupportedExtensions()`, and `getExtension()`. Therefore, we encourage one to rely on Tor Browser for prevention of such fingerprinting.

- **Floating-point Value Precision.** We now discuss the precision of floating-point values used in UNIGL. Our integer simulation of floating-point values can meet the needs of graphics tasks, because WebGL does not need a high-precision definition of floating-point variables. Specifically, the semantic meaning of many WebGL variables only requires a relatively low-precision value. Take color values for example: Each color value only ranges from 0 to 255 and thus a high-precision, 16-bit floating-point value will not represent more colors. In fact, many WebGL implementation only support a medium (10 bits) or low (8 bits) for float variables.
- **Future WebGL-based fingerprinting.** State-of-the-art WebGL fingerprinting is based on differences in floating-point operations. We empirically verify this via carefully-designed experiments following WebGL specifications in Section 2.1.1 and our evaluation of UNIGL against Cao et al.’s WebGL fingerprinting [19]. Future WebGL fingerprinting techniques and those that do not rely on WebGL are out of scope of the paper.
- **Ethic concerns.** We discussed with our Institutional Review Board (IRB) about the ethics of the proposed research and experiment, because we require a human to run our experiment on their machines. The conclusion is that the proposed research does not require IRB approval. The reason is that although browser fingerprinting may be used to collect private information, the fingerprint itself, just like

cookies, is just an identifier, which does not contain any private information. Our experiment only collects fingerprints but not any private information associated with the fingerprint.

7 Related Work

We discuss related work in this section.

Browser Fingerprinting. Browser fingerprinting is a second-generation web tracking that goes beyond cookies or super cookies [30–33, 35, 53] to utilize inherent features inside web browsers. For example, there are many works [12, 13, 20, 23, 45, 63] performing measurement works on browser fingerprinting. Browser fingerprinting can rely on many features. Laperdrix et al. [34], i.e., AmIUnique, is a comprehensive study on 17 features of browser fingerprinting. Then, Vastel et al. studied the dynamics of fingerprints [59], i.e., how browser fingerprints change over time, and the privacy implication of browser fingerprint inconsistencies [60]. Researchers also study specific features of browser fingerprinting, such as fonts [23], AudioContext [20], and JavaScript engine [40, 42]. The defense target of the paper is WebGL-based fingerprinting, which was first proposed by Mowery et al. [41] and then thoroughly examined by Cao et al. [19].

Floating-point Timing Channel. A floating-point timing channel [29, 51] of web browser refers to that the duration of a floating-point operation can be used to break same-origin policy. Other than floating-point timing channel, many other timing channels [16, 22, 24, 25, 28, 39, 46, 57, 58, 64, 65] have also been studied. As a comparison, the side channel studied in the paper is caused by the different results of floating-point operations, such as conversion from low resolution to high resolution, but not the different duration of floating-point operations. Therefore, we consider that floating-point timing channels are out of scope of the paper, and one should refer to existing works [18] for solutions.

Defense against Fingerprinting. To the best of knowledge, none of existing works can defend against WebGL-based browser fingerprinting while still preserving its functionality. The reason is that we believe we are the first to point out floating-point operations are the root cause for WebGL-based browser fingerprinting. UNIGL is also the first system to find out and then redefines such floating-point operations that cause rendering discrepancies.

In the related work, Tor Browser [48], as discussed, is the pioneer work in defending against browser fingerprinting, but it disables WebGL for privacy. PriVaricator [44] adds noises to browser fingerprinting, which can be defeated if the adversary runs the fingerprinting multiple times. Similarly, Multilogin Browser [5] also creates a virtual browser profile with a random fingerprint. TrackingFree [47] only defends against the first-generation web tracking, i.e., these based on cookies or super cookies, but not browser fingerprinting.

Rewriting Technique. In the past, both academia [21, 27, 38] and industry [6, 8] have adopted rewriting techniques in dif-

ferent scenarios. In academia, WebShield [38] and BrowserShield [52] rewrite webpages in a proxy to enable web defense techniques. Erlingsson et al. [21] enforce cyber security policies by rewriting binaries. In industry, ShapeSecurity [8], a commercial company, provide products to rewrite websites and prevent bots and malware. Google’s PageSpeed Module [6] also rewrites webpages to improve their performance.

As a comparison, there are unique challenges in rewriting GLSL languages in UNIGL, because it contains two shaders and many internal variables, such as varyings and uniforms. Specifically, UNIGL not only redefines floating-point operations, but also implements rasterization and interpolation in fragment shader so that corresponding floating-point operations in these two procedures can be made uniform.

Determinism. Determinism is a technique used to defend against side-channel attacks. For example, StopWatch [36, 37], Deterministic Browser [18], and DeterLand [61] adopt determinism to defend against timing channels. Burias et al. [17] design a deterministic information-flow control system to defend against cache attacks and then Stefan et al. [56] prove that such cache attacks are still possible given a reference clock. Aviram et al. [15] use provider-enforced deterministic execution to prevent timing channels within a shared cloud domain. As a comparison, UNIGL makes the execution results the same but not the execution time across different browsers.

8 Conclusion

In this paper, we propose UNIGL, a novel system that rewrites GLSL programs and renders them uniformly across different browsers, thus preventing WebGL-based browser fingerprinting. UNIGL redefines all the floating-point operations, either explicitly written in the shader, or implicitly invoked by the WebGL system.

We implemented an open-source prototype of UNIGL as a browser add-on. Our evaluation shows that UNIGL can defend against state-of-the-art WebGL-based fingerprinting, i.e., there exists only one rendering result when Amazon Mechanical Turks visit our demo website from different browsers on different machines. Our evaluation also shows that the performance of UNIGL can satisfy the needs for the screen refresh rate, i.e., the FPSes of graphics tasks rendered by UNIGL are around 60 Hz.

In the future, we believe that browser vendors should integrate UNIGL natively into browsers. If they choose to do so, they can directly use integer simulation for all the components including vertex shader, rasterization & interpolation engine, and fragment shader, because the outputs from vertex shader, though unavailable in the JavaScript-level, are accessible and can be made uniform directly in native browser. Additionally, a native implementation does not need the rendering task division step in UNIGL because there will be no “uniform” variable limit in the low-level.

Acknowledgment

We would like to thank our shepherd, Ben Stock, and anonymous reviewers for their helpful comments and feedback. This work was supported in part by National Science Foundation (NSF) grant CNS-18-12870 and an Amazon Research Award. 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 or Amazon.

References

- [1] Canvas defender. <https://multiloginapp.com/canvasdefender-browser-extension/>.
- [2] Google maps meets webgl. https://www.youtube.com/watch?v=X3EO_zehMkM.
- [3] Hwmonitor—voltages, temperatures and fans speed monitoring. <https://www.cpuid.com/softwares/hwmonitor.html>.
- [4] Issue 793217: “document_start” hook on child frames should fire before control is returned to the parent frame. <https://bugs.chromium.org/p/chromium/issues/detail?id=793217>.
- [5] Multilogin. <https://multilogin.com/>.
- [6] Pagespeed module: open-source server modules that optimize your site automatically. <https://developers.google.com/speed/pagespeed/module/>.
- [7] Reduce resolution of performance.now to prevent timing attacks. <https://bugs.chromium.org/p/chromium/issues/detail?id=506723>.
- [8] Shape security. <https://www.shapesecurity.com/>.
- [9] Trackoff privacy software. <https://www.trackoff.com/en>.
- [10] WebGL games. <https://www.crazygames.com/t/webgl>.
- [11] [wikipedia] back-face culling. https://en.wikipedia.org/wiki/Back-face_culling.
- [12] 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*, CCS '14, pages 674–689, New York, NY, USA, 2014. ACM.
- [13] Gunes Acar, Marc Juarez, Nick Nikiforakis, Claudia Diaz, Seda Gürses, Frank Piessens, and Bart Preneel. FPDetective: Dusting the web for fingerprinters. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, CCS '13, pages 1129–1140, 2013.
- [14] Marc Andryscio, Andres Nötzli, Fraser Brown, Ranjit Jhala, and Deian Stefan. Towards verified, constant-time floating point operations. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 1369–1382, New York, NY, USA, 2018. ACM.
- [15] Amittai Aviram, Sen Hu, Bryan Ford, and Ramakrishna Gummadi. Determining timing channels in compute clouds. In *Proceedings of the 2010 ACM Workshop on Cloud Computing Security Workshop*, CCSW '10, pages 103–108, New York, NY, USA, 2010. ACM.
- [16] Andrew Bortz and Dan Boneh. Exposing private information by timing web applications. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 621–628, New York, NY, USA, 2007. ACM.
- [17] Pablo Buiras, Amit Levy, Deian Stefan, Alejandro Russo, and David Mazieres. A library for removing cache-based attacks in concurrent information flow systems. In *International Symposium on Trustworthy Global Computing*, pages 199–216. Springer, 2013.
- [18] Yinzhi Cao, Zhanhao Chen, Song Li, and Shuijiang Wu. Deterministic browser. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, 2017.
- [19] Yinzhi Cao, Song Li, and Erik Wijmans. (cross-)browser fingerprinting via os and hardware level features. In *Annual Network and Distributed System Security Symposium*, NDSS, 2017.
- [20] Steven Englehardt and Arvind Narayanan. Online tracking: A 1-million-site measurement and analysis. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, 2016.
- [21] Ulfar Erlingsson and Fred B Schneider. Irm enforcement of java stack inspection. In *IEEE S&P*, 2000.
- [22] Edward W. Felten and Michael A. Schneider. Timing attacks on web privacy. In *Proceedings of the 7th ACM Conference on Computer and Communications Security*, CCS '00, pages 25–32, New York, NY, USA, 2000. ACM.
- [23] David Fifield and Serge Egelman. Fingerprinting web users through font metrics. In *Financial Cryptography and Data Security*, pages 107–124. Springer, 2015.
- [24] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. Aslr on the line: Practical cache attacks on the mmu. In *Annual Network and Distributed System Security Symposium*, NDSS, 2017.
- [25] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space aslr. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 191–205, Washington, DC, USA, 2013. IEEE Computer Society.
- [26] Darius Kazemi. Counting uniforms in webgl. <https://bocoup.com/blog/counting-uniforms-in-webgl>.
- [27] Emre Kiciman and Benjamin Livshits. Ajaxscope: a platform for remotely monitoring the client-side behavior of web 2.0 applications. In *SIGOPS*, 2007.
- [28] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '96, pages 104–113, London, UK, UK, 1996. Springer-Verlag.
- [29] David Kohlbrenner and Hovav Shacham. On the effectiveness of mitigations against floating-point timing channels. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017.*, pages 69–81, 2017.
- [30] Balachander Krishnamurthy, Konstantin Naryshkin, and Craig Wills. Privacy leakage vs. protection measures: the growing disconnect. In *Web 2.0 Security and Privacy Workshop*, 2011.
- [31] Balachander Krishnamurthy and Craig Wills. Privacy diffusion on the web: a longitudinal perspective. In *Proceedings of the 18th international conference on World wide web*, pages 541–550. ACM, 2009.
- [32] Balachander Krishnamurthy and Craig E Wills. Generating a privacy footprint on the internet. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 65–70. ACM, 2006.
- [33] Balachander Krishnamurthy and Craig E Wills. Characterizing privacy in online social networks. In *Proceedings of the first workshop on Online social networks*, pages 37–42. ACM, 2008.
- [34] Pierre Laperdrix, Walter Rudametkin, and Benoit 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.
- [35] Adam Lerner, Anna Kornfeld Simpson, Tadayoshi Kohno, and Franziska 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.
- [36] Peng Li, Debin Gao, and Michael K. Reiter. Mitigating access-driven timing channels in clouds using stopwatch. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Budapest, Hungary, June 24-27, 2013*, pages 1–12, 2013.
- [37] Peng Li, Debin Gao, and Michael K. Reiter. Stopwatch: A cloud architecture for timing channel mitigation. *ACM Trans. Inf. Syst. Secur.*, 17(2):8:1–8:28, November 2014.
- [38] Zhichun Li, Yi Tang, Yinzhi Cao, Vaibhav Rastogi, Yan Chen, Bin Liu, and Clint Sbsa. Webshield: Enabling various web defense techniques without client side modifications. In *NDSS*, 2011.

- [39] Yali Liu, Dipak Ghosal, Frederik Armknecht, Ahmad-Reza Sadeghi, Steffen Schulz, and Stefan Katzenbeisser. Hide and seek in time - robust covert timing channels. In Michael Backes and Peng Ning, editors, *ESORICS*, volume 5789 of *Lecture Notes in Computer Science*, pages 120–135. Springer, 2009.
- [40] Keaton Mowery, Dillon Bogenreif, Scott Yilek, and Hovav Shacham. Fingerprinting information in javascript implementations. In *WEB 2.0 SECURITY & PRIVACY (W2SP)*, 2011.
- [41] Keaton Mowery and Hovav Shacham. Pixel perfect: Fingerprinting canvas in html5. In *W2SP*, 2012.
- [42] Martin Mulazzani, Philipp Reschl, Markus Huber, Manuel Leithner, Sebastian Schrittwieser, Edgar Weippl, and FC Wien. Fast and reliable browser identification with javascript engine fingerprinting. In *WEB 2.0 SECURITY & PRIVACY (W2SP)*, 2013.
- [43] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. Rendered insecure: Gpu side channel attacks are practical. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 2139–2153, New York, NY, USA, 2018. ACM.
- [44] Nick Nikiforakis, Wouter Joosen, and Benjamin Livshits. Privaricator: Deceiving fingerprinters with little white lies. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15, pages 820–830, New York, NY, USA, 2015. ACM.
- [45] Nick Nikiforakis, Alexandros Kapravelos, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *IEEE Symposium on Security and Privacy*, 2013.
- [46] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1406–1418, New York, NY, USA, 2015. ACM.
- [47] Xiang Pan, Yinzi Cao, and Yan Chen. I do not know what you visited last summer - protecting users from third-party web tracking with trackingfree browser. In *NDSS*, 2015.
- [48] M Perry, E Clark, and S Murdoch. The design and implementation of the tor browser [draft][online], united states, 2015.
- [49] Mike Perry, Erinn Clark, Steven Murdoch, and Georg Koppen. The design and implementation of the tor browser. <https://www.torproject.org/projects/torbrowser/design/>.
- [50] Jason Peterson. How to start building your own WebGL-based vr app. <https://medium.com/adventures-in-consumer-technology/how-to-start-building-your-own-webgl-based-vr-app-cdaf47b8132a>.
- [51] Ashay Rane, Calvin Lin, and Mohit Tiwari. Secure, precise, and fast floating-point operations on x86 processors. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 71–86, 2016.
- [52] Charles Reis, John Dunagan, Helen J. Wang, Opher Dubrovsky, and Saher Esmeir. Browsershield: vulnerability-driven filtering of dynamic html. In *OSDI: USENIX Symposium on Operating Systems Design and Implementation*, 2006.
- [53] Franziska Roesner, Tadayoshi Kohno, and David Wetherall. Detecting and defending against third-party tracking on the web. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 12–12, Berkeley, CA, USA, 2012. USENIX Association.
- [54] Michael Schwarz, Moritz Lipp, and Daniel Gruss. Javascript zero: Real javascript and zero side-channel attacks. In *NDSS*, 2018.
- [55] Peter Snyder, Lara Ansari, Cynthia Taylor, and Chris Kanich. Browser feature usage on the modern web. In *Proceedings of the 2016 Internet Measurement Conference*, IMC '16, pages 97–110, New York, NY, USA, 2016. ACM.
- [56] Deian Stefan, Pablo Buiras, Edward Z Yang, Amit Levy, David Terei, Alejandro Russo, and David Mazières. Eliminating cache-based timing attacks with instruction-based scheduling. In *European Symposium on Research in Computer Security*, pages 718–735. Springer, 2013.
- [57] Tom Van Goethem, Wouter Joosen, and Nick Nikiforakis. The clock is still ticking: Timing attacks in the modern web. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1382–1393, New York, NY, USA, 2015. ACM.
- [58] Tom Van Goethem, Mathy Vanhoef, Frank Piessens, and Wouter Joosen. Request and conquer: Exposing cross-origin resource size. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security, 2016.
- [59] Antoine Vastel, Pierre Laperdrix, Walter Rudametkin, and Romain Rouvoy. Fp-stalker: Tracking browser fingerprint evolutions.
- [60] Antoine Vastel, Pierre Laperdrix, Walter Rudametkin, and Romain Rouvoy. FP-Scanner: The Privacy Implications of Browser Fingerprint Inconsistencies. In *Proceedings of the 27th USENIX Security Symposium*, Baltimore, United States, August 2018.
- [61] Weiyi Wu and Bryan Ford. Deterministically deterring timing attacks in deterland. In *Conference on Timely Results in Operating Systems (TRIOS)*, 2015.
- [62] Zhihao Yao, Saeed Mirzamohammadi, Ardalán Amiri Sani, and Mathias Payer. Milkomeda: Safeguarding the mobile gpu interface using WebGL security checks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 1455–1469, New York, NY, USA, 2018. ACM.
- [63] Ting-Fang Yen, Yinglian Xie, Fang Yu, Roger Peng Yu, and Martin Abadi. Host fingerprinting and tracking on the web: Privacy and security implications. In *Proceedings of NDSS*, 2012.
- [64] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K. Reiter. Homealone: Co-residency detection in the cloud via side-channel analysis. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 313–328, Washington, DC, USA, 2011. IEEE Computer Society.
- [65] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-vm side channels and their use to extract private keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 305–316, New York, NY, USA, 2012. ACM.

Appendix

A Statistics of Collected Fingerprints

In the appendix, we show some statistics about collected fingerprints using the WebGL tasks provided by the original Cao et al.'s website. Figure 11 shows the anonymous set for the collected data, which is broken down into three different browsers. The size of anonymous set is relatively small—if we limit it to be three, we include 77% of all the collected fingerprints. The largest anonymous set is just with about 10 fingerprints. Among all the browsers, Safari is the most fingerprintable as compared with others: It is probably also because the number of Safari users is relatively small.

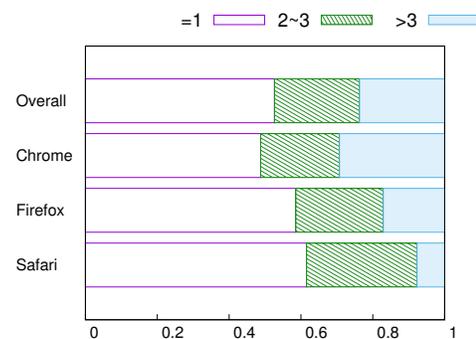


Figure 11: Anonymous Set for Collected Fingerprints