Pixel Perfect: Fingerprinting Canvas in HTML5

Keaton Mowery and Hovav Shacham
Department of Computer Science and Engineering
University of California, San Diego
La Jolla, California, USA

ABSTRACT

Tying the browser more closely to operating system functionality and system hardware means that websites have more access to these resources, and that browser behavior varies depending on the behavior of these resources.

We propose a new system fingerprint, inspired by the observation above: render text and WebGL scenes to a `<canvas>` element, then examine the pixels produced. The new fingerprint is consistent, high-entropy, orthogonal to other fingerprints, transparent to the user, and readily obtainable.

1. INTRODUCTION

Browsers are becoming increasingly sophisticated application platforms, taking on more of the functionality traditionally provided by an operating system. Much of this increasing sophistication is driven by the HTML5 suite of specifications, which make provisions for a programmatic drawing surface (`<canvas>`), three-dimensional graphics (WebGL), a structured client-side datastore, geolocation services, the ability to manipulate browser history and the browser cache, audio and video playback, and more.

The natural way for browsers to implement such features is to draw on the host operating system and hardware. Using the GPU for 3D graphics (and even for 2D graphics compositing[1]) provides substantial performance improvements, as well as battery savings on mobile devices. And using the operating system's font-rendering code for text means that browsers automatically display text in a way that is optimized for the display and consistent with the user's expectations.[2]

This paper proceeds from the following simple observation: Tying the browser more closely to operating system functionality and system hardware means that websites have more access to these resources, and that browser behavior varies depending on the behavior of these resources. The first part of this observation has security implications: codebases not designed to handle adversarial input can now be exposed to it.[3] The second part of the observation, which we focus on, has privacy implications: different behavior can be used to distinguish systems, and thereby fingerprint the people using them.

Our results.

We exhibit a new system fingerprint based on browser font and WebGL rendering. To obtain this fingerprint, a website renders text and WebGL scenes to a `<canvas>` element, then examines the pixels produced. Different systems produce different output, and therefore different fingerprints. Even very simple tests — such as rendering a single sentence in a widely distributed system font — produce surprising variation. The new fingerprint has several desirable properties:

- *It is consistent.* In our experiments, we obtain pixel-identical results in independent trials from the same user.
- *It is high-entropy.* In 294 experiments on Amazon's Mechanical Turk, we observed 116 unique fingerprint values, for a sample entropy of 5.73 bits. This is so even though the user population in our experiments exhibits little variation in browser and OS.
- *It is orthogonal to other fingerprints.* Our fingerprint measures graphics driver and GPU model, which is independent of other possible fingerprints discussed below.
- *It is transparent to the user.* Our tests can be performed, offscreen, in a fraction of a second. There is no indication, visual or otherwise, that the user's system is being fingerprinted.
- *It is readily obtainable.* Any website that runs JavaScript on the user's browser can fingerprint its rendering behavior; no access is needed besides what is provided by the usual web attacker model.

Our fingerprint can be used as a black box or as a white box. A website could render tests to a `<canvas>`, extract the resulting pixmap, then use a cryptographic hash to obtain a short, convenient fingerprint. Because the fingerprint is consistent, the pixmap (and therefore its hash) will be identical in multiple runs on one machine, but take on different values depending on hardware and software configuration. This is a black-box use of the fingerprint, since it extracts

[1] For example, IE9 uses the GPU for compositing, and recent releases of Chrome use the GPU to accelerate 2D operations on the canvas.

[2] By contrast, the first release of Safari for Windows imported font rendering code from Mac OS X, which offended some users; see `http://www.joelonsoftware.com/items/2007/06/12.html`.

[3] Indeed, one test in the WebGL conformance suite induces a hard system crash on many systems [8]; and the TrueType font handling code in Windows and OS X, which is exposed to attackers by the WebFont specification, was patched to fix an exploitable parsing vulnerability as recently as December of last year [13, 4].

distinguishing entropy without being concerned with the implementation details.

Alternatively, a website could use a particular test pixmap as evidence that a user is running some particular configuration of browser, operating system, graphics driver, GPU, and, perhaps, display. To identify a user system, the site can compare the pixmap it produces against a labeled corpus, such as the corpus we obtained using Mechanical Turk. An intriguing possibility is that GPU quirks could be used to identify a pixmap without comparing against a corpus. However it is performed, such a white-box use of our fingerprint in this way reveals private information about users' systems.[4] It could also be used to target an attack more precisely, by identifying specific vulnerable system configurations. Trying to exploit only those systems that appear likely to be vulnerable could reduce the number of crashes caused by the attack, and therefore the likelihood that it is detected by the operating system vendor.

Fingerprints on the web have constructive and destructive uses [14]. A use is constructive if users benefit from being fingerprinted. For example, a bank could fingerprint a user's machine, then require additional authentication for login attempts from systems whose fingerprint does not match. A use is destructive if users do not benefit from being tracked, or do not wish to be tracked. Users can attempt to avoid tracking by using their browsers' "private browsing" modes [1] or the Tor anonymity service [5].

Users of Tor may be willing to endure a slower, less attractive browsing experience to avoid being tracked. (Note that, although Torbutton disables WebGL, it allows text rendering to a `<canvas>`, and is thus at present partly vulnerable to our fingerprint.) For mainstream browser users, however, the possibility of fingerprinting might be an unavoidable consequence of browsers' closer ties to operating system functionality and system hardware.

Related work: Fingerprints on the web.

The earliest mentions known to us of using differences in GPU rendering to fingerprint users are in 2010 discussions on the WebGL mailing list about whether the WebGL renderer information available to JavaScript should provide information about the GPU and driver. Steve Baker argued [2] that it is possible to identify a GPU without this information: "I bet that if I wrote code to read back every glGet result and built up a database of the results - and wrote code to time things like vertex texture performance - then I bet I could identify most hardware fairly accurately." Benoit Jacob later observed [10] that

> We haven't yet started accounting for GPU rendering analysis (not just WebGL: in the upcoming generation of browsers, most rendering goes through the GPU and is subject to GPU/driver/config-based rendering differences.

Jacob also suggests the fingerprinting approach we take: "Rendering analysis could proceed by rendering stuff into a canvas 2D and getting its ImageData." One way to view our research is as demonstrating experimentally that Baker and Jacob were correct in expecting substantial additional leakage from GPU-based rendering. In addition, we show that there is substantial information leakage from font rendering to `<canvas>`.

Many other researchers have proposed techniques to fingerprint web users. These techniques rely on many browser features, including the history and file cache [11], information in HTTP headers and available plugins [12, 6], differences in JavaScript and DOM API support [7], JavaScript performance [14], available fonts [3], and deviations from JavaScript standards conformance [16].

2. HTML5 AND CSS3

In this section, we introduce the emerging web technologies used in our experiments. First, we present information about the `<canvas>` element, a major portion of what is termed HTML5[5], along with its support for text rendering. Next, we examine WebFonts, part of the CSS3 specification[6]. Lastly, we briefly discuss WebGL, an experimental specification[7] currently managed by the Khronos Group (which also maintains the OpenGL specification).

These three specifications are not finalized, and so could change in ways that benefit or hinder fingerprinting success. However, our fingerprinting mechanisms use extremely basic features of these platforms, such as rendering text and inspecting pixels — removal of these features would be dramatic indeed.

2.1 HTML5 Canvas

One of the most interesting new elements in HTML5, `<canvas>` provides an area of the screen which can be drawn upon programmatically. It enjoys widespread support, being available in the most recent versions of Chrome, Firefox, Internet Explorer, Opera, and Safari as well as Mobile Safari and Android Browser.

The basic approach to drawing on a canvas is simple: acquire a graphics context, and use the context's API to effect your changes. In the current HTML5 specification, the only defined context is "`2d`". The `2d` context provides basic drawing primitives such as `fillRect`, `lineTo`, and `arc`, as well as more complicated features such as Bézier curves, color gradients, and copying in an existing image.

2.1.1 Canvas Text

We chose to focus on the text support found in the `2d` context. Given a font size, family, and baseline, the `2d` context can draw any arbitrary text string to the canvas. No wrapping is performed; the `2d` context will happily draw text directly off the edge of the canvas. Lastly, `<canvas>` supports CSS-like text styling, allowing for any combination of font and size. For an example of how text is rendered, see Figure 1.

2.1.2 Pixel Extraction

In order for `<canvas>` to be a useful fingerprint, there must be some way to examine its behavior. Fortunately, `<canvas>` makes this extremely easy, providing several ways to inspect its data with pixel accuracy.

[4] As evidence that such information is private, we note that Chrome knows a great deal about the graphics subsystem — see `chrome://gpu` — but does not expose this information to JavaScript.

[5] `http://www.whatwg.org/specs/web-apps/current-work/`

[6] `http://www.w3.org/TR/css3-fonts/`

[7] `http://www.khronos.org/registry/webgl/specs/1.0/`

```
<script type="text/javascript">
  var canvas = document.getElementById("drawing");
  var context = canvas.getContext("2d");
  context.font = "18pt Arial";
  context.textBaseline = "top";
  context.fillText("Hello, user.", 2, 2);
</script>
```

Figure 1: Render text on a canvas

First, the `2d` context provides the method `getImageData()`. Given a rectangular region of the canvas, this method returns an `ImageData` object. Contained in this object are the RGBA values (as integers) for every pixel in the requested region.

Second, the canvas object itself provides a `toDataURL(type)` method. When passed "image/png", this method returns a data url consisting of the Base64 encoding of a PNG image containing the entire contents of the canvas. As this is a very convenient canvas-level method, we used this approach to extract data in our experiments. During black-box use of these fingerprints, the test suite could simply hash these data URLs, thereby removing the need to upload entire images from each client.

It is worthwhile to note that these methods do preserve the same origin policy — if an image from a different origin has been drawn on this canvas, they will throw a `SecurityError` exception instead of returning pixel data. Therefore, our `<canvas>` fingerprints must only contain image resources that are under our control.

2.2 WebFonts

WebFonts, specified in CSS3, allow web designers to load a font face on-demand, rather than relying solely on the fonts installed on each client machine. To include a font, the web designer inserts a `@font-face` CSS rule with a `src` attribute pointing to a font in an appropriate format. The browser then downloads the font and makes it available for use on the page. Fortunately for us, web fonts can be used when writing to a `<canvas>` as well.

To include WebFonts, we depend on the WebFont Loader[8], co-developed by Google and Typekit. With this library, WebFonts can be loaded solely through the use of JavaScript, and callbacks can be established for certain events (such as the font becoming available or, conversely, failing to load). By attaching our rendering to a successful load, we are guaranteed to use the correct font while writing to the canvas.

2.3 WebGL

WebGL provides a JavaScript API for rendering 3D graphics in a `<canvas>` element. Modeled after OpenGL ES 2.0, WebGL is currently a draft specification and implemented and enabled in Chrome, Firefox, and Opera, as well as implemented but disabled in Safari. Each of these browsers provides a hardware-accelerated implementation, using the installed graphics hardware to render each frame. To mitigate serious misbehaviour and crashes, all of these browsers enable WebGL only for a whitelisted set of graphics cards and drivers.

[8] `https://developers.google.com/webfonts/docs/webfont_loader`

Current WebGL implementations expose their functionality through a separate canvas context (which will eventually be named "`webgl`"). The WebGL API is too complex to describe here in sufficient detail, but is stylistically similar to the desktop OpenGL API. It provides for vertex and fragment shaders, written in OpenGL Shading Language (GLSL), that, after compilation, run directly on the graphics card. WebGL also provides for OpenGL-style textures, as well as different lighting primitives. More advanced techniques, such as specular highlighting, bump mapping, and transparency, can be achieved through custom GLSL shaders.

2.4 Security Implications

These new capabilities, while providing more and more ways for developers to produce interesting and useful web content, do come at a cost. For efficiency's sake, inputs from the web are passed farther and farther down the software stack: for example, GLSL shaders are compiled directly from web pages and run on the graphics card, allowing arbitrary data to pass between the JavaScript execution engine and the kernel-level graphics driver. Other attack surfaces are possible: malicious or misguided GLSL shaders can crash or hang the entire operating system on OSX and Windows XP or cause GPU resets on Windows 7[8].

WebFonts, while appearing more innocent, can also be a security concern. Remote code execution vulnerabilities while parsing TrueType fonts have been discovered in Windows[13], OSX, Debian, Red Hat, and iOS[4].

While we do not use these exploits in this paper, we take advantage of the fact that these new web technologies, for efficiency's sake, push untrusted web content deep into the operating system stack. In our case, however, we simply examine the results of these operations, exposing differences in implementation (however slight).

3. EXPERIMENTS

In this section, we discuss the tests that underly our fingerprinting scheme, as well as the support infrastructure we built in order to deliver the tests and inspect their results. We will also detail the process of fingerprint collection from a large number of disparate users on the web.

3.1 Tests

For our fingerprints, we use six tests: `text_arial`, `text_arial_px`, `text_webfont`, `text_webfont_px`, `text_nonsense`, and `webgl`. Each test follows the same basic outline: render test data to a canvas and extract its contents as an encoded PNG.

3.1.1 Arial Text

In our first two tests, we render a short sentence in Arial, a font known for its ubiquity on the web. To exercise each letterform, we use the pangram "How quickly daft jumping zebras vex.", along with some added punctuation.

For `text_arial`, the text is rendered to the canvas in 18pt Arial. In `text_arial_px`, we change the font specification to 20px Arial. The actual code for these two tests is almost identical to the snippet in Figure 1 — complicated tests aren't needed for fingerprinting!

Example images produced by these two tests are shown in Figure 2.

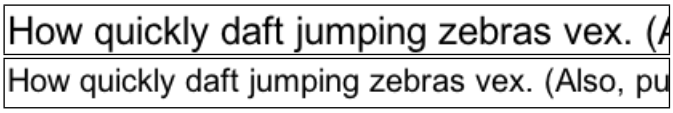


Figure 2: `text_arial` (top) and `text_arial_px`

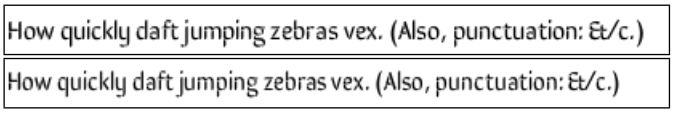


Figure 3: `text_webfont` (top) and `text_webfont_px`

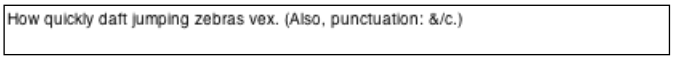


Figure 4: `text_nonsense`

3.1.2 WebFont Text

These two tests are extremely similar to the Arial tests, with the added complexity of loading a new font from a web server. In a more sophisticated or targeted fingerprint, the delivered font could be carefully tuned by the fingerprinter to exercise corner cases in font loading.

In our case, however, we use the WebFont Loader to load "Sirin Stencil" from the Google Web Fonts server[9]. Once it loads, we render the same pangram as in our Arial tests. For `text_webfont`, the text is set in 12pt Sirin Stencil, while `text_webfont_px` uses 15px Sirin Stencil.

Example images are shown in Figure 3.

3.1.3 Nonsense Text

Code-wise, this test is nearly identical to the two Arial tests. However, instead of a valid font specification, we set the `2d` font specification to "not even a font spec in the slightest". This exercises the fallback handling mechanisms in the browser: what does it do with an invalid font request? The browser's choice of fallback font, as well as its positioning and spacing, can be quite telling.

Also, note that this behavior is also the fallback font handling mechanism for when the browser is presented with a valid font specification for an unavailable font. Using this technique, tests can be written to probe for the existence of a particular font on target machines. If enough of these tests are run, the fingerprinter can derive a fairly comprehensive list of the installed fonts on the target machine.

An example output is shown in Figure 4.

3.1.4 WebGL

`webgl` is our only test whose code spans more than a few lines. As WebGL scenes go, however, this scene is almost minimal. We create 200 polygons, approximating the hyperbolic paraboloid $z = \frac{y^2}{2} - \frac{x^2}{3}$, with $-3 \leq y \leq 3$ and $-3 \leq x \leq 3$. Over this surface, we apply a single texture: a 512 by 512 pixel rasterized version of ISO 12233, the ISO standard for measuring lens resolution. Designed for measuring sharpness and resolution in electronic still-picture cameras, this texture contains many areas with high detail. We then add an ambient light with color (0.1, 0.1, 0.1) and a directional light of color (0.8, 0.8, 0) and direction (2,4,9). Placing our surface at $z = -10$, we render this simple tableau.

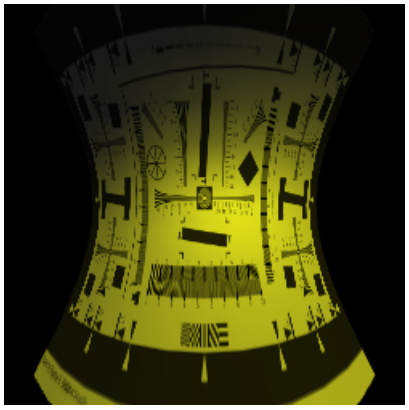


Figure 5: An example run of the `webgl` test

A example, rendered on OS X 10.7.3 with Chrome 18 on a AMD Radeon HD 6490M, is shown in Figure 5.

3.1.5 Test Speed

Speed is an important characteristic for fingerprints. Tests that take minutes are categorically less useful than tests that take seconds, especially if they are deployed to protect online accounts (imagine waiting even twenty seconds to log into your webmail!). In our case, each test takes a mere fraction of a second to run — indeed, the longest delays occur while fetching the image assets. Consisting of a 76 KiB PNG image and a 24 KiB WOFF font, these fetches would not be out of place during any page load on today's web. For comparison, once it has loaded, the Quake 3 WebGL Demo[10] runs at 60 FPS with around 15% CPU utilization in Chrome 18 with a 2 GHz Core i7 and an AMD Radeon HD 6490M. There is room for a substantial number of `<canvas>`-based fingerprints to run before adversely affecting user experience.

3.2 Infrastructure

In general, web designers can depend on their sites rendering in a consistent manner across various browsers and operating systems. Therefore, we expected that any fingerprintable differences will be subtle, perhaps not even visible to a human observer.

To view these trace differences, we built a small webapp which can administer the tests and examine their results. Experiments are served as pure JavaScript, and results are collected as data URL-encoded PNGs. Our framework then compares these results as images, allowing it to group identical results and display pixel-level differences between these groups.

We use two types of image comparison: pixel-level difference and difference maps. When constructing a pixel-level difference, the framework first creates a new image of the appropriate size. Then, each pixel's color is set to the channel-wise difference between the two images at that location. If this color is anything other than transparent pure black (which indicates that there is no difference between the two images at this pixel), we set the alpha value of the differing pixel to 255, rendering it fully opaque. For difference maps, each pixel in the map is set to either white or

[9] `http://www.google.com/webfonts`

[10] Quake 3 Demo: `http://media.tojicode.com/q3bsp/`

black, depending on whether the original images differ. A purely white difference map indicates identical images, while perfectly black indicates difference in every pixel.

Unfortunately, we do not have ready access to several hundred distinct consumer-level computer systems. To collect enough data to demonstrate the applicability of our fingerprints, we turned to Amazon Mechanical Turk. We modified our framework to deliver multiple tests on a single page, along with extended instructions for the human workers on Mechanical Turk. The next section explains our process and results more thoroughly.

3.3 Data Collection

We collected samples from 300 distinct members of the Mechcanical Turk marketplace, paying each a small sum to report their graphics card and graphics driver version. Meanwhile, our five fingerprinting tests ran in the background. We also collected various metadata, such as the browser's user agent string and the WebGL-reported renderer, vendor, and version.

In Safari, acquiring information on the user's graphics card and driver is trivial: simply ask the WebGL canvas context. For example, given a WebGL context `gl`, calling `gl.getParameter(gl.RENDERER)` on Safari 5.1.3 might return "AMD Radeon HD 6490M OpenGL Engine", while calling `gl.getParameter(gl.VERSION)` might return "WebGL 1.0 (2.1 ATI-7.18.11)", which includes the version number of the current graphics driver. Currently, however, WebGL is disabled by default in Safari, and must be manually enabled through the Developer menu.

In all three browsers which ship with WebGL enabled, however, these values are redacted. For example, in Chrome 18.0.1025.39, asking for the WebGL renderer returns "WebKit WebGL", while the version is "WebGL 1.0 (OpenGL ES 2.0 Chromium)". Firefox and Opera Next are similarly unhelpful, returning generic statements about the version of WebGL, without reference to the installed hardware. We conclude that browser vendors consider hardware and driver version to be identifying information, and any information our fingerprints extract about them can be considered a loss of privacy.

Owing to these constraints, the user-facing portion of our survey asked users of Chrome, Firefox, and Opera to manually report their graphics card and driver. For Chrome and Firefox, the user was instructed to copy text from the browser pages `chrome://gpu` and `about:support`, respectively. Opera does not appear to have any such mechanism for hardware discovery, and so these users were asked to discover the information through other means.

3.3.1 Platform Representation

Windows represents the lion's share of user platforms, with 276 samples. A full 226 of these are from Windows 7, with 9 Windows Vista, 40 Windows XP, and 1 Windows 8 rounding out the total. We also have 13 samples from OS X, ranging from 10.5.8 to 10.7.3, and 11 samples from Linux.

Chrome is overwhelmingly present in our data set, with 222 samples ranging from version 10 to 19. Firefox comprises almost all of the rest, with 71 samples between versions 8 and 11. We also have 4 samples from Opera 9.8, 2 from Safari 5, and 1 from Android.

These numbers do not represent the current software usage of the internet as a whole. We attribute this to text in the survey, asking users to use a WebGL-enabled browser to complete the task. Notably, since Internet Explorer does not support WebGL, we posit that every Mechanical Turk user using IE either skipped our survey, or returned in either Chrome or Firefox. The overwhelming prevalence of Windows 7 is also correlated with WebGL capability — users who have up-to-date browsers might also be more likely to have up-to-date operating systems.

3.3.2 Persistence

During the course of data collection, twenty-three distinct users performed our survey twice. Due to our setup, we collected results for `text_arial`, `text_webfont`, `text_nonsense`, and `webgl` when they first performed the survey. However, for all twenty-three users, each of these tests were identical to their later submissions. This suggests that our fingerprints are consistent — for a given browser/OS/graphics card platform, performing the same test will always give identical results.

To further test this hypothesis, we ran all of our tests on 5 identically-provisioned lab computers, running Firefox 11 on Windows XP. As expected, all five computers produced identical results on each test, providing further evidence that our fingerprints are stable across identical hardware and software stacks. Additionally, we note that our fingerprints are unable to distinguish between users who use the exact same hardware and software.

3.3.3 Errors

While running the experiment, six users experienced failures in the `text_webfont` and `text_webfont_px` tests, returning a blank PNG instead of one containing text. Upon investigation, we attribute these failures to a known race condition in the WebFont Loader library. Since the race condition is a transient error, we ignore these six samples whenever they affect our fingerprint information leakage, as including them will improve differentiation.

3.3.4 Data Quality

We treat the user's report and user agent string as ground truth. While examining the data, we saw no evidence of any forgery, either of user agent strings or graphics card information.

Since we conducted the survey on Mechanical Turk, however, some level of imperfection is to be expected. Indeed, twenty-four users did not submit their graphics card information, either by copying unrelated text into the survey box or failing to fill in the box at all. In the authors' favorite submission, the worker simply entered "It was very nice." into the text field. These unclassified samples are included in our results, since they represent the state of a `<canvas>` fingerprint in the wild, with no inside information about hardware.

4. RESULTS

The most important feature of any fingerprint is differentiation: if every system fingerprinted performs identically, what use is the fingerprint? With this goal in mind, we now examine the results of our tests as applied to Mechanical Turk users.

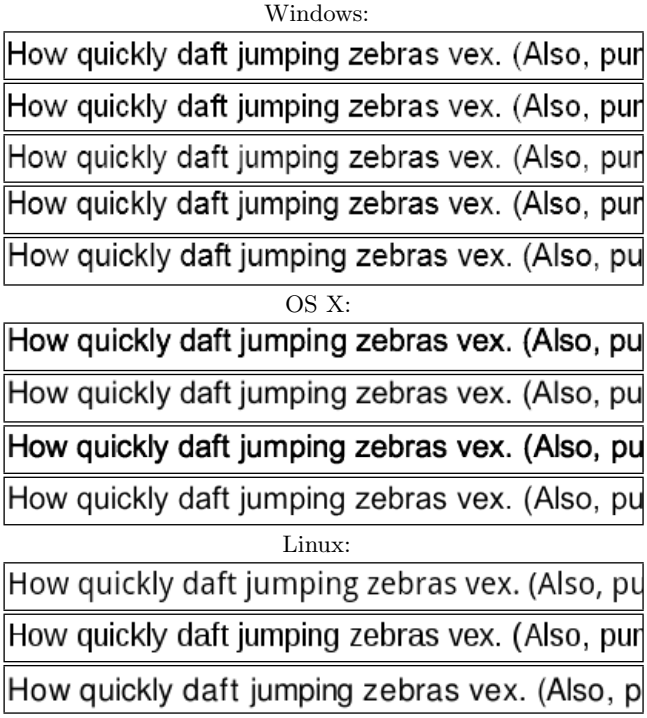


Figure 6: 13 ways to render 20px Arial

4.1 Arial Font Rendering

In general, we find a surprising amount of differentiation in the ways that fonts are rendered. Even Arial, a font which is 30 years old, renders in new and interesting ways depending on the underlying operating system and browser. In the 300 samples collected for the `text_arial` test, there are 50 distinct renderings. For `text_arial_px`, this number drops to 43. This amount of diversity is astounding, showing differences even between computers running the same operating system and browser version. In Figure 6, you can see some of the different results that appear. Interestingly, some of the Linux samples shown are not using Arial at all, but are substituting an unidentified yet similar font. Font subsitutions such as these provide an extra dimension of distinguishability between otherwise matching computers.

The largest cluster of identical renderings on `text_arial` contains 172 samples, taken in versions of Chrome ranging from 15 to 18 on Windows XP, Vista, or 7. We attribute the size of this group to the relative popularity of this browser/OS combination in our data set. However, seven other groups contain samples taken on this platform as well, indicating that other, more hidden variables might be discoverable in this test.

4.1.1 Classification

While a repeatable, trivial fingerprint borne out of simple text rendering is quite promising, even more information can be derived with this test. 10 of 50 groups contain samples taken on Linux machines, while 5 contain samples from OS X. Each of the fifty groups contains samples from only a single operating system family, implying that this fingerprint alone is sufficient to distinguish operating systems!

Similarly, almost every group contains samples from only a single browser family. The single exception occurs on OS X, where Chrome and Safari can produce identical renderings. Otherwise, the font handling in each browser is distinctive enough to leave a usable fingerprint.

Given these results, we conclude that rendering a simple pangram in Arial on a `<canvas>` is enough to leak the user's operating system family and (almost always) browser family.

4.1.2 Differences

Collecting text samples from such a wide variety of sources allows us to find slight differences among rendering engines, which could be exploited for better and more precise fingerprints. In Figure 7, we present the original rendering for the 172-member Windows/Chrome group, along with several difference maps. The most obvious difference occurs across platforms, where we see a marked difference in the kerning of the text, leading to the text rendering at two different lengths. For samples on the same platform, the difference map shows the outline of each letter, indicating that antialiasing or subpixel hinting (such as ClearType) was used. From this, a fingerprinter might be able to deduce information regarding the user's display or ClearType settings. Finally, the most interesting difference map comes from a single sample, claiming to be Chrome 17.0.963.56 on Windows 8. Here, we see only a few pixels' difference, located solely near round edges, indicating subtle differences in the font rendering engine in this system. Such differences might be found in currently deployed systems by using other fonts and glyphs, rendering this sort of fingerprint even more potent.

More generally, our technique is capable enough to flush out any differences between two font handling stacks. During our experiments, we observed that at least operating system, browser version, graphics card, installed fonts, subpixel hinting, and antialiasing all play a part in generating the final user-visible bitmap. Additionally, our fingerprint can recognize the impact of any other variables that impact font rendering, such as the exact placement of pixels on a physical LCD screen (which ClearType might take into account). More data and better platform characteriziation is necessary for identification and individual classification of these more subtle variables.

Lastly, in both Figure 7 and Figure 6, there is a surpsrising amount of diversity in the length of rendered text. This suggests an even simpler fingerprinting mechanism: create specially crafted sentences as DOM elements and measure their length via JavaScript. When implemented as a proof-of-concept, this technique shows a measurable and repeatable difference in the length of text between Firefox 10 and Chrome 18 on OSX. This simpler fingerprint should reveal a strict subset of the information that our `<canvas>`-based text rendering does, but does so using far simpler methods (and thus will be much more difficult to defend against).

4.1.3 Entropy

Fingerprints are useful only inasmuch as they differentiate users. Since our fingerprints reveal differences in hardware and software stacks, popular hardware configurations reduce the identifying power of the tests. Conversely, however, some setups might produce unique results, identifying their user precisely. To estimate the tests' overall effectiveness, we will compute the distribution entropy of our groupings. This metric indicates how many predictive bits the test reveals, or, more precisely, the differential loss of user

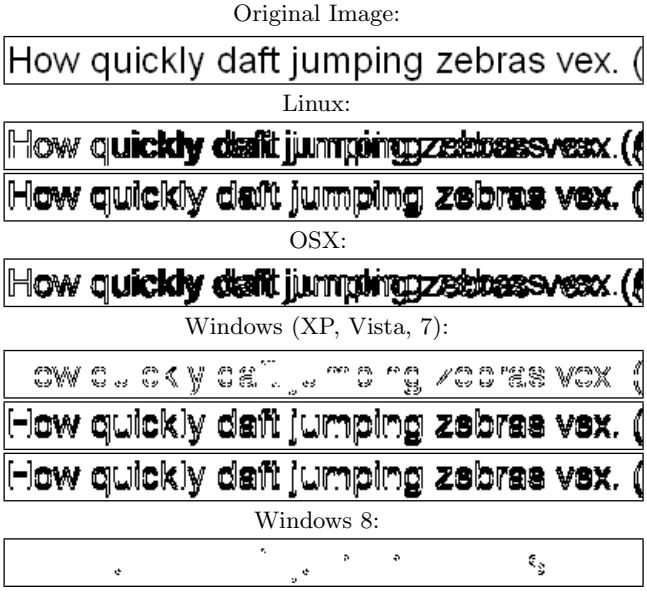


Figure 7: Difference maps for a group on `text_arial`

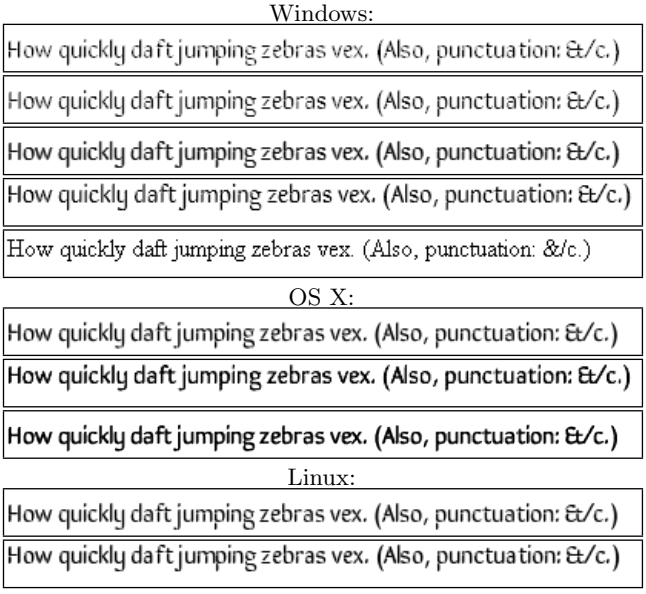


Figure 8: 10 ways to render 12pt Sirin Stencil

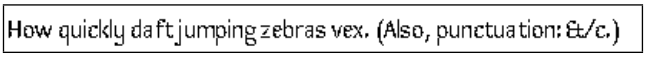


Figure 9: Sirin Stencil

privacy resulting from the test. However, since we do not believe that the hardware and software in our 300 samples is a representative sample of the internet as a whole, these metrics should be treated as rough guidelines at best.

To measure distribution entropy, we use the formula

$$E = -\sum_{i=1}^{n} p(x_i) log_2 p(x_i)$$

where $p(x_i)$ is the size of the ith group divided by the number of samples. `text_arial`, across 50 groups, shows a distribution entropy of 3.05 bits. `text_arial_px`, having only 43 distinct groups, reveals 2.86 bits. We note that these numbers do not represent the true entropy of these tests, as we do not have enough data to accurately model the distribution of platforms in the wild.

4.2 WebFont Rendering

Using WebFonts allows us to standardize the font under test. Unlike in the Arial tests, where slight differences in the installed font could conceivably exist, any differences in the WebFont test must be a direct consequence of the font engine used to render the text.

In general, `text_webfont` shows a similar distribution to the Arial tests. From the 294 properly completed tests, there are 45 distinct ways to render our sample sentence. 10 of these are presented in Figure 8. As in the Arial tests, each group consists of a single OS family/browser family pair, with the sole exception of Safari and Chrome on OS X, lending further evidence that text rendering can uniquely identify platform details.

Interestingly, as is visible in the last Windows sample, some clients did not use Sirin Stencil at all; rather, they substituted in a font of their own. In our data, we found five of these samples: four (Chrome 16 and 17, Firefox 10 and 11) using Times New Roman, the fifth (Opera 9.8) using what appears to be Arial. Samples containing the correct webfont exist for each of these browsers, so we must assume that these five users have disabled WebFonts for security reasons, or that there was an error loading the font. If the former, these users may be doing themselves a disservice: their browsers stand out quite strongly in our fingerprints. Indeed, three of these samples are unique, with only the Chrome pair sending identical results.

`text_webfont_px` shows almost identical results, with 44 groups from 294 samples. Therefore, we shall focus only on `text_webfont` from now on.

As in our Arial tests, the largest group consists solely of Chrome on Windows, with 164 identical samples. Oddly, the Windows 8 sample appears in this group, along with many of the samples from the large Arial group. This suggests that the font handling in Windows 8 has not been entirely changed, and similar subtle differences might exist in other, currently indistinguishable font engines.

Another interesting render is shown in Figure 9. 13 separate users submitted this result, each using Chrome (17, 18) on Windows (XP, Vista, 7). After being surprised at the relative poor quality of this text, we were able to reproduce it exactly by disabling the option "Smooth edges of screen fonts" in the Windows 7 Performance Options preferences, which disables Microsoft's ClearType subpixel hinting, as well as any antialiasing on fonts system-wide. Our `text_webfont` test, therefore, leaks this setting to an online fingerprinter, and this leakage suggests that other ClearType configuration might be detectable as well. For example, if ClearType performs differently with distinct screen DPIs or LCD pixel layouts, these subtle differences will reveal themselves to this very simple fingerprinting technique, leaking even more information about the user's hardware.

4.2.1 *Entropy*

`text_webfont`, with 294 samples in 44 groups, shows a distribution entropy of 2.93 bits. `text_webfont_px` is similar, at 2.95 bits.

4.3 WebGL

When we first started this project, we predicted that we would need to try quite a few underhanded tricks in order to see differences between graphics cards. Surprisingly, this is not the case. Our experiments show that graphics cards leave a detectable fingerprint while rendering even the simplest scenes.

As described in Section 3.1.4, our WebGL test creates a single surface, comprised of 200 polygons. It applies a single black and white texture to this surface, and uses simple ambient and directional lights. We also enable antialiasing.

Of the 300 users who participated in our study, 30 submitted no data for this test. In our framework, this absence indicates either that WebGL is disabled or that an error occurred during the test.

Under visual inspection, the 270 remaining images appear identical. When examined at the level of individual pixels, however, we discovered 50 distinct renders of the scene.

This level of heterogeneity is, frankly, quite surprising. Our scene is rendered with basic matrix operations, and we expected far more consistency among graphics cards. One possible explanation suggests that graphics cards, in the name of efficiency, cut corners with respect to graphics processing. Perhaps renders are nondeterministic, but in such minor ways as to be undetectable for humans.

Looking at the subset of our data in Table 1, we see that this is not the case: most graphics cards produce pixel-perfect output as compared to others of their model. There is also a resemblance among the members of the same line (note groups 1, 5, and 24), suggesting that graphics card manufacturers perhaps share hardware or driver implementations between coexisting and evolving product lines.

The graphics cards, browsers, and operating systems present in all 51 groups can be found in Appendix A.

4.3.1 Classification

Of course, simply knowing that two implementations differ is useful, but understanding how might give clues as to why. In Figure 10, we see the original image for group 24, our largest, as well as its difference maps against several other renders.

Examining these in detail, there are several different ways in which these groups differ. In the group 1 and group 36 difference maps, we see that most of the difference is located at the edges of color regions and polygons. This suggests that these graphics cards are performing antialiasing slightly differently, or perhaps simply linearly interpolating textures in almost imperceptably different ways. In contast, we see that the renderers in Group 20 produced slightly different colors when lighting the white portions of the texture, as compared to group 24. However, group 23 differs at almost every single non-background pixel. The two renders are visually indistinguishable, suggesting that the differences occur in the very least significant bits of each color.

The most interesting difference, however, appears between group 24 and group 25. These renders differ by only a few pixels! Indeed, the hardware used to produce these images is extremely similar: group 24 consists mainly of Intel G41 (device ID 0x2e32), Intel HD Graphics (device IDs 0x0042, 0x0046, 0x0df1), and Mobile Intel 4 Series (device ID 0x2a42), while group 25 consists solely of Intel HD Graphics systems (device IDs 0x0102 and 0x0116). If we assume that distinct device ID numbers indicate distinct products, these few pixels strongly suggest that even extremely similar graphics systems can be differentiated simply through rendering the right images!

Also, it is worth noting that out of 50 total groups, 49 contain samples from a single operating system family. Group 42 (in 1 is the only exception, containing both Linux and OS X. However, given the relative lack of samples from Linux and OS X, we cannot determine whether this delineation is due to significant differences in each operating system's graphics handling, or whether we simply do not have enough samples comparing identical graphics cards across diverse software stacks.

4.3.2 Entropy

Due to the large number of unique graphics cards and their consistent effects, `webgl` gives a distribution entropy of 4.30 bits, over 300 samples in 51 groups. Again, the actual entropy revealed by this test depends upon the frequency of each graphics card as deployed in the wild, which we can not extrapolate from our data. Therefore, this entropy should be considered a rough estimate, at best.

4.4 Comprehensive Fingerprinting

Given the relative success of each individual test, we shall now examine their efficacy when combined. If their predictive power lies solely in browser and operating system family, we will expect a relatively similar number of distinct groups once all fingerprints are combined. With this approach, each group consists of samples for which all six tests are identical.

Among the 294 samples which successfully completed all six tests, there are 116 distinct groups. The largest of these contains 51 samples, and consists almost solely of Chrome 17.0.963.56 on Windows 7, with a single Windows Vista mixed in. As for graphics cards, it contains Intel G41 Express Chipsets (DID 0x2e32), Intel Graphics Media Accelerator HD 0x0046), Intel HD Graphics (DID 0x0042, 0x0046, 0x0df1), Intel 4 Series (DID 0x2a42), and Intel 45 Express (DID 0x2a42). As mentioned in Section 4.3.1, further and more sophisticated WebGL fingerprints may be able to differentiate these graphics systems.

4.4.1 Entropy

Overall, our fingerprints do combine beneficially: among the 116 groups, our five extremely simple tests show a distribution entropy of 5.73 bits. We believe that more specialized and targeted tests could reveal even more, perhaps down to the exact installed graphics card, operating system, and browser family.

5. DEFENSES

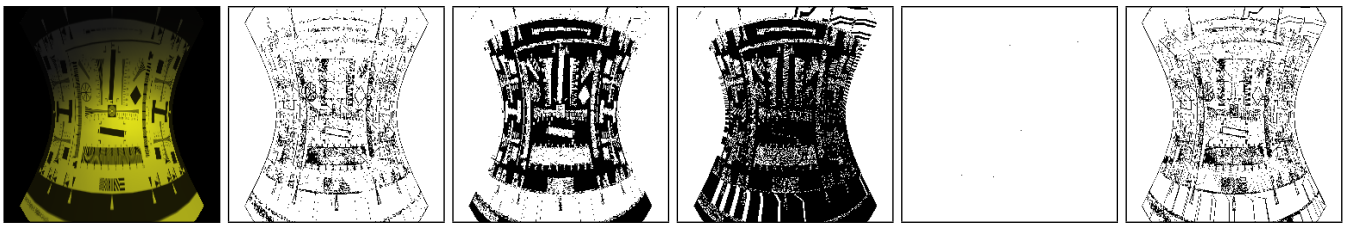
In this section, we propose several methods of preventing `<canvas>`-based fingerprinting and consider their impact.

First, browser vendors could completely disable canvas pixel extraction. While obviously preventing any potential `<canvas>`-based fingerprinting, this fix removes a useful capability of the platform — imagine building a webapp for photo editing or drawing. Therefore, let us only consider defends that do not overly undermine the potential of `<canvas>`.

One might imagine a defense whereby the browser adds random pixel noise whenever pixels are extracted. Under this regime, directly comparing image results becomes far more difficult. However, slight noise can be easily circum-

Table 1: Selected groupings of identical `webgl` renders. Each group corresponds to a single pixmap.

Group	#	Graphics Cards	Browsers	OS
1	3	ATI Mobility Radeon HD 4250 (Device 9712)	Chrome 17	Windows 7
	1	ATI Radeon HD 2400 Pro (Device 94c1)	Chrome 17	Windows 7
	1	ATI Radeon HD 2600 Pro (Device 9589)	Chrome 17	Windows 7
	3	ATI Radeon HD 3200 Graphics (Device 9612)	Chrome 17	Windows 7
	1	ATI Radeon HD 3800 Series (Device 9505)	Chrome 19	Windows 7
	2	ATI Radeon HD 4200 (Device 9710)	Chrome 17	Windows 7
5	1	ATI Mobility Radeon HD 4300 Series (Device 9552)	Chrome 17	Windows 7
	1	ATI Mobility Radeon HD 4330 (Device 9552)	Chrome 17	Windows 7
	1	ATI Mobility Radeon HD 4530 (Device 9553)	Chrome 17	Windows 7
	1	ATI Mobility Radeon HD 4670 (Device 9488)	Chrome 17	Windows 7
	1	ATI Mobility Radeon HD 545v (Device 9553)	Chrome 17	Windows 7
	1	ATI Mobility Radeon HD 550v (Device 9480)	Chrome 17	Windows 7
	1	ATI Radeon HD 4350 (Device 954f)	Chrome 17	Windows 7
	1	ATI Radeon HD 4550 (Device 9540)	Chrome 17	Windows 7
20	7	Intel(R) 82945G Express Chipset Family (Device 2772)	Chrome 17	Windows 7
	1	Intel(R) 82945G Express Chipset Family (Device 2772)	Chrome 17	Windows Vista
	1	Intel(R) 82945G Express Chipset Family (Device 2772)	Chrome 18	Windows 7
	1	Intel(R) 82945G Express Chipset Family (Device 2772)	Firefox 10	Windows XP
	1	Intel(R) 82945G Express Chipset Family (Device 2772)	Firefox 8	Windows 7
	1	Intel(R) HD Graphics Family (Device 0102)	Firefox 10	Windows 7
	2	Mobile Intel(R) 945 Express Chipset Family (Device 27a2)	Chrome 17	Windows 7
23	1	Intel(R) G33/G31 Express Chipset Family (Device 29c2)	Firefox 10	Windows 7
24	1	Intel(R) G41 Express Chipset (Device 2e32)	Chrome 15	Windows 7
	7	Intel(R) G41 Express Chipset (Device 2e32)	Chrome 17	Windows 7
	2	Intel(R) G41 Express Chipset (Device 2e32)	Chrome 19	Windows 7
	1	Intel(R) Graphics Media Accelerator HD (Device 0046)	Chrome 17	Windows 7
	1	Intel(R) HD Graphics (Device 0042)	Chrome 15	Windows 7
	1	Intel(R) HD Graphics (Device 0042)	Chrome 16	Windows 7
	2	Intel(R) HD Graphics (Device 0042)	Chrome 17	Windows 7
	1	Intel(R) HD Graphics (Device 0042)	Firefox 10	Windows 7
	2	Intel(R) HD Graphics (Device 0046)	Chrome 16	Windows 7
	23	Intel(R) HD Graphics (Device 0046)	Chrome 17	Windows 7
	1	Intel(R) HD Graphics (Device 0046)	Chrome 17	Windows 8
	5	Intel(R) HD Graphics (Device 0046)	Firefox 10	Windows 7
	1	Intel(R) HD Graphics (Device 0046)	Firefox 10	Windows XP
	1	Intel(R) HD Graphics (Device 0046)	Firefox 11	Windows 7
	2	Intel(R) HD Graphics (Device 0046)	Firefox 9	Windows 7
	3	Intel(R) HD Graphics (Device 0df1)	Chrome 17	Windows 7
	1	Intel(R) HD Graphics (Device 0df1)	Chrome 19	Windows 7
	2	Mobile Intel(R) 4 Series Express Chipset Family (Device 2a42)	Chrome 16	Windows 7
	9	Mobile Intel(R) 4 Series Express Chipset Family (Device 2a42)	Chrome 17	Windows 7
	1	Mobile Intel(R) 4 Series Express Chipset Family (Device 2a42)	Chrome 17	Windows Vista
	1	Mobile Intel(R) 4 Series Express Chipset Family (Device 2a42)	Firefox 10	Windows 7
	1	Mobile Intel(R) 4 Series Express Chipset Family (Device 2a42)	Firefox 10	Windows XP
	7	Mobile Intel(R) 45 Express Chipset Family (Device 2a42)	Chrome 17	Windows 7
	1	Mobile Intel(R) 45 Express Chipset Family (Device 2a42)	Chrome 18	Windows 7
	1	UNKNOWN	Chrome 17	Windows 7
	1	UNKNOWN	Chrome 17	Windows Vista
	1	UNKNOWN	Chrome 18	Windows 7
25	1	Intel(R) HD Graphics Family (Device 0102)	Firefox 10	Windows XP
	1	Intel(R) HD Graphics Family (Device 0116)	Firefox 8	Windows 7
	3	Intel(R) HD Graphics Family (Device 0116)	Firefox 9	Windows 7
36	1	NVIDIA GeForce 6150SE nForce 430 (Device 03d0)	Chrome 17	Windows 7
	1	NVIDIA GeForce 6200 (Device 0221)	Firefox 10	Windows XP
	1	NVIDIA GeForce 7100 / NVIDIA nForce 630i (Device 07e1)	Firefox 10	Windows 7
	1	NVIDIA GeForce 7150M / nForce 630M (Device 0531)	Chrome 17	Windows 7
	1	NVIDIA GeForce 7300 SE/7200 GS (Device 01d3)	Chrome 17	Windows 7
	1	UNKNOWN	Chrome 15	Windows XP
42	1	NVIDIA GeForce 8600	Chrome 17	Linux
	1	NVIDIA GeForce 9400	Chrome 17	OSX 10.7.3
	1	NVIDIA GeForce 9800	Chrome 17	Linux
	1	NVIDIA GeForce 320M (Device 08a0)	Firefox 10	OSX 10.6



(a) Original (Intel G41) (b) Group 1 (Radeon HD 2400) (c) Group 20 (Intel 82945G) (d) Group 23 (Intel G33/G31) (e) Group 25 (Intel HD Graphics) (f) Group 36 (GeForce 6200)

Figure 10: Original render and difference maps for Group 24

vented: simply repeat the test a few times and compare the results (perhaps by averaging or selecting the most common pixel color). While increasing the noise combats this, it also degrades the performance of `<canvas>` significantly for legitimate applications. Applying the same noise on multiple runs would only aid in fingerprinting. Therefore, we conclude that adding noise is not a feasible defense against our fingerprinting methods.

Thinking further, we note that our fingerprints measure functional differences in both the hardware and software running on the device. A sure-fire way to defend against leaking this information, then, is for every system to produce identical results. To do so, browser vendors will need to agree on a list of "`<canvas>`-safe" fonts, and then ship these fonts, along with text rendering libraries such as Pango, as a supplement to the browser. To support WebGL, browsers would ignore the graphics card entirely and render scenes in a generic software renderer such as Mesa 3D. While this approach might be acceptable where privacy is of the utmost importance, the performance impact would be unacceptable in a shipping browser. Note that performing this emulation is a signal in and of itself, revealing that the user is taking precautions to be anonymous.

The easiest effective defense, then, is to simply require user approval whenever a script requests pixel data. Modern browsers already implement this type of security — for example, user approval is required for the HTML5 geolocation APIs. This approach continues the existing functionality of `<canvas>` while disallowing illegitimate uses, at the cost of yet another user-facing permissions dialog.

More complicated defense schemes can certainly be imagined. Imagine, perhaps, a `<canvas>` implementation that uses hardware acceleration to produce the pixels displayed to the user, but regenerates the entire image with an emulated implementation whenever the site requests pixel data. Such complicated schemes might sucessfully defend against a `<canvas>` fingerprint, but add significant complexity to HTML5 APIs and behavior.

6. CONCLUSIONS

We have demonstrated that the behavior of `<canvas>` text and WebGL scene rendering on modern browsers forms a new system fingerprint. The new fingerprint is consistent, high-entropy, orthogonal to other fingerprints, transparent to the user, and readily obtainable.

We believe that such fingerprints are inherent when the browser is — for performance and consistency — tied closely to operating system functionality and system hardware.

We do not yet have the data necessary to estimate the entropy of our fingerprint over the entire population of the web, but given our preliminary findings, 10 bits is a (possibly very) conservative estimate. Indeed, Benoit Jacob has estimated the entropy of just the GPU model at 9 bits [9].

We were surprised at the amount of variability we observed in even very simple tests, such as rendering a sentence in 12-point Arial. We conjecture that it is possible to distinguish even systems for which we obtained identical fingerprints, by rendering complicated scenes that come closer to stressing the underlying hardware. Note that an attacker could refine a fingerprint in a black-box way by having some victims render experimental scenes in addition to the ones used for fingerprinting. Those scenes that allow otherwise identical systems to be distinguished should be added to the fingerprint.

We are pessimistic about the possibility of eliminating the fingerprints we identified without seriously degrading browser functionality and performance, or require yet more user approval dialogs to enable basic functionality. Perhaps the time has come to acknowledge that fingerprints are unavoidable on the modern web.

For browsers specifically designed to limit the ability of attackers to fingerprint users, such as Tor's modified Firefox with Torbutton [15], more drastic steps may be necessary. Torbutton already disables WebGL, and it likely should disable GPU-based compositing and 2D canvas acceleration.[11] But even this step does not address the fingerprint obtained through font rendering. It may be necessary for Tor's Firefox to eschew the system font-rendering stack, and instead implement its own — based, for example, on GTK+'s Pango library.

Acknowledgments

We are grateful to Úlfar Erlingsson, Eric Rescorla, Stefan Savage, and Geoff Voelker for helpful discussions about this work. This material is based upon work supported by the National Science Foundation under Grants No. CNS-0831532 and CNS-0964702, and by the MURI program under AFOSR Grant No. FA9550-08-1-0352.

[11] WebGL could perhaps be implemented in software. As an example, Chrome 17 falls back on a software GL backend on systems without a supported GPU.

7. REFERENCES

[1] G. Aggarwal, E. Bursztein, C. Jackson, and D. Boneh. An analysis of private browsing modes in modern browsers. In I. Goldberg, editor, *Proceedings of USENIX Security 2010*, pages 79–93. USENIX, Aug. 2010.

[2] S. Baker. Re: [public WebGL] about the VENDOR, RENDERER, and VERSION strings. Public WebGL mailing list, Nov. 2010. `https://www.khronos.org/webgl/public-mailing-list/archives/1011/msg00221.html`.

[3] K. Boda, Á. M. Földes, G. G. Gulyás, and S. Imre. User tracking on the Web via cross-browser fingerprinting. In P. Laud, editor, *Proceedings of NordSec 2011*, LNCS. Springer-Verlag, Oct. 2011.

[4] Cve-2010-3855. Online: `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-3855`, Dec. 2011.

[5] R. Dingledine and N. Mathewson. Tor: The second-generation onion router. In M. Blaze, editor, *Proceedings of USENIX Security 2004*, pages 303–19. USENIX, Aug. 2004.

[6] P. Eckersley. How unique is your Web browser? In M. Atallah and N. Hopper, editors, *Proceedings of PETS 2010*, volume 6205 of *LNCS*, pages 1–18. Springer-Verlag, July 2010.

[7] G. Fleischer. Attacking Tor at the application layer. Presentation at DEFCON 2009, Aug. 2009. Online: `http://pseudo-flaw.net/content/defcon/`.

[8] J. Forshaw. Webgl - a new dimension for browser exploitation. Online: `http://www.contextis.com/resources/blog/webgl/`, May 2011.

[9] B. Jacob. Re: [public WebGL] about the VENDOR, RENDERER, and VERSION strings. Public WebGL mailing list, Nov. 2010. `https://www.khronos.org/webgl/public-mailing-list/archives/1011/msg00229.html`.

[10] B. Jacob. Re: [public WebGL] information leakage and the extension regisrty. Public WebGL mailing list, Dec. 2010. `http://www.khronos.org/webgl/public-mailing-list/archives/1012/msg00083.html`.

[11] A. Juels, M. Jakobsson, and T. N. Jagatic. Cache cookies for browser authentication (extended abstract). In V. Paxson and B. Pfitzmann, editors, *Proceedings of IEEE Security and Privacy ("Oakland") 2006*, pages 301–05. IEEE Computer Society, May 2006.

[12] J. R. Mayer. "Any person... a pamphleteer": Internet anonymity in the age of Web 2.0. Senior thesis, Princeton University, Apr. 2009.

[13] Microsoft security bulletin ms11-087. Online: `http://technet.microsoft.com/en-us/security/bulletin/ms11-087`, Dec. 2011.

[14] K. Mowery, D. Bogenreif, S. Yilek, and H. Shacham. Fingerprinting information in JavaScript implementations. In H. Wang, editor, *Proceedings of W2SP 2011*. IEEE Computer Society, May 2011.

[15] M. Perry. Torbutton design documentation, June 2010. Online: `http://www.torproject.org/torbutton/en/design/index.html.en`.

[16] P. Reschl, M. Mulazzani, M. Huber, and E. Weippl. Poster abstract: Efficient browser identification with JavaScript engine fingerprinting. In *Poster Session of ACSAC 2011*, Dec. 2011. Online: `http://www.acsac.org/2011/program/posters/Reschl.pdf`.

APPENDIX

A. DATA CHARACTERIZATION

Table 2: Groups of identical `webgl` renders. Number in parentheses indicate device IDs, where appropriate.

#	Size	Graphics Cards	Browsers	OS
1	11	ATI Mobility Radeon HD 4250; ATI Radeon HD 2400 Pro, 2600 Pro, 3200, 3800, 4200	Chrome (17, 19)	Windows 7
2	8	AMD Radeon HD 6250, 6300, 6310, 6670, 6800; ATI Radeon HD 5450, 5670	Chrome (17, 18)	Windows 7
3	4	AMD Radeon HD 6250; Mobility Radeon HD 5650; ATI Radeon HD 5670; Unknown	Firefox (9)	Windows 7
4	1	ATI HD 5850	Opera (9.8)	Windows 7
5	8	ATI Mobility Radeon HD 4300, 4330, 4530, 4670, 545v, 550v; ATI Radeon HD 4350, 4500	Chrome (17)	Windows 7
6	9	ATI Mobility Radeon HD 5470, 5730, 5700 Series; Intel HD Graphics (0116)	Chrome (17)	Windows 7
7	1	ATI Radeon 2100	Chrome (17)	Windows 7
8	1	ATI Radeon HD 2600 Pro	Chrome (17)	OS X 10.6.8
9	1	ATI Radeon HD 4300/4500 Series (954f)	Firefox (8)	Windows 7
10	4	ATI Radeon HD 5700 Series (68be), 5800 Series (6899); Unknown	Firefox (10, 11)	Windows 7
11	1	ATI Radeon HD 6750M	Firefox (10)	OS X 10.7
12	1	Unknown	Android 2.3.4	Android
13	1	Intel GMA 950	Chrome (18)	OS X 10.7.3
14	1	Intel GMA X3100	Chrome (17)	OS X 10.5.8
15	1	Intel GMA 950	Firefox (10)	OS X 10.6
16	1	Intel HD Graphics 3000	Firefox (10)	OS X 10.7
17	1	Intel HD Graphics 3000	Firefox (12)	OS X 10.7
18	3	Intel 946GZ Express Chipset (2972), Mobile Intel 956 Express Chipset (2a02)	Chrome (10, 17); Firefox (10)	Windows (7, XP)
19	5	Intel 82845G (2562), G41 Express Chipset (2e32), 945GM/GU Express Chipset (27a2); Unknown	Chrome (18)	Windows XP

Table 2: Groups of identical `webgl` renders (cont.)

#	Size	Graphics Cards	Browsers	OS
20	14	Intel 82945G Express Chipset (2772); Intel HD Graphics (0102); Mobile Intel 945 Express Chipset (27a2)	Chrome (17, 18); Firefox (8, 10)	Windows (7, Vista, XP)
21	2	Intel 82945G Express Chipset (2772)	Firefox (8, 10)	Windows (XP)
22	24	Intel G33/G31 Express Chipset (29c2); Intel Graphics Media Accelerator 3150 (a011); Unknown	Chrome (13, 17, 18); Firefox (8, 9, 10, 11)	Windows 7
23	1	Intel G33/G31 Express Chipset (29c2)	Firefox (10)	Windows 7
24	80	Intel G41 Express Chipset (2e32); Intel Graphics Media Accelerator HD (0046); Intel HD Graphics (0042, 0046, 0df1); Mobile Intel 4 Series (2a42); Mobile Intel 45 Series (0x2a42); Unknown	Chrome (15-19); Firefox (9, 10, 11)	Windows (7, 8, Vista, XP)
25	5	Intel HD Graphics (0102, 0116)	Firefox (8, 9, 10)	Windows (7, XP)
26	13	Intel HD Graphics (0102, 0116); Unknown	Chrome (16, 17); Firefox (10, 11)	Windows 7
27	1	Mesa DRI Intel Ironlake Desktop	Chrome (17)	Linux
28	10	Mobile Intel 965 Express Chipset (2a02, 2a12); Intel GMA X3100 (2a02)	Chrome (14, 17, 18); Firefox (8, 9, 10)	Windows (7, XP)
29	1	NVIDIA GeForce 7025	Firefox (10)	Linux
30	1	NVIDIA GeForce 9400M	Firefox (10)	OS X 10.6
31	19	NVIDIA GeForce 210, 315M, 8400GS, 8500 GT, 8600M GT, 9200M GS, 9500 GT, 9600 GT, 9600M GT, G 105M, G 210M, GTS 250, GTS 360M, GTX 295	Chrome (13, 17); Firefox (10)	Windows (7, Vista)
32	2	NVIDIA GeForce 315M; Unknown	Firefox (5, 6)	Windows 7
33	1	NVIDIA GeForce 320M	Safari (5)	OS X 10.7.3
34	6	NVIDIA GeForce 410M, GT 425M, GTX 460, GTX 460 SE	Chrome (17); Firefox (10)	Windows (7, XP)
35	1	NVIDIA GeForce GTX 550	Chrome (17)	Linux
36	6	NVIDIA GeForce 6150SE, 6200, 7100, 7150M, 7300 SE/7200GS; Unknown	Chrome (15, 17); Firefox (10)	Windows (7, XP)
37	1	NVIDIA GeForce 6200	Firefox (4)	Windows 7
38	1	NVIDIA GeForce 6800 Series (00f9)	Firefox (10)	Windows 7
39	1	NVIDIA GeForce 7025	Chrome (18)	Windows XP
40	1	NVIDIA GeForce 7300SE/7200GS	Firefox (10)	Windows 7
41	1	NVIDIA GeForce 7650 GS	Chrome (17)	Windows 7
42	4	NVIDIA GeForce 8600, 9800, 320M, 9400	Chrome (17); Firefox (10)	Linux; OS X (10.6, 10.7.3)
43	4	NVIDIA GeForce 8400 GS, 9500 GT, Quadro FX 1800M; Unknown	Chrome (15, 17, 19); Firefox (13)	Windows 7
44	1	NVIDIA GeForce GT 220	Firefox (10)	Windows 7
45	1	NVIDIA GeForce GT 330M	Chrome (19)	OS X 10.7.3
46	1	NVIDIA GeForce GT 440	Opera (9.80)	Windows 7
47	1	Intel 965GM	Firefox (8)	Linux
48	1	Intel 965GM	Firefox (10)	Linux
49	1	Mesa DRI Intel Ironlake Mobile	Firefox (10)	Linux
50	1	Mesa DRI Intel Sandybridge Mobile	Firefox (10)	Linux
No WebGL	30	AMD Radeon HD 6670, 7450M; ATI Radeon HD 3200, X1200, X1900, XPRESS 200; Intel GMA 3100; Intel 82865G; Intel G41; Intel HD Graphics; NVIDIA GeForce 315M, 6150SE, 6200, FX 5500; NVIDIA Quadro 6000; VIA Integrated; Unknown	Chrome (15, 16, 17); Opera (9.80); Safari (5)	Linux; OSX (10.6.8); Windows (7, Vista, XP)