PriVaricator: Deceiving Fingerprinters with Little White Lies

Nick Nikiforakis, Wouter Joosen
KU Leuven

Benjamin Livshits
Microsoft Research

Abstract

This paper proposes a solution to the problem of browser-based fingerprinting. An important observation is that making fingerprints non-deterministic also makes them hard to link across subsequent web site visits. Our key insight is that when it comes to web tracking, the real problem with fingerprinting is not *uniqueness* of a fingerprint, it is *linkability*, i.e. the ability to connect the same fingerprint across multiple visits. In PriVaricator we use the power of randomization to "break" linkability by exploring a space of parameterized randomization policies. We evaluate our techniques in terms of being able to prevent fingerprinting and also in terms of not breaking existing (benign) sites. The best of our randomization policies renders *all* the fingerprinters we tested ineffective, while causing minimal damage on a set of 1,000 Alexa sites on which we tested, with no noticeable performance overhead.

1 Introduction

Browser-based fingerprinting, proposed as a theoretical threat to online privacy several years ago, has by now emerged as a full-fledged alternative to traditional cookie-based tracking. Recent work has demonstrated the growing proliferation of JavaScript-based fingerprinting on the web [1, 16]. Today, companies such as BlueCava [3], ThreatMetrix [19] and iovation [11] routinely fingerprint millions of web users. However, despite several attempts, mostly involving privacy-enhancing browser extensions, there has been a dearth of comprehensive privacy-enhancing technologies addressing in-browser fingerprinting. In this paper, we propose a comprehensive approach to prevent reliable fingerprinting in the browser, called PriVaricator.

Key insight: Much has been made of the fact that it is possible to derive a unique fingerprint of a user, primarily via JavaScript as shown by the Panopticlick project [8]. However, the insight behind our techniques is the realization that the culprit behind fingerprinting is not the fact that a user's fingerprint is unique, but that it is *linkable*, *i.e.* it can be reliably associated with the same user over multiple visits. While popular prevention techniques have attempted to make the fingerprints of large groups of users look *the same* [20], the key insight our paper explores involves doing the *opposite*. PriVaricator modifies the browser to make every visit appear different to a fingerprinting site, resulting in a *different* fingerprint that cannot be easily linked to a fingerprint from another visit, thus frustrating tracking attempts.

Randomization policies: In this paper we explore a space of *randomization policies* designed to produce unique fingerprints. The basis of our approach is to change the way the browser represents certain key properties, such as `offsetHeight` (used to measure the presence of fonts) and plugins, to the JavaScript environment. We observe that creatively misrepresenting — or lying — about these values introduces an element of non-determinism, which generally makes fingerprints unlinkable over visits.

Note that the randomization is not as easy as it might sound: as discussed in [16], producing practically impossible combinations of, say, browser headers and the `navigator` object, can actually *reduce* user privacy. Intuitively, blatant lying is *not* such a good idea; however, misrepresenting key properties of the browser environment in a subtle way goes a long way toward combating fingerprinters. In summary, a randomization policy should 1) produce unlinkable fingerprints and 2) not break existing sites.

Implementation: We have implemented PriVaricator on top of the Chromium web browser. We proceed to systematically evaluate the space of randomization policies in an effort to 1) maximize the effectiveness, as measured by fingerprinting sites rendered ineffective and 2) minimize the number of broken benign sites, as measured by the Alexa list of popular sites.

Deployment: We position PriVaricator as an enhancement to the private browsing mode already present in the majority of browsers. Existing private modes help prevent stateful tracking via cookies; PriVaricator focuses on preventing stateless tracking. We believe that it is better to integrate PriVaricator into the browser itself as opposed to providing it via an extension. One of the reasons for this, is the fact that most privacy extensions so far have only enjoyed a small deployment base, which in fact often makes it *easier* for the fingerprinter to identify the user [16].

Evaluation: We discovered that a number of our policies are able to render the fingerprinters we tested ineffective, while creating minimal damage to benign sites. In particular, the best of our policies renders *all* the fingerprinters we tested on ineffective, while only altering the visual appearance of, on average, 0.7% of the content offered by the top 1,000 Alexa sites. Using three JavaScript benchmark suites, we show that the modifications needed to implement PriVaricator on top of the Chromium browser cause a negligible performance overhead.

2 Background

A device *fingerprint* is a set of system attributes that are usually combined in the form of a string. This combination of attributes is generally designed to be unique with a high likelihood and, as such, can function as a device identifier. Attributes that range over a broader set of values (*e.g.*, the list of fonts and plugins) are more identifying than values shared by many devices (*e.g.*, version of the operating system). Stability is a desirable property in a fingerprinting strategy; choosing attributes with values that are more stable over time (*i.e.*, that change only infrequently or very gradually) facilitate reliable identification, compared to those that change frequently and unpredictably.

Web-based device fingerprinting is the process of collecting sufficient information through the browser to perform stateless device identification. The collected information is generally obtained via JavaScript and includes the device's screen size, the versions of installed browser plugins, and the list of installed fonts. Collected fingerprints may be used as user identifiers for web tracking, *i.e.* linking of visits to (one or multiple) web pages as made by the same user or device. An example of a simple fingerprinting strategy is provided by the fingerprintjs library [21]:

```
var hasher = function(value, seed){ return value.
    length % seed; }
var fingerprint = new Fingerprint(hasher).get();
```

The last line computes the fingerprint for the user, which in this case is a long integer. Depending on the specific choices of each fingerprinter, a user's device fingerprint can be an integer, a hash of the user's fingerprintable attributes, or a server-side/client-side generated GUID.

2.1 Why Fingerprint?

When it comes to the motivation behind web-based device fingerprinting, two reasons have emerged as most common.

Third-party tracking: Probably the most common use of fingerprinting involves tracking the user across multiple, possibly unrelated, web sites to construct an interest profile for the user; this profile can then be employed to deliver targeted ads. As has been argued before, fingerprinting is an effective and stealthy alternative to stateful cookie-based user tracking. In a sense, the better fingerprinting works, the more information is learned about the user with a higher degree of reliability, leading to better ad targeting and thus to higher conversion rates for the advertisers. This creates a direct incentive for ad delivery networks to invest in better fingerprinting strategies, especially given that fingerprinters might not necessarily obey browser-provided Do-Not-Track (DNT) headers [1].

Fraud prevention: Third-party tracking, an activity that has provoked much outrage on the part of both privacy advocates and some users, is not the only raison d'être behind fingerprinting. It is sometimes argued that fingerprints can be used for fraud prevention. We refer the interested reader to some of the literature from the fingerprinting companies themselves [11, 18, 19] for further details. We should note that it is not obvious that collected fingerprints cannot be also sold to third parties or abused for tracking purposes by the companies that collect them.

In connection to fraud prevention, advocates of fingerprinting claim that a device fingerprint is a powerful tool for finding related transactions either as an identifier in itself or as a means of finding transactions with related characteristics. Fingerprints also can be used to find out when account information is being shared illegally. The gathered fingerprints can be augmented with device reputation information and be used to blacklist fraudulent users and their activities.

Opt-out: While some of the aforementioned fingerprinting companies offer opt-out pages for the user, it is highly non-obvious what a successful opt-out really means. Ironically, to know that a user has opted-out of tracking, the fingerprinters still first need to compute the fingerprint (assuming cookies are disabled) and then, if they are honest, proceed to disregard information from that session.

Compared to stopping stateful tracking that can be achieved at the client-side, via disabling or clearing cookies, and the help of many extensions, this server-side approach to opting out is not satisfying because, ultimately, the user needs to trust the fingerprinting server. With PriVaricator, *reliable* fingerprinting is rendered impossible in the first place.

2.2 BlueCava Explained

Figure 1 shows a partial code snippet from BlueCava's fingerprinting code, which we simplified to improve readability. Function `getPlugins` (lines 2–14) is responsible for gathering the list of plugins and their descriptions, as those are reported by a user's browser. The `getFontMeasurement` function (lines 16–32) is a helper function that is called by the `getFonts` function (lines 34–50). These functions together, allow fingerprinters to extract a list of fonts installed on a user's machine without the browser explicitly providing such information to scripts.

As explained by Nikiforakis *et al.* [16], code of this nature takes advantage of the fact that different font families have stylistic differences which affect the width and height of a block of text rendered with them. By testing these text dimensions against the fallback font of browsers (`ground_truth` obtained in line 39), the script can go through a long list of fonts (line 36) and mark the ones that deviate from that ground truth, as present (if a font family is not installed, the browser will use the fallback font and thus the condition of the `if` statement on lines 44–45 will not be satisfied).

Finally, this script combines this information together with other fingerprintable attributes of a user's browser into a fingerprint string (line 59). Depending on each specific case, the fingerprint will

```
 1  /* Extracting plugin information */
 2  function getPlugins(){
 3      plugs = []
 4      for (var d = 0; d < navigator.plugins.length; d++) {
 5          var e = navigator.plugins[d];
 6          [...]
 7          plugs.push({
 8              Name: e.name,
 9              Description: e.description,
10              FileName: e.filename
11          });
12      }
13      return plugs;
14  }
15
16  /* JS-based detection of fonts */
17  function getFontMeasurement(font_family){
18      var h = document.getElementsByTagName("BODY")[0];
19      var d = document.createElement("DIV");
20      var s = document.createElement("SPAN");
21      d.appendChild(s);
22      d.style.fontFamily = font_family;
23      s.style.fontFamily = font_family;
24      s.style.fontSize = "72px";
25      s.innerHTML = "mmmmmmmmmmlil";
26      h.appendChild(this.d);
27      textWidth = s.offsetWidth;
28      textHeight = s.offsetHeight;
29      h.removeChild(d);
30
31      return [textWidth, textHeight];
32  }
33
34  function getFonts(){
35      /* Long list of fonts */
36      var fonts = ["cursive","monospace","serif","sans-serif","fantasy","Arial","Arial Black","Arial Narrow",...]
37
38      /* Measuring ground truth */
39      var ground_truth = getFontMeasurement("sans");
40      var discovered_fonts = []
41
42      for(var i=0; i < fonts.length; i++){
43          c_measurement = getFontMeasurement(fonts[i]);
44          if (c_measurement[0] != ground_truth[0] ||
45              c_measurement[1] != ground_truth[1]){
46              discovered_fonts.push(fonts[i]);
47          }
48      }
49      return discovered_fonts;
50  }
51  /* Get more fingerprintable information from a user's:
52    - timezone
53    - screen dimensions
54    - math constants
55    - ...
56  */
57  [...]
58
59  var fingerprint = combineIntoFingerprint(getPlugins(),getFonts(),...);
60  sendFingerprint(fingerprint);
61  [...]
```

Figure 1: Partial fingerprinting code from BlueCava.

Fingerprinting	**Script**	**Plugin**	**Screen**	**Uses**	**Access to**	
provider	**name**	**enumeration**	**properties**	**canvas**	`offsetWidth`	`offsetHeight`
Bluecava	`BCAC5.js`	✓	✓	✗	63	63
Perferencement	`tagv22.pkmin.js`	✓	✓	✗	155	155
CoinBase	`application-9a3a[...].js`	✓	✓	✗	592	197
MaxMind	`device.js`	✓	✓	✗	261	27
Inside graphs	`ig.js`	✓	✓	✗	1,050	48

Figure 2: Techniques used by various fingerprinters in the wild.

potentially be encrypted and then sent to the fingerprinter via a range of techniques: through a JavaScript XHR request, using an image beacon, or through HTTP cookie headers.

3 Overview

At the heart of PriVaricator is a strategy for misrepresenting the way parts of the browser environment are presented to the JavaScript language runtime. Previous studies of fingerprinters in the wild [1] have identified certain parts of the browser environment reflected into JavaScript as key to producing a reliable fingerprint. These properties range from commonplace ones such as `navigator.userAgent` to ones that are significantly more obscure such as `getBoundingClientRect` which may be used instead of the `offsetHeight` and `offsetWidth` attributes of DOM elements, to test for the presence of particular fonts on a user's machine.

Of course, our wish is to misrepresent environment properties of features that would be most damaging to fingerprinters without breaking existing code. As such, lying about `navigator.userAgent` is generally not a good idea: this may very well cause the server to send HTML designed for a different browser. However, subtly changing the results of offset measurements turns out to be a better option.

Figure 2 lists some of the representative fingerprinters found in the wild, showing which fingerprinting features they use. Note that canvas-based fingerprinting described in [14] does not appear to be widely used in practice so far. Based on this information, combined with statistics about which features provide the highest number of bits of identifying information, in this paper, we primarily focus on randomizing 1) plugins and 2) fonts. Both of these provide 21.7 bits of identifying information, according to Panopticlick [8].

3.1 Explicit Fingerprinting

Note that in PriVaricator we do not claim to solve the *entire* problem of web-based device fingerprinting; indeed, the focus of PriVaricator is on explicit attempts to fingerprint users via capturing the details of the browser environment. We are *not* attempting to provide protection against sophisticated side channels such as browser performance [13] which may be used as part of fingerprinting. Our focus is on *explicit fingerprinting*, *i.e.* JavaScript-based fingerprinting which operates by computing a function of environment variables exposed within the browser.

```
Expressions
  exp            ::=  numExp | stringExpr | boolExp
                      | exp.exp(exp[,...]) | exp(exp[,...])
                      | boolCond?exp:exp | exp binaryOp exp
                      | exp.exp | exp[exp]
Booleans
  boolCond       ::=  boolExp | ¬boolExp
                      | exp ∨ exp
                      | exp ∧ exp
  boolExp        ::=  true | false
                      | exp boolOp exp
Numerics
  numExp         ::=  numericFunc(exp) | 1 | 2 | ...
                      | numExp binaryOp numExp
                      | − numExp
                      | parseInt(stringExpr)
                      | indexOf(stringExpr, stringExpr)
                      | abs(numExp)
                      | min(numExp, numExp)
                      | numEnvProp
String expressions
  stringExpr     ::=  stringFunc(exp) | "" | ...
                      | encodeURI(stringExpr)
                      | decodeURI(stringExpr)
                      | substr(stringExpr)
                      | concat(stringExpr, stringExpr)
                      | replace(stringExpr, stringExpr)
                      | replace(/stringExpr/, stringExpr)
                      | toString(numExp)
                      | toString(numExp, numExp)
                      | stringEnvProp
Operators
  binaryOp       ::=  + | − | ∗ | / | % | << | >>
  boolOp         ::=  = | ≠ | < | <= | > | >=

Fingerprinting properties
  envProp        ::=  stringEnvProp | numEnvProp |
  stringEnvProp  ::=  navigator.userAgent
                      | navigator.appName | ...
                      | exp.navigator.plugins
  numEnvProp     ::=  exp.offsetWidth | exp.offsetHeight | ...
```

Figure 3: BNF for explicit fingerprinting programs supported by PriVaricator. The start symbol is exp. An effective fingerprinting function will have at least one reference to envProp symbols, usually more.

Furthermore, we are not attempting to address fingerprinting implemented using plugins such as Flash, although a technique similar to PriVaricator could be developed to hinder fingerprinting in ActionScript programs built into Flash.

Figure 3 shows a BNF for fingerprinting programs over environment properties captured as envProp. Our examination of popular fingerprinters suggests that they compute a fingerprint, as captured by the BNF, and then proceed to communicate it to the server, using any of the techniques that we discussed in the previous section.

3.2 Randomization Policies

Our strategy in PriVaricator is to intercept each of the accesses to DOM properties of interest and augment the values returned to the JavaScript environment using a set of *randomization policies*. A wide range of randomization policies may apply in principle; for example, for integer values of properties such as `offsetWidth`, a slight change to the returned value is enough. For a property that is more structural and complex, such as the `toDataURL` function used in canvas-based fingerprinting that returns the current state of the canvas as an image [14], a different randomization policy may be used; an example policy for images may add slight visual *noise* to the returned image.

Policies for offset measurements: For the values of `offsetHeight`, `offsetWidth`, and `getBoundingClientRect` in PriVaricator, we propose the following randomization policies: a) Zero; b) Random(0..100); and c) ± 5% Noise. When these policies are active, instead of returning the original offset value, they return zero, a random number between 0 and 100, and the original number ±5% noise, respectively. What these policies have in common is that they perform arithmetic operations on numbers with deterministic and non-deterministic results. While one can probably envision many more randomization policies, we focused on those that generate plausible offset values (e.g. no generation of negative numbers) as well as those that will create enough noise to confuse fingerprinting efforts. For instance, for the third policy, if the percentage of noise added to an offset is too little, then, for small offset values, it may be rounded off to the same original integer value and become ineffective.

These policies are controlled by a *lying threshold* (denoted as θ) and a *lying probability* (denoted as $P(lie)$). θ controls how fast PriVaricator starts lying, *i.e.*, after how many accesses to `offsetWidth` or `offsetHeight` values, will the policy kick in. $P(lie)$ specifies the probability of lying, *after* the θ threshold has been surpassed.

Policies for plugins: For the randomization of plugins, we define a probability $P(plug\_hide)$ as the probability of hiding each individual entry in the plugin list of a browser, whenever the `navigator.plugins` list is populated.

Example: As an example, a configuration of

$$\begin{aligned} \mathsf{Rand\_Policy} &= \mathsf{Zero}, \\ \theta &= 50, \\ P(lie) &= 20\%, \\ P(plug\_hide) &= 30\% \end{aligned}$$

instructs PriVaricator to start lying after 50 offset accesses, to only lie in 20% of the cases, to respond with the value 0 when lying, and to hide approximately 30% of the browser's plugins. In Section 5 we investigate which combinations of values provide the best tradeoff between the production of unlinkable fingerprints and the breakage of benign websites.

3.3 Breakage Concerns

Building an effective fingerprinting prevention tool involves balancing the effectiveness of preventing fingerprinters and breaking real sites. To better understand the latter, we decided to crawl the top 10,000 Alexa sites to determine which ones use properties that are of interest to fingerprinters.

Access to property `offsetHeight` tends to be pretty telling. Overall, 82.3% of scripts have 0 accesses to `offsetHeight`. However, 1.87% of scripts have more than 50 accesses when visited at runtime. Figure 4 summarizes the result of our crawl, sorted by the number of runtime accesses to `offsetHeight`. Fortunately, the majority of sites seem to be ranked not very high. However, some of the sites listed, such as `spiegel.de`, are clearly important and we should take care not to break them in PriVaricator.

4 Implementation

In Section 3 we discussed the possible randomization policies that can be applied on the browser interfaces that are commonly abused for fingerprinting purposes. Since web-based device fingerprinting happens on the client side, the aforementioned policies could, in theory, be applied via an HTTP proxy, a browser extension, or built into the browser itself.

		Screen				*Navigator*									*HTML element*		
Alexa Rank	**Domain**	**colorDepth**	**height**	**pixelDepth**	**width**	**mimeType**	**platform**	**language**	**userAgent**	**appName**	**vendor**	**appCodeName**	**appVersion**	**plugins**	**getBounding-ClientRect**	**offsetWidth**	**offsetHeight**
6,444	`bunte.de`	0	1	0	1	0	0	2	8	0	0	0	0	1	2	205,115	202,909
8,039	`nzz.ch`	3	34	1	34	4	4	4	176	5	0	0	5	4	248	187881	187,349
191	`spiegel.de`	2	4	0	4	0	0	0	15	3	0	0	1	4	7	154,265	149,293
4,037	`wistia.com`	1	2	0	2	0	0	3	81	0	0	0	0	0	0	109,347	109,299
1,369	`zeit.de`	0	4	1	4	0	0	2	8	3	2	0	0	5	1,318	70,025	72,268
8,894	`menards.com`	0	0	0	0	0	0	0	3,783	0	0	0	0	0	37	43,847	38,715
4,754	`groupon.fr`	0	0	0	0	0	0	0	1	0	0	0	0	0	15	150,717	36,627
7,488	`xinmin.cn`	0	0	0	1	0	0	0	70,380	0	0	0	3	0	4,426	34,229	31,996
2,320	`celebuzz.com`	2	55	0	55	4	2	0	23	0	0	0	0	4	326	27,831	27,779
1,370	`wetter.com`	4	30	0	30	6	1	8	18	5	0	0	1	7	212	22,578	21,764

Figure 4: How widely are various fingerprintable browser properties used in the wild.

We ultimately chose to instrument the browser itself, although we examined other approaches at first.

Strawman approach: JavaScript-level interception: During preliminary experimentation, we attempted to detect accesses to fingerprintable properties by using getters, as defined in ECMAScript5, on the objects and attributes of choice, *e.g.*, `navigator.plugins`. At first glance, given the amount of obfuscation routinely found in JavaScript code, this seems a better strategy than attempting to instrument JavaScript code at the source level (e.g. via an HTTP proxy). The JavaScript code that defined these getters was injected in a page using a browser extension.

During that time, however, we encountered many browser-specific issues that eventually steered us towards modifying the browser itself. For instance, in order to be able to lie about the `offsetWidth` and `offsetHeight` of any given element, we need to intercept the requests of these attributes on all elements on a page, since we cannot *a priori* know which element(s) are going to be used for font detection. Unlike the `navigator` and `screen` objects which are created by the browser and thus always available, HTML elements are created initially when parsing a page's HTML code, as well as on-demand, whenever a programmer wishes to do so through JavaScript. As such, we need to intercept the creation of all HTML elements and define getters upon their creation.

The natural way to do this, is to "poison" the correct object prototype, so that all future JavaScript objects that inherit from that prototype will also inherit the getters. We discovered that although our prototype poisoning was working in Mozilla Firefox, it failed to work as expected in Google Chrome. By investigating the issue, we discovered that in Chrome, the `offsetWidth` and `offsetHeight` properties are not part of the `HTMLElement` prototype, but rather they are defined and initialized upon the creation of new elements. Interestingly, this is not the case for the the `getBoundingClientRect` method which also returns an element's `offsetWidth` and `offsetHeight`, and yet is defined in the expected prototype.

In addition to this browser-specific behavior, the use of getters also suffers from transparency issues. That is, a (malicious) script can check for the existence of getters using, among others, the `Object.getOwnPropertyDescriptor` method. Achieving transparency at the language level is fundamentally difficult [10].

Our implementation: For the reasons of better compatibility and transparency, we ultimately chose to implement our randomization policies within the browser, by changing the appropriate C++ code in the classes responsible for creating the `navigator` object, and the ones measuring the dimensions of elements. These changes are, by nature, very local; our full prototype involves modifications to a total of seven files in the WebKit implementation of the Chromium browser, version 34.0.1768.0 (242762). Figure 5 gives a taste for one of the representative changes we made. We believe these changes can be easily ported to other browsers.

The `antifpFakeMeasurement` function shown in Figure 5 is responsible for returning the appropriately modified offset value (based on the active randomization policy) as well as "resist" lying depending on the `LYING_PROBABILITY` value. The values of all capitalized variables are randomization policy parameters; once a suitable policy is chosen, these parameters can be conveyed to the browser via configuration files, the registry, or environment variables.

The second function shown in Figure 5 is Chromium's function for calculating the `offsetWidth` of a DOM element. As can be seen from the code, we have augmented the function with a counter that records the total number of offset accesses performed by scripts of a specific domain. If that counter exceeds our `LYING_THRESHOLD`, then instead of returning the calculated value, we return the result of our policy by applying `antifpFakeMeasurement` on the original offset value.

```
static int antifpFakeMeasurement(int o_measurement
    ) {
    // PriVaricator policies
    enum Policies {
        zero = 0, random_0_100, percent_noise,
            no_change
    };

    // Should we lie?
    if (rand() % 100 >= LYING_PROBABILITY){
        return o_measurement;
    }
    [...]
    switch (RANDOMIZATION_POLICY){
        case zero: return 0;
        case random_0_100: return rand() % 100;
        case percent_noise: {
            int sign;
            if (rand() % 100; < 50)
                sign = -1;
            else
                sign = 1;

            return o_measurement + (sign * (
                NOISE_FRACTION * o_measurement));
        }
        case no_change: return o_measurement;
        default: return o_measurement;
    }
}
// augmenting Chrome/WebKit code
int Element::offsetWidth() {
    [...]
    bool should_lie = false;
    int o_measurement;

    // Increase offset access counter and get
        current value (per domain)
    int offset_cnt = document().frame()->script().
        incAndGetOffsetAccessCounter();

    if (offset_cnt > LYING_THRESHOLD)
        should_lie = true;

    if (RenderBox* renderer = renderBox()) {
        if (renderer->
            canDetermineWidthWithoutLayout()){
            o_measurement =
                adjustLayoutUnitForAbsoluteZoom(
                renderer->fixedOffsetWidth(), *
                renderer).round();

            if (should_lie == false) {
                return o_measurement;
            } else {
                return antifpFakeMeasurement(
                    o_measurement);
            }
        }
    [...]
}
```

Figure 5: Randomization of `offsetWidth`.

Browser	JSBench		SunSpider		Kraken	
Chromium	69.70	±0.19	142.66	±0.57	1,161.44	±10.68
PriVaricator	69.57	±0.30	143.18	±0.76	1,147.78	±08.00

Figure 6: Performance comparison of "vanilla" Chromium and Chromium equipped with PriVaricator. All measurements are in *ms*.

5 Evaluation

The goal of PriVaricator's evaluation is three-fold. In addition to ensuring that the overhead of PriVaricator is minimal (Section 5.1), we want to maximize the effectiveness of fingerprinting prevention (Section 5.2), while minimizing the overall damage to the way users perceive the web (Section 5.3). When it comes to privacy-enhancing technologies, this tradeoff is not entirely new. For example, Mozilla Firefox decided to misreport (to JavaScript programs) the computed styles for links in order to prevent history leaks [2], after they had been demonstrated on a large scale.

5.1 Performance Overhead

In order to assess the performance overhead of PriVaricator, we used three independently-developed JavaScript benchmark suites: SunSpider version 1.0.2, Kraken version 1.1, and JSBench version 2013.1. Even though these benchmark suites already take repeated measurements, we also executed each suite five times, clearing the browser's cache in between runs. The experiments were run on a desktop machine, running a recent Ubuntu Linux distribution, with an Intel Core i5-3570 CPU @ 3.40 GHz processor, and 8 GB of RAM.

Figure 6 shows the average benchmark execution times (in *ms*) and standard deviations for an unmodified version of the Chromium browser, and for the same Chromium browser with our modifications present and enabled. Two out of the three benchmarks reported that our runs with PriVaricator executed, on average, slightly *faster* than the ones of the unmodified browser. Given the standard deviation of our measurements, our instrumentation is not, in reality, speeding up the browser; instead, these measurements show that the added overhead of PriVaricator is so negligible that it does not exceed the inherent noise in the reported execution time of browser benchmarks.

5.2 Preventing Fingerprinting

While one can fully analyze the client-side JavaScript code of fingerprinters, the way in which a user's fingerprintable attributes are combined and mapped to a fingerprint (also known as a *device identifier*) is not necessarily a client-side operation. In order to assess how our randomization policies affect a fingerprinter's ability of identifying us, we chose four services that can be used as black-box oracles. Some of these revealed the device identifier as part of the opt-out process, while with others more investigation was required. Unfortunately, finding fingerprinters that are willing to disclose information about their internal workings is a major challenge and it took us some time to understand how to test these four fingerprinters. For this evaluation, we measured how PriVaricator stands against BlueCava, Coinbase, PetPortal, and fingerprintjs, as explained below.

BlueCava: Similarly to other third-party trackers, BlueCava provides an opt-out page (`http://bluecava.com/opt-out`) for users who wish to opt-out of tracking by BlueCava. On this page, users are fingerprinted and their fingerprintable attributes are sent to BlueCava's server. The server then responds with a device identifier, *e.g.* `18B1-EBFC-A3F0-6D81-6DE8-D8DA-CA56-A22B`, and whether this device identifier has already opted-out in the past. The details of how a user's fingerprintable attributes are combined into a device identifier are proprietary and are unknown to us.

PetPortal: Boda *et al.* have created a cross-browser fingerprinting suite as part of their research in browser fingerprinting [5], available at `http://fingerprint.pet-portal.eu/`. As in the case of BlueCava, the user's fingerprintable attributes are delivered to the server, which then sends back a device identifier and whether the device identifier belongs to a new, or returning, user.

Coinbase: Even though Coinbase does not provide an opt-out page, the algorithm for deriving a device identifier from a user's fingerprintable attributes is part of their client-side JavaScript code. More specifically, when a site includes remote JavaScript code for obtaining Coinbase's "Pay with Bitcoin" button, the remote code creates an `iframe`, in which the fingerprinting code runs [6]. Once the fingerprint is computed, it is MD5-ed and then set as a cookie on the user's machine. When the user clicks on the payment button, her fingerprint will be automatically submitted to the Coinbase server via the user's cookies.

fingerprintjs: Finally, fingerprintjs is an open-source fingerprinting library which, like Coinbase, runs fully on the client-side. fingerprintjs is inspired by Panopticlick [8] and contains most of its features. Interestingly, fingerprintjs is also the only library, that we encountered, that fingerprints a user's machine using the HTML5 canvas as proposed by Mowery *et al.* [14]. It should be noted that it is not yet entirely clear how effective canvas-based fingerprinting is, in practice. Lastly, note that fingerprintjs does not support JavaScript-based font detection.

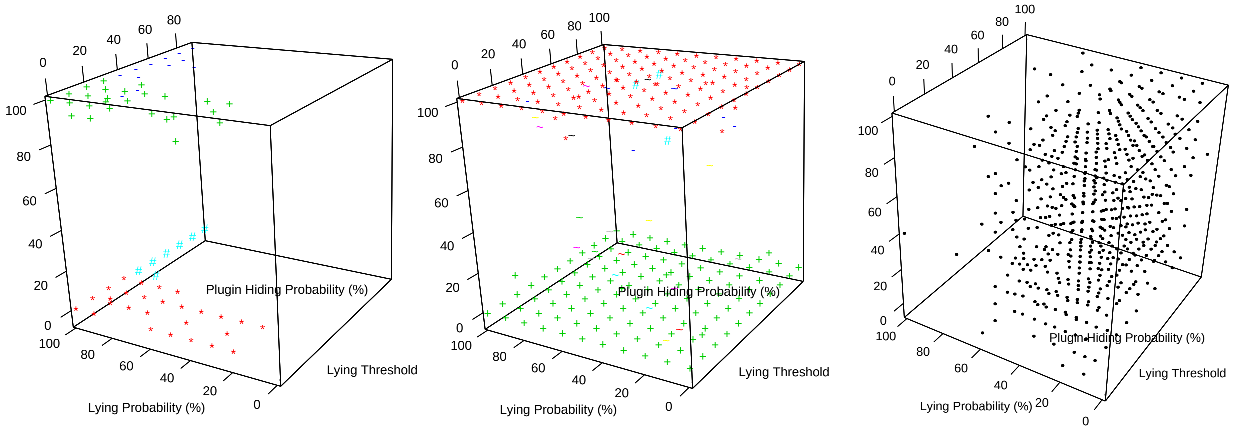
5.2.1 Experimental Setup

In all four cases, the individual fingerprinting providers gave us a way of assessing the efficacy of PriVaricator, simply by visiting each provider multiple times using different randomization settings, and recording the fingerprint provided by each oracle. To explore the space of possible policies in detail, we performed an automated experiment where we visited each fingerprinting provider 1,331 times, to account for 11^3 parameter combinations, where each parameter of our randomized policy (lying threshold, lying probability, and plugin-hiding probability) ranged from 0 to 100 in increments of 10.

Before we present the results of this experiment we would like to elaborate on two of our decisions.

Panopticlick: We chose against the use of Panopticlick since the feedback that Panopticlick provides to the user is of a semi-qualitative nature, *e.g.* "You are unique among 3 million users". This type of statement does not allow us to compare the fingerprints received from multiple visits, and thus does not allow us to reason about the effect that our parameterized randomization policies have against it. In addition, since all the sets of attributes collected by the studied fingerprinters are supersets of Panopticlick, we have no reason to expect that our results would have been dramatically different, had we been able to include Panopticlick in our study.

Focusing on the Random(0..100) policy: Even though we propose multiple lying policies about our offsets (Zero value, Random(0..100), and ± 5% Noise) we only run this set of experiments using the Random(0..100) policy. We argue that, given



(a) BlueCava **(b)** fingerprintjs **(c)** PetPortal

Figure 7: Distribution of device identifiers across three different fingerprinters.

the workings of JavaScript-based font detection, any of our proposed lying policies would give equivalent results. This is because, as explained in Section 2.2, fingerprinting providers first establish ground truth using a font-family that they expect to be present on all devices (*e.g.* sans), and then compare the `offsetWidth` and `offsetHeight` of text using other font-families against that ground truth.

In all of our policies, PriVaricator will cause deviations from that ground truth and may even poison the fingerprinters' ground truth itself, if the ground truth is acquired *after* the lying threshold ($\boldsymbol{\theta}$) of our policies is surpassed. As such, all of our lying policies will cause the fingerprinter to believe that our machine has fonts that it actually does not have (false positives). False negatives are a rare phenomenon since our lying policy would have to inadvertently set the values of `offsetWidth` and `offsetHeight` to perfectly match the ground truth.

5.2.2 Results

The results of this set of experiments are shown in Figure 7. In all three scatter-plots, the *x*-axis represents the probability of lying, the *y*-axis represents the lying threshold, while the *z*-axis represents the probability of hiding each individual plugin in our browser's list of plugins. For the first two graphs, colors and symbols represent *clusters* of fingerprints, *e.g.*, all green plus signs denote the same fingerprint, within a given service.

BlueCava and Coinbase: For BlueCava, in Figure 7a one can see that their fingerprinting algorithm can only track us mostly along the left-most edges of the graph. For example, when our plugin-hiding probability is 0, *i.e.*, we always show all plugins, and the lying threshold ranges from 0 to 30, we get the same fingerprint (red asterisks at the "bottom" of the graph). What is also interesting is how fingerprints change when the lying threshold is less than 30, or greater than 30.

Based on our understanding of BlueCava's algorithm, this is because when our threshold is lower than 30, we then poison the ground truth of the JavaScript-based font detection algorithm, which leads to having an increased number of fonts marked as "present." This theory is further strengthened by the observation that the size of the clusters under that threshold is larger than the size of clusters above it, since the lying probability has less of an effect when the ground truth is wrong.

At the same time, it is also evident that most of the cube is empty, that is, in all points other than the ones present, every fingerprint was *unique*, yielding 94.58% of all fingerprints being unique. This shows how fragile BlueCava's identification is against our randomization policies. Coinbase's results were very similar to Bluecava's, thus we chose to briefly discuss them in the paper's Appendix.

fingerprintjs: For fingerprintjs, Figure 7b, the arrangement of points is visibly different from BlueCava's. Since this library does not have support for JavaScript-based font detection, our choices of lying probability and lying threshold have no effect. What has the most influence is the value of the plugin-hiding probability. As a matter of fact, it is evident that fingerprintjs can only track us either when we have no plugins showing, *i.e.*, hiding probability equals 100%, or all plugins showing, *i.e.*, hiding probability is 0%.

In nearly all intermediate points (78.15% of the total set of collected fingerprints), randomness works

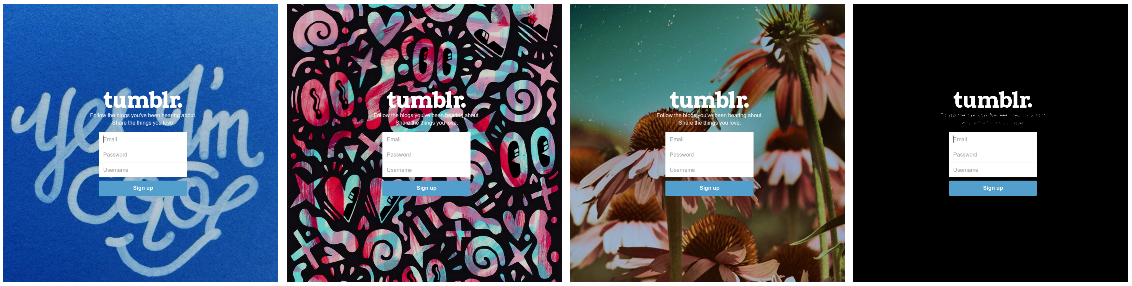

tumblr.
Follow the blogs you've been hearing about.
Share the things you love.
Email
Password
Username
Sign up

Figure 8: The first three screenshots show `tumblr.com` and its ever-changing backgrounds. The last image is an automatically-generated mask for assessing the true breakage caused by PriVaricator. Black areas denote masked-out content.

in our favor by returning different sets of plugins, which, in turn, result in different fingerprints. These results show how important it is to *combine* randomization approaches in order to deter fingerprinters who do not utilize all fingerprintable attributes of a user's browsing environment.

PetPortal: Lastly, Figure 7c, shows the results of our experiment against PetPortal. Note that for this figure, because of the large number of clusters, to make the results more readable, we show all the configurations that resulted in *unique fingerprints*, instead of showing clusters of same fingerprints. It is evident that PetPortal succeeds more in tracking us than BlueCava, Coinbase, and fingerprintjs.

In contrast with the other three services, we were able to get unique fingerprints in "only" 43.61% of the 1,331 parameter combinations. One can notice from this graph that we defeat tracking when the lying probability is in the range of 10% to 60%. When the lying probability exceeds 60% we begin lying too often, which likely results in having most fonts marked as "present." There, we also see a lack of effect from the plugin-hiding probability which cannot recover us from being accurately fingerprinted. This likely means that PetPortal places more weight on the discovered fonts, and less on the claimed plugins.

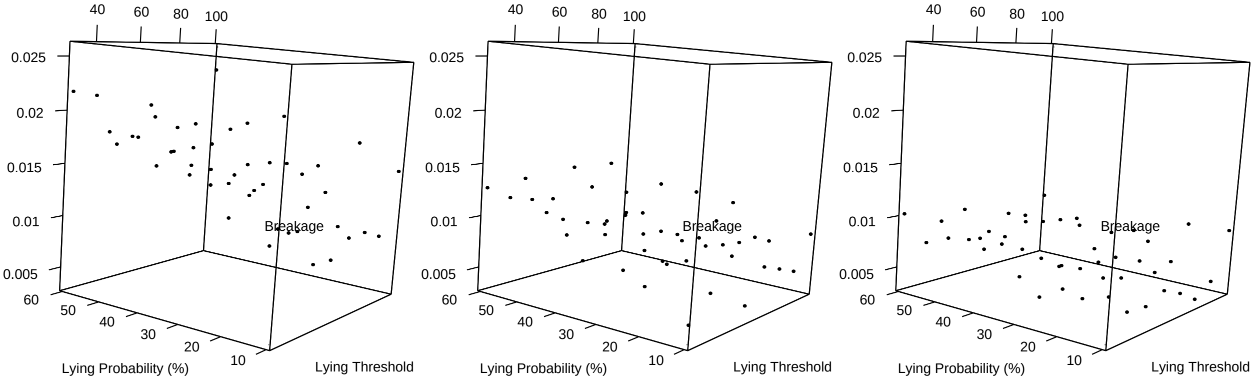
Summary: Overall, our experiments showed that, while the specific choices of each fingerprinter affect the uniqueness of our fingerprints, PriVaricator was able to deceive all of them for a large fraction of the tested combination settings. Moreover, the presence of clusters of identical fingerprints demonstrates that most fingerprinting providers derive a fingerprint by following a more complicated approach than just hashing all fingerprintable attributes together. Finally, comparatively speaking, PetPortal was the most resistant to PriVaricator.

5.3 Assessing the Breakage

In the previous section, we demonstrated that PriVaricator was able to withstand fingerprinting by measuring the number of unique fingerprints received, for a total of 1,331 settings combinations. By computing the intersection of the points resulting in unique fingerprints across all four fingerprinting providers, (essentially identical to PetPortal's results), we obtain a range of settings, all of which provide prevention from reliable fingerprinting. In this section, we assess the level of breakage of benign sites for each of those parameter combinations.

Experimental setup: The `offsetWidth` and `offsetHeight` properties of an element provide information to a JavaScript program about the size of that element, as is currently rendered on a user's screen. When PriVaricator lies about these values, it creates a potential for visual breakage. For example, by reporting that an element is smaller than it actually is, PriVaricator could cause the page to place it in a smaller container, hiding part of its content from the user. Numerically, we define *breakage* as the fraction of pixels that are different when a site is loaded with a vanilla browser (PriVaricator turned off) and with PriVaricator.

To assess the breakage, we instrumented Chromium to visit the main pages of the top 1,000 Alexa sites, for 48 different combinations of lying probability and lying threshold; these were the parameter combinations that resulted in unique fingerprints, as described in the previous section. To contain the dimensionality of this experiment, we statically assigned the plugin-hide probability to zero (showing all plugins) since we reasoned that the main pages of the most popular sites of the web likely behave the same for users with different plugins. At every site visit, the browser waited for 25 seconds and then captured a screenshot (1,050x850 pixels) of the rendered content.



(a) Random(0..100) policy **(b)** Zero policy **(c)** ± 5% Noise policy

Figure 9: Breakage caused by PriVaricator, given a series of possible configurations and a randomization policy

In order to separate between visual differences caused by PriVaricator, and visual differences caused by the inherent variation of a site, *e.g.* ads, image carousels, and newly posted content, we collected a new vanilla-browser screenshot every ten visits of a page, resulting in a total of five extra screenshots.

Since any visual variation detected on these five screenshots can be attributed to a website's dynamic content, we computed a visual mask of differences appearing on them, and used it when comparing a screenshot captured using a specific policy parameter combination, to the vanilla one. This mask can be applied to all PriVaricator screenshots to exclude the naturally varying parts of a page from subsequent breakage comparisons. For illustration, Figure 8 shows three different vanilla-browser screenshots of `tumblr.com` and the computed mask.

Finally, while in the previous section the choice of randomization policy was not important, in this section, different policies are likely to produce different visual results, *e.g.*, receiving a value that is 5% off the expected one, versus receiving a value that is completely random. Thus, the entire experiment had to be repeated three times, once for every randomization policy: (a) Random(0..100); b) Zero; and c) ± 5% Noise). Overall, we collected a total of approximately 159,000 images, occupying 54 GB of disk space, which we compared in order to quantify the breakage caused by PriVaricator.

Policy	**Min**	**Mean**	**Max %**
Random(0..100)	0.8%	1.4%	2.1%
Zero	0.4%	0.8%	1.3%
± 5% Noise	**0.4%**	0.6%	0.9%

(a) Summary of breakage results.

Policy	**Min**	**Mean**	**Max %**
Random(0..100)	0.8%	1.5%	2.3%
Zero	0.4%	0.9%	1.4%
± 5% Noise	**0.4%**	0.7%	1.0%

(b) Breakage when ignoring pages with masked content greater than 30% (approx. 84% of pages remaining).

Figure 10: Breakage summary with and without including the sites with large masks.

Results: The results of our breakage experiments are first detailed in Figure 9 and then summarized in Figure 10. Figure 10a presents the minimum, average, and maximum breakage for all three policies when considering the fractions of different pixels across all sites. Since we noticed that, in some cases, the computed masks were too *large*, we also calculated the breakage of sites when ignoring sites that had masks with size larger than 30% of the total image; this is shown in Figure 10b. This way, we ignore sites that would give PriVaricator an unfair advantage by hiding real breakage under a site's natural variation. While the latter set of numbers is slightly larger than the former, it is evident, not only that the ± 5% Noise incurs the least breakage but that the breakage itself is, on average, less than 1%.

Every point in Figure 9 is the average breakage of all 1,000 Alexa sites visited with PriVaricator using a specific $\langle P(lie), \theta \rangle$ configuration, and one of our three lying policies. For instance, in Figure 9b, the average breakage of sites when visited by PriVaricator configured with a lying probability equal to 10% and a lying threshold of 30 accesses is 0.004, under the Zero policy. In other words, the sites visited by PriVaricator with that specific combination of settings had, on average, 0.4% different pixels when compared to the vanilla screenshots.

For the breakage caused by the Random(0..100) policy (Figure 9a) and Zero policy (Figure 9b), one can discern a positive relationship between the lying probability and the resulting breakage. This relationship makes intuitive sense. The more often PriVaricator lies using these policies, the more often a website receives an unexpected value of 0, or a random number between 0 and 100. On the other hand, this relationship is significantly weaker in ± 5% Noise policy results (Figure 9c). We argue that this is because the modified offset value is relatively close to the value that a script would otherwise expect, thus minimizing the number of sites breaking because of such small modifications.

Inspecting breakage: Finally, in order to understand how a user would experience potential breakage, we manually reviewed the 100 screenshots (under the ± 5% Noise policy) with the largest reported breakage. In this analysis, we discovered that in only 8 cases, the differences could be attributed to PriVaricator. The rest of the screenshots (92/100) were very different from the vanilla screenshots due to a site's inherent variations and errors, not captured in any of the five vanilla screenshots. In many cases, the sites would show an "in-page" pop-up asking the user to participate in a survey. Usually, this pop-up would add a semi-transparent gray overlay over the page, causing our automatic comparison algorithms to report a very large visual difference.

Next to surveys, the reported breakage was due to missing or not-fully loaded ads, error-pages and image carousels. In one case, PriVaricator had caused a slight stretch of a site's background image. While this led to a large computed breakage, users would not notice the change if they could not compare the page with the original non-stretched version.

Finally, we manually inspected the sites making the most use of offset accesses (listed in Figure 4) by visiting them with PriVaricator and clicking on a few links on each site. All sites were operational and usable, with the only difference being the location and movement of some objects, e.g. moving ads, whose motion and placement was slightly affected by the randomization policies of PriVaricator.

Summary: Overall, the results of our breakage experiments show that the negative effect that PriVaricator has on a user's browsing experience is negligible. Moreover, our manual analysis revealed that we have likely overestimated the breakage since most of the pages with the highest reported breakage turned out to be false positives.

Our low breakage results also allow us to avoid the temptation of cherry-picking configurations from Figure 9c, which would likely lead to issues related to over-fitting. Instead, any of the many parameter configurations could be picked for deployment, e.g., picking one at random when a user starts a private mode session. We opine that an average breakage of 0.7% (likely an upper bound with the actual damage as much as 10*x* less) provides an acceptable trade-off for the extra privacy that the user gains in return.

6 Discussion

Transparency: As with any defense strategy there is a question of *transparency.* Since PriVaricator is using randomness to report different values for popular fingerprintable attributes, a motivated fingerprinter could test for the presence of unexpected randomness, *e.g.*, by enquiring about the dimensions of an element 100 times, and then check for the presence of differences in the reported dimensions of the, otherwise unmodified, element. One possible solution that alleviates this transparency issue is setting up a "lie cache", where the browser would report the same false value for multiple inquires about the dimensions of the same unmodified element, or for many identical elements. Attempts to discern "hidden" real values that are misrepresented by PriVaricator (e.g. through a statistical analysis of the returned values) would also be alleviated by such a lie cache. We leave the exploration of this solution and other similar ones, for future work.

Deployment challenges: The key advantages of PriVaricator are its negligible overhead and the relative ease of porting. It is easy to underestimate the importance of low overhead, but given the current emphasis on browser performance, it is unlikely that a privacy solution that suffers a large performance hit will be deployed. Our design of PriVaricator has emphasized minimal modifications to existing technology, which leads to small overhead and negligible porting costs; overall, the automatically generated patch of our modifications to Chromium (including comments) is only 947 lines long.

7 Related Work

In this section, we provide an overview of literature focusing on browser fingerprinting.

History of fingerprinting: The work of Mayer [12] and Eckersley [8] presents large-scale studies that show the possibility of effective stateless web tracking via only the attributes of a user's browsing environment. These studies prompted some follow-up efforts [9, 21] to build better fingerprinting libraries. Yen *et al.* [23] performed a fingerprinting study by analyzing month-long logs of Bing and Hotmail and showed that the combination of the User-agent HTTP header with a client's IP address were enough to track approximately 80% of the hosts in their dataset.

While the majority of fingerprinting efforts have focused on fonts and plugins, Mowery and Shacham proposed fingerprinting through the rendering of text and WebGL scenes to a `<canvas>` element [14]. Different browsers will display text and graphics in a different way which, however small, can be used to differentiate and track users. The downsides to this method are that these technologies are only available in the latest versions of modern browsers and that the canvas-generated entropy is not sufficient for it to be used as the only fingerprinted attribute.

Extensions: Several user-agent-spoofing browser extensions (UserAgent Switcher, UserAgent RG, UAControl, Ultimate User Agent Switcher, etc.) have emerged as well, primarily for Chrome and Firefox browsers. As analyzed by Nikiforakis *et al.* [16], these extensions may actually be counterproductive, since they considerably narrow down the population of possible users to fingerprint, in addition to frequently reporting impossible combinations of environment variables. PriVaricator goes deeper than these extensions, focusing on portions of the environment that can be spoofed to break fingerprinters, while not significantly affecting other sites.

Firegloves, proposed by Boda *et al.* [4, 5] but no longer supported, was a browser extension that attempted to frustrate fingerprinting attempts by faking the screen resolution and timezone, presenting an empty `navigator.plugins` list, limiting the number of font families allowed to load per tab, and randomizing the return value of `offsetWidth` and `offsetHeight` of elements. As we showed in Section 5.2, presenting an empty list of plugins is as bad as presenting a full list of plugins, since the majority of browsers support at least one plugin. In contrast, PriVaricator chooses to randomize the existing list of plugins which results in a large number of different plugin combinations. The randomized offset value of Firegloves is a random value between 0 and 1,000. As shown in our evaluation of site breakage, Section 5.3, this randomization approach produces the most breakage. Moreover, our randomized values were constrained to a range of 0 to 100, meaning that if one assumes a positive correlation between the size of the set of possible offset values and breakage, Firegloves has the potential to cause considerably more breakage. Lastly, since Firegloves performs its operations through getters and setters, it suffers from the transparency and compatibility problems we mentioned in Section 4.

Side channels: Researchers have proposed a variety of side channels for browser fingerprinting, although we are not aware of them being used in practice. Mowery *et al.* [13] proposed the use of benchmark execution time as a way of fingerprinting JavaScript implementations, under the assumption that specific versions of JavaScript engines will perform in a consistent way. Closely related is the work of Mulazzani *et al.* who used the errors produced by JavaScript engines when executing standard test suites to differentiate between browsers [15].

Olejnik *et al.* [17] show that web history can also be used as a way of fingerprinting for tracking purposes. The authors make this observation by analyzing a corpus of data from when the CSS-visited history bug was still present in browsers. Today, however, all browsers have corrected this issue and thus, extraction of a user's history is not as straightforward, especially without user interaction [22]. In more recent work, Dey *et al.* propose the use of detectable hardware imperfections of smartphone sensors, as a way of tracking their users [7].

We are not aware of wide-scale fingerprinting on the web using any of these side channels. This is part of the reason we chose to focus on explicit fingerprint in PriVaricator.

8 Conclusion

This paper proposes PriVaricator, an addition to privacy modes present in modern browsers. The goal of PriVaricator is to combat stateless tracking, which is being done primarily using device-fingerprinting JavaScript code. We use careful randomization as a way to make subsequent visits to the same fingerprinter difficult to link together. We evaluate several families of *randomization functions* to find those that result in the best balance between fingerprinting prevention and breaking existing sites. While our implementation has focused on randomizing font- and plugin-related properties, we demonstrate how our approach can be made general with pluggable randomization policies.

Our best randomization policies reliably prevent *all* fingerprinting when tested with several well-known device fingerprinting providers, while incurring minimal damage on the content of the Alexa top 1,000 sites. Furthermore, we found the runtime overhead of PriVaricator to be negligible.

References

[1] G. Acar, M. Juarez, N. Nikiforakis, C. Diaz, S. Gürses, F. Piessens, and B. Preneel. FPDetective: Dusting the Web for fingerprinters. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2013.

[2] D. Baron. Preventing attacks on a user's history through CSS:visited selectors. `http://dbaron.org/mozilla/visited-privacy`, 2010.

[3] BlueCava. Measuring Mobile Advertising ROI Using BlueCava Device Identification. 2011.

[4] K. Boda. Firegloves. `http://fingerprint.pet-portal.eu/?menu=6`.

[5] K. Boda, A. M. Földes, G. G. Gulyás, and S. Imre. User tracking on the web via cross-browser fingerprinting. In *Proceedings of the Nordic Conference on Information Security Technology for Applications (NordSec)*, 2012.

[6] Bitcoin Payment Buttons - Bitcoin Merchant Tools. `https://coinbase.com/docs/merchant_tools/payment_buttons`.

[7] S. Dey, N. Roy, W. Xu, R. R. Choudhury, and S. Nelakuditi. AccelPrint: Imperfections of Accelerometers Make Smartphones Trackable. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2014.

[8] P. Eckersley. How Unique Is Your Browser? In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, pages 1–17, 2010.

[9] E. Flood and J. Karlsson. Browser fingerprinting, 2012.

[10] C. Fournet, N. Swamy, J. Chen, P.-E. Dagand, P.-Y. Strub, and B. Livshits. Fully abstract compilation to javascript. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, Jan. 2013.

[11] iovation. Solving online credit fraud using device identification and reputation. 2007.

[12] J. R. Mayer. Any person... a pamphleteer. Senior Thesis, Stanford University, 2009.

[13] 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.

[14] K. Mowery and H. Shacham. Pixel perfect: Fingerprinting canvas in HTML5. In *Proceedings of the Workshop on Web 2.0 Security and Privacy (W2SP)*. IEEE Computer Society, May 2012.

[15] M. Mulazzani, P. Reschl, M. Huber, M. Leithner, S. Schrittwieser, and E. Weippl. Fast and reliable browser identification with javascript engine fingerprinting. In *Web 2.0 Workshop on Security and Privacy (W2SP)*, May 2013.

[16] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 541–555, 2013.

[17] Ł. Olejnik, C. Castelluccia, and A. Janc. Why Johnny Can't Browse in Peace: On the Uniqueness of Web Browsing History Patterns. In *Proceedings of the Workshop on Hot Topics in Privacy Enhancing Technologies (HOTPETS)*.

[18] K. Sivaramakrishnan. The Drawbridge Cross Device Matching Technology.

[19] ThreatMetrix. Device Fingerprinting and Fraud Protection Whitepaper.

[20] Tor: Cross-Origin Fingerprinting Unlinkability. `https://www.torproject.org/projects/torbrowser/design/#fingerprinting-linkability`.

[21] V. Vasilyev. fingerprintjs library. `https://github.com/Valve/fingerprintjs`, 2013.

[22] Z. Weinberg, E. Y. Chen, P. R. Jayaraman, and C. Jackson. I still know what you visited last summer: Leaking browsing history via user interaction and side channel attacks. In *Proceedings of the IEEE Symposium on Security and Privacy*, SP '11, pages 147–161, 2011.

[23] T.-F. Yen, Y. Xie, F. Yu, R. P. Yu, and M. Abadi. Host fingerprinting and tracking on the Web: Privacy and security implications. In *Proceddings of the Network and Distributed System Security Symposium (NDSS)*, 2012.

A Coinbase

Figure 11 shows the distribution of non-unique device identifiers when testing PriVaricator against Coinbase. Colors and symbols represent clusters of identical values, e.g., all green plus-signs denote the same device identifier as generated and reported by Coinbase. As was the case with Bluecava (discussed in Section 5.2), PriVaricator deceives Coinbase in the vast majority of cases (97.86% of the extracted device identifiers were unique).

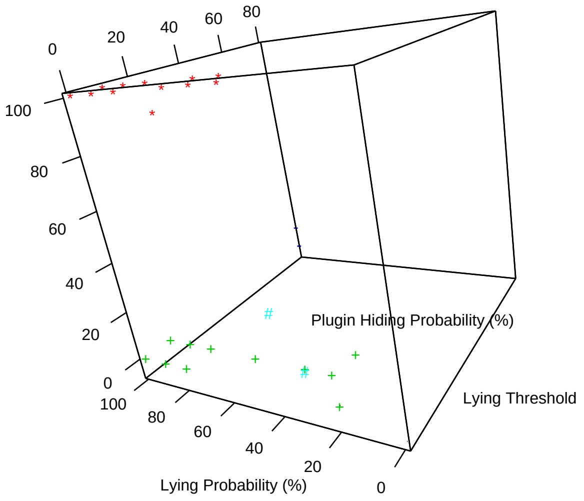


Figure 11: Distribution of non-unique device identifiers of PriVaricator against the Coinbase service.