EssentialFP: Exposing the Essence of Browser Fingerprinting

Alexander Sjösten*† Daniel Hedin*‡ Andrei Sabelfeld*

*Chalmers University of Technology
†TU Wien
‡Mälardalen University

***Abstract*—Web pages aggressively track users for a variety of purposes from targeted advertisements to enhanced authentication. As browsers move to restrict traditional cookie-based tracking, web pages increasingly move to tracking based on browser fingerprinting. Unfortunately, the state-of-the-art to detect fingerprinting in browsers is often error-prone, resorting to imprecise heuristics and crowd-sourced filter lists. This paper presents EssentialFP, a principled approach to detecting fingerprinting on the web. We argue that the pattern of (i) gathering information from a wide browser API surface (multiple browser-specific sources) and (ii) communicating the information to the network (network sink) captures the essence of fingerprinting. This pattern enables us to clearly distinguish fingerprinting from similar types of scripts like analytics and polyfills. We demonstrate that information flow tracking is an excellent fit for exposing this pattern. To implement EssentialFP we leverage, extend, and deploy JSFlow, a state-of-the-art information flow tracker for JavaScript, in a browser. We illustrate the effectiveness of EssentialFP to spot fingerprinting on the web by evaluating it on two categories of web pages: one where the web pages perform analytics, use polyfills, and show ads, and one where the web pages perform authentication, bot detection, and fingerprinting-enhanced Alexa top pages.**

***Index Terms*—web security and privacy, browser fingerprinting, JavaScript, information flow**

1. Introduction

Web pages aggressively track users for a variety of purposes such as targeted advertisement, enhanced security, and personalized content [46]. Web tracking is subject to much debate [33], [19] that, in the light of privacy-enhancing legislation [14], has led the major browser vendors to introduce anti-tracking measures.

From cookies to fingerprinting. While cookies were, traditionally, used to keep track of users, a growing awareness of privacy concerns (e.g., the "do not track" flag [32]) has made cookies less efficient as the only means of tracking as shown by Yen et al. [73]. To compensate, web pages are moving to collect *browser fingerprints*, where seemingly benign browser features can be combined to uniquely identify users [59]. Recent studies show that (i) browser fingerprinting is becoming increasingly prevalent [36], [46]; (ii) modern techniques include hardware fingerprinting through the Canvas [62] and WebGL [62], [42] APIs; and (iii) on average, a fingerprint can track a browser instance for 54.48 days [72]. Today there are several open-source browser fingerprinting libraries, such as FingerprintJS [12], ImprintJS [15], and ClientJS [6], where FingerprintJS is the most updated and supersedes ImprintJS and ClientJS to a large extent. These libraries are highly configurable, allowing developers to define what specific browser features should be used by enabling flags corresponding to the desired features. From analyzing newer data, researchers have also shown that the number of uniquely identifiable users based on the fingerprint has gone down. However, these non-unique fingerprints are fragile, and if a user modifies a few features of the fingerprint, there is a high probability the fingerprint will become unique [49].

Privacy-violating fingerprinting. As with tracking in general, web pages fingerprint users for a variety of purposes. To thwart the privacy-violating fingerprinting efforts browser vendors have proposed mitigations, which include randomizing the output of known fingerprinting vectors by Brave [34], using privacy budgets by Chrome [23], blocking third-party requests suspected of being tracking related by Edge [18] and Firefox [13] based on, e.g., the Disconnect list [9], and making more devices look identical by Safari [28]. The diversity of these techniques shows that each comes with its own pros and cons, with no clear principle on how to detect fingerprinting. This is further reflected in a large number of both false positives and negatives [1], [2], [3], [4].

In addition to the effort in limiting tracking and fingerprinting by browser vendors, a user can install browser extensions like AdBlock, Privacy Badger, or Ghostery to help block privacy-intrusive scripts. These extensions use techniques ranging from crowd-sourced filter lists (collections of rules dictating what should be blocked), analyzing behavior, and using anonymous data from the users. As an example, the crowd-sourced filter lists have the obvious limitations: to keep the filter lists up-to-date is time-consuming, and when fingerprinting scripts are added to the filter lists they can easily be evaded by serving them from different Internet domains [43].

Recent research approaches to limit fingerprinting include randomization of features [63], [58], [70], modifying the fingerprint per session [69], [47], and making users look identical through virtualization [60], [48]. We discuss these and further related approaches in Section 7.

As seen above, current techniques for fingerprint detection focus on specific vectors. Hence, they fall short of addressing the general case: there is currently no uniform

solution for identifying fingerprinting. This leads us to our first research question: *RQ1: What is the essence of browser fingerprinting?*

Different types of fingerprinting. What makes the problem of fingerprinting intricate is that not all fingerprinting is "bad" fingerprinting [59]. Indeed, fingerprinting can be justified to increase security when used to improve e.g. bot detection, fraud detection, and protection against account hijacking [37]. Where to draw the line between "bad" and "good" fingerprinting is an open and arguably subjective question. Approaches that try to draw this line are bound to result in both false positives and negatives.

The stance of this paper is thus neutral: we focus on identifying the *presence* of fingerprinting, hence providing necessary input into the decision process (by the user and browser) on whether to allow it or not. This motivates our second research question: *RQ2: How do we reliably expose fingerprinting scripts in a principled way, without relying on ad-hoc heuristics and crowd-sourcing, while at the same time not having to judge the fingerprinting as "bad" or "good"?*

Fingerprinting. Our key observation is that the essence of fingerprinting can be captured by: (i) gathering information from a wide browser API surface (the *API imprint*) and (ii) communicating the information to the network (network sink). The communication in (ii) may either be direct (via, e.g., `XMLHttpRequest`) or indirect (via, e.g., the cookie) and may be done by sending the raw data piece by piece or (more commonly) as a precomputed fingerprint.

The *flow* of information is key to reliable detection: we must track how information flows from the API imprint to the network sinks. Only looking at an application's API access pattern does not suffice due to the risk for false positives (every use of the API would count towards fingerprinting). In particular, in the presence of *polyfills*, the API access patterns naturally become rather large, thus increasing the risk of false positives significantly.[1]

Lightweight information flow control. Based on this, we propose EssentialFP, a principled approach that utilizes dynamic *Information-Flow Control (IFC)* as a means to expose fingerprinting. EssentialFP utilizes a variant of dynamic IFC known as *observable tracking* [38], [68]. By labeling the values originating from the API imprint with their access paths and capturing (the accumulation) of labels exfiltrated via network sinks, observable tracking ensures that all important flows are correctly accounted for. We implement this approach in JSFlow [50], a state-of-the-art dynamic IFC monitor for ECMA-262 v5 that allows fine-grained information flow tracking. Although IFC techniques can be extended to handle timing and other side-channel attacks, JSFlow does not support this; such attacks are, thus, currently out of scope for EssentialFP.

Scope. We would like to stress that our work is a *feasibility study* rather than a *scalability study*. The first step before deciding whether a high-performance monitor can be integrated into an existing JavaScript runtime is to understand what security and privacy benefits it will bring. Our focus is on providing a platform for experimenting with and providing a deep understanding of JavaScript on web pages; large-scale evaluation on thousands of web pages is not in scope of this work. This is in line with previous work on information-flow tracking for JavaScript [45], [52], [40], [50] whose strength is a deep understanding of JavaScript behavior rather than approximative analysis of thousands of pages. Future work on scalability will have its own challenges because existing JavaScript runtimes such as V8 are fast-moving targets with a focus on advanced performance optimizations. Yet, an encouraging indication is that Bichhawat et al. instrumented WebKit's JavaScript engine to implement dynamic IFC, with an average performance overhead of roughly 29% [40], showing it is possible to implement dynamic IFC with a tolerable overhead.

1. Polyfills, such as Modernizr [20] and core-js [8], are libraries intended to extend older browsers with support for new features. To be able to do this they probe and enhance the execution environment by injecting any missing features.

Contributions. In summary, this paper offers the following contributions:

(i) We develop EssentialFP a principled approach to fingerprinting detection based on observable tracking (Section 3), define the sources and sinks, and design a metric that allows us to characterize fingerprinting patterns via aggregated labels.
(ii) We present the design and implementation of EssentialFP to allow JSFlow to track information within web APIs, and how to track label combinations of known fingerprinting patterns (Section 4).
(iii) We present an empirical study, where we visit web pages based on different categories (non-fingerprinting and fingerprinting), demonstrating the effectiveness of EssentialFP (Section 5).

The code of our tool and its benchmarks are available online [10].

2. Observable Information-Flow Control

There are various forms of IFC, all sharing the same fundamental concepts but differing in how information is tracked and what security guarantees they provide. This paper utilizes observable IFC and we refer the reader to [66], [51] for other common variants.

Observable IFC is a form of dynamic IFC, i.e., that takes place at runtime. Dynamic analyses have the benefit of being able to handle the challenges posed by JavaScript including dealing with obfuscation and minification, two prevalent techniques that are known to cause issues for static analyses.

To track how the information flows all values are given a runtime security label. The security labels of values taking part in a computation are modified during execution to reflect the flow of information caused by the computation. This is done by tracking two types of information flows: *explicit* and *implicit* flows. Explicit flows correspond to data flows [44] in traditional program analysis, and occurs when one or more values are combined into a new value. Implicit flows correspond to control flows [44] in traditional program analysis, and occurs between values when one value indirectly influences another value via the control flow of the program.

Similar to many other IFC techniques, observable tracking maintains a security label associated with the control flow, the so-called *pc* label, to track implicit flows.

The *pc* label is used to ensure any values influencing the control flow are taken into account when computing the labels of side effects. Implicit flows in an application are not only of theoretical importance [54], [68]. If implicit flows are not tracked, important flows are missed. Consider the following code taken from FingerprintJS.

```
var getNavigatorPlatform = function (options){
  if (navigator.platform) {
    return navigator.platform
  } else {
    return options.NOT_AVAILABLE
  }
}
```

If `navigator.platform` is present, there is an observable implicit flow from it to the return value of the function. This code snippet from FingerprintJS represents a common pattern where the link between the original API source and the sink would be lost in the negative case.

3. Approach

To allow EssentialFP to detect browser fingerprinting, our approach relies on a precomputed baseline API imprint that captures all parts of the API that contain fingerprinting-sensitive information. This is used to label any information that originates from the baseline API imprint with the access path, causing all fingerprinting-sensitive information to carry its origin as a security label. During execution, the observable tracking ensures values that reach the network sinks are correctly labeled. This approach is more robust compared to, e.g., OpenWPM [21], [46] that instruments access to the JavaScript API, links every access to the corresponding script, and stores everything in a database for offline analysis, since it does not implicitly assume that access equates exfiltration. As discussed above, this is important in the presence of, e.g., polyfills, that have large API access patterns but do not exfiltrate any information. Similar to OpenWPM, the analysis of the collected labels in our approach is done after the execution of the page, but nothing prevents a runtime solution where information would be allowed to be exfiltrated until a certain threshold has been met. Such a solution would be related to the use of a *privacy budget* [7], with the difference that it measures the budget on the information exfiltrated from the browser, rather than the read information.

3.1. Computing the API imprint

In order to create the baseline API imprint locally, we use three known and widely used open-source fingerprinting libraries: FingerprintJS [12], ImprintJS [15], and ClientJS [6]. It is worth noting that a full list of sources for entropy would be ideal, but this information is not readily available. By using the three aforementioned libraries a high-quality approximation that captures the state-of-the-art in fingerprinting is possible.

The baseline API imprint is computed by combining the API imprints of FingerprintJS, ImprintJS, and ClientJS running in isolation on pages crafted for the purpose. The API imprints for FingerprintJS and ImprintJS were generated by executing the respective library with all fingerprinting features turned on. As ClientJS does not have the same configurability the API imprint was generated by calling the ClientJS function `getFingerprint`.

Further, to be able to analyze the kind of fingerprinting detected, API imprints for each feature of FingerprintJS and ImprintJS were also created.

3.2. Detecting Fingerprinting

The baseline API imprint identifies the part of the execution environment that contains fingerprinting-sensitive information. By labeling all values originating from this part with the access path, tracking the information flow dynamically, and monitoring the label creation and flow during the execution we are able to detect fingerprinting in applications. For each page and API endpoint, we accumulate the labels of all values reaching the endpoint. From this set of endpoints, we select the potential network sinks, i.e., endpoints that can be used to communicate information, such as `XMLHttpRequest`, or store information which can be transmitted, such as `document.cookie`. The result of the label collection is two maps: one mapping network sinks to the accumulated label of exfiltrated values, and one mapping all script origins to the set of created labels. The maps allow us to analyze each web page for both internal creation of fingerprints where information is collected and combined, and fingerprinting, i.e., where the collected information is sent via a network sink. More precisely we can distinguish between the following uses:

- Traditional use: information is gathered, composed, and sent (detected as both internal creation of fingerprints and fingerprinting)
- Piece by piece: information is gathered and sent (detected as fingerprinting without internal creation of fingerprints)
- Local use: information is gathered, composed, and used, but not sent (detected as internal creation of fingerprints without fingerprinting)
- No fingerprinting: no information is composed, used, or sent (detected as neither internal creation of fingerprints nor fingerprinting)

First, to identify the presence of internal creation of fingerprints, we measure the maximum overlap of the created labels for each script with the baseline API imprint. This gives us a good indication that a fingerprint is computed, since a large overlap of the wide baseline API imprint is unlikely for isolated scripts in a normal application. Second, to identify the presence of fingerprinting, we compute the largest overlap of each identified network sink. While somewhat direct, this approach suffices given how fingerprint exfiltration is currently implemented. If the exfiltration becomes more sophisticated by, e.g., using different sinks, a more precise approach of handling the sinks will be required both to identify different sinks but also to differentiate between uses of the same sink. This would require gathering more information about the sink, such as names or IP addresses. While possible using our approach, it was not necessary for the experiment.

In addition to detecting the presence of fingerprinting, it is also interesting to try to identify the kind of fingerprinting that takes place. To this end, we use the per-flag extracted API imprints of FingerprintJS and ImprintJS.

This gives us the possibility to characterize detected fingerprinting in terms of features and to identify common patterns using heatmaps. We note this is not possible for ClientJS as it does not have the same customization as FingerprintJS and ImprintJS.

4. Design and Implementation

To perform measurements and detect fingerprinting we have created an information flow aware browser: EssentialFP. EssentialFP is a modified version of Chromium that uses JSFlow [50] to execute JavaScript by deploying JSFlow as a library. By performing the injection inside Chromium, right before the script is sent to V8 for execution, we guarantee that all scripts, including dynamically injected scripts, are subject to the injection.

4.1. Extending JSFlow

The current release of JSFlow supports ECMA-262 v5 (ES5) [16] along with the mandated standard runtime environment. In order to use JSFlow to run actual web pages, JSFlow must be extended to support new features defined in ECMA-262 version 6 [29] and later standards (ES6+). In addition, JSFlow must also be extended to mediate between its own execution environment and the execution environment of the browser, as well as collecting the created and exfiltrated labels seen during page execution.

4.1.1. Extending JSFlow to ES6+. Initial attempts showed that a large portion of web pages use ES6+ features and, thus, do not run fully using an interpreter that only supports ES5. Extending JSFlow with support for ES6+ is a large undertaking that would require both extending the core engine of JSFlow, as well as the standard libraries. This entails rewriting a large portion of JSFlow. As a middle ground, we opted for adding support for ES6+ in JSFlow by using a combination of transpiling and polyfilling. Before any script is executed, JSFlow transpiles the code from ES6+ to ES5 using Babel [5]. This will produce a new program that should be semantically equivalent to the original but only use ES5 features. In addition to transpiling, we use polyfills to provide the parts of the ES6 runtime that JSFlow does not implement. Before any code is executed JSFlow executes a runtime bundle containing all support libraries and polyfills needed for proper execution. The runtime bundle extends the JSFlow runtime environment with polyfills from core-js [8] (for the ES6+ standard runtime), regenerator-runtime [27] (needed by some of Babel's transformations), and window-crypto [35] (for the `Window.crypto` functionality).

4.1.2. Connecting the Execution Environments. In order to use JSFlow to execute scripts in a browser environment, it is imperative to connect the JSFlow execution environment to the browser's V8 execution environment. We implement the connection using a bidirectional and connected mediation between V8 values and the JSFlow counterparts, where modifications done in either execution environment are reflected in the other. This effectively extends JSFlow with the APIs provided by the browser and allows scripts to interact with the browser as if they are running directly in V8. To implement the mediation, we scale the technique presented by Sjösten et al. [67] to full JavaScript in the browser setting.

Masquerading JSFlow values as V8 values. To mediate values from JSFlow to V8 the security labels must be removed: a process called *unlabeling*. For primitive values, unlabeling is the only mediation required since JSFlow builds on the primitive values of V8. Mediating non-primitive values such as `Functions` or `Objects` requires recursive mediation of their parts. To retain the connection between the original JSFlow value and the V8 value we use Proxies [24]. The proxies allow JSFlow objects to masquerade as V8 objects and perform recursive on-the-fly mediation on access, similar to that of the membrane pattern [61].

Masquerading V8 values as JSFlow values. To mediate values from V8 to JSFlow the security labels must be added: a process called *relabeling*. As above, for primitive values labeling is the only mediation required, while mediating non-primitive values such as `Functions` or `Objects` requires recursive mediation of their parts. The recursive mediation is provided by wrapper objects that implement the JSFlow internal Ecma-object interface and perform recursive on-the-fly mediation on access. The mediation follows a read-once-write-always semantics. That is, when a property is read, if it is defined on the host object, it is brought from the V8 execution environment, wrapped, and cached as an ordinary JSFlow property on the wrapper. Subsequent reads interact with the wrapper as an ordinary JSFlow object. When a property is written, it is written both unmediated to the wrapper as well as mediated to the host.

4.2. Label Models

The labeling and unlabeling of entities when mediating between JSFlow and V8 rely on label models which provide an abstract view of the computation of the mediated V8 values. In the experiments performed in this paper, we use a simple label model, that allows parts of the execution environment to be marked as sources or sinks. Reading from a source labels the value with the access path of the source and writing to a sink causes the label of the written value to be collected for analysis. As an example, reading the V8 runtime property `navigator.userAgent` would label the resulting value with the label `<global.navigator.userAgent>`, where `global` represents the global `window` object. To compute the baseline API imprint of the fingerprinting libraries, we use a model that labels every mediated part of the V8 API to record which parts of the API that are accessed by the libraries.

Over and under labeling. Observable IFC is in theory subject to both over and under labeling [51] but performs well on the actual code currently found in the wild. With respect to this work, the label model described above risks under labeling, since it does not track flows that go via the extended environment. For instance, writing a labeled value to the *Document Object Model (DOM)* would force unlabeling to occur, and the labels would be lost. While this may be an issue for a more practical implementation,

providing such a model for the full execution environment of a browser is not within the scope of this work. We refer the reader to [67] for an insight into the complexity of creating such models.

We encountered two examples of this issue in our practical experiments: two of the analyzed pages used mediated versions of `btoa` to base-64 encode the gathered information. This caused the labels to be lost before reaching the network sink. In those examples, we remedied the loss of precision by implementing `btoa`, but it points to the issue of losing labels when using functions that were automatically mediated from the V8 execution environment. Although this is not an issue in our initial experiments, finding better-suited label models for standard API interaction that more precisely track flows of information via the mediated API functionality is an attractive goal for future explorations.

4.3. Extending Chromium

Modifying and maintaining modifications on a commercial product like Chromium requires a lot of work. The updates are frequent and the changes are often major, potentially requiring a lot of effort to cope with. For this reason, we pick Chromium 78.0.3904.70[2] and try to keep the modifications to a minimum.

To be able to inject JSFlow we focus the modifications to the point in Chromium where the script source code is transferred from the rendering engine Blink to V8 as a string for execution. There, the following functionality was inserted.

- If the intercepted script is the first script to execute in the context, then JSFlow is first injected followed by the injection of the JSFlow runtime containing polyfills and other supporting libraries.
- When JSFlow has been injected the script is rewritten to contain a call to the main execution method of JSFlow passing the original source code as an argument.

The injection works in the same way for inline scripts, scripts fetched via a URL, or retrieved by other means. At the point of injection, Chromium has already extracted the script source into a string. This way, all scripts on a page are executed via JSFlow regardless of if they originate from an inline script tag, a script tag using a URL, or an event handler.

4.4. Protecting JSFlow

Scripts have the capability of modifying the V8 execution environment by, e.g., overwriting standard library implementations. Since JSFlow itself runs in the same V8 execution environment as a web page and is mediating to and from the execution environment, there is a need to protect the integrity of JSFlow. Further, JSFlow is implemented by using (parts of) the standard ES6+ execution environment, so modifications by scripts may unintentionally break JSFlow in the presence of uncontrolled mediation. To defend against this, JSFlow must run the scripts defensively to protect key parts of the environment from being affected by script execution. To this end, the standard execution environment of JSFlow does not perform any mediation, meaning scripts are unable to modify parts of the V8 execution environment also implemented by JSFlow. The parts of the V8 execution environment not implemented in the JSFlow execution environment are all mediated via the JSFlow global window object. This object is a hybrid between a JSFlow global object and a wrapper, providing protection from tampering by hiding sensitive parts of the execution environment. The parts of the execution environment that are either defined by JSFlow via this global window object or that are part of the hidden environment will not be mediated, with the hidden environment being implemented by polyfills. The remaining, mediated part of the global object follows the read-once-write-always mediation provided by the JSFlow wrappers.

2. This was the stable version when developing the patch.

4.5. Collecting Labels

Every value that is going through JSFlow is given a label by the label model, and when two values are combined, the corresponding label is computed as the least upper bound of the labels. In our setting, since labels are the access paths of the information used to create the values, the least upper bound corresponds to the union of the paths. Take, for example, the following code snippet.

```
let a = navigator.userAgent;
let b = navigator.language;
let c = a + ' ' + b;
```

The value `a` will be labeled `<navigator.userAgent>` and the value `b` will be labeled `<navigator.language>`. As the value `c` is the aggregation of values `a` and `b` it will be labeled with both sources forming the label `<navigator.userAgent, navigator.language>`.

In order to analyze the aggregated labels, JSFlow was extended with the ability to store the labels seen during execution on a script basis. In practice this is done on label creation; whenever the least upper bound is computed in JSFlow, the computed label will be stored internally in a map that maps script origins to label sets. This allows us to detect whether any values that may correspond to a fingerprint were created by the script. In addition, in order to be able to detect fingerprinting, we are interested in the flow from the API imprint to the network sinks. To track this we also store an accumulated label per API endpoint. Every time a JSFlow value is unlabeled to be passed into the V8 runtime (by an assignment or function call), we add the removed label to a map that maps API endpoints to labels. During the execution of a page, JSFlow regularly writes the accumulated labels via a function on the global object if such a function exists. This way automated tools, like crawlers, are able to collect labels from JSFlow by providing the extraction function. In our case, we use a Puppeteer-based [25] crawler that stores the collected labels to disk for later analysis.

5. Empirical Study

In order to validate our approach, we have conducted an empirical study by crawling web pages belonging to

one of two different categories: non-fingerprinting and fingerprinting web pages. The fingerprinting web pages are divided into three classes:

- (bot detection) web pages that perform bot detection,
- (authentication) web pages that use some form of fingerprinting as part of their authentication process, and
- (alexa) web pages in Alexa top 100,000 that perform fingerprinting that is not part of bot detection or authentication.

A total of 30 web pages were selected: 20 fingerprinting and 10 non-fingerprinting. Of the 20 fingerprinting pages, 5 are in the bot detection class, 5 in the authentication class, and 10 in the alexa class. Depending on the category and class different rationals were used in the selection.

Non-fingerprinting. To find pages in the non-fingerprinting category that contain interesting behavior such as analytic scripts, polyfills, or ads the following method was used. First, an initial selection of pages with potentially interesting behavior was created based on popularity and type of content. The assumption was that popularity is the result of a conscious effort and that popular pages are likely to contain both analytic scripts, polyfills (to reach a wide audience), and ads. In this category, we find, e.g., major news outlets. Second, we used the Brave browser to filter the selection based on the presence of tracking and analytic scripts as well as ads. The process was manual and based on visiting the pages using Brave while discarding all pages that Brave did not classify as containing scripts in either of those categories. For the pages indicated to contain interesting scripts, we further verified that the scripts were used by (and not only present in) the web pages by verifying that the scripts were executed when visiting the pages with script blocking turned off. Third, to ensure the web pages were free from fingerprinting the method to find fingerprinting pages described below was used in addition to the information given by Brave.

Authentication. To find pages that incorporate fingerprinting in their authentication process the following method was used. First, an initial selection of major bank pages was created based on the assumption that banks both include authentication and have a vested interest in protecting their users. Second, the selection was filtered by searching for the presence of known fingerprinting sources such as calls to `toDataURL` or calls to `navigator.plugins` by analyzing the scripts as strings while parsed in V8. Third, for the remaining candidates, we manually analyzed the scripts of the page by visiting the page with the debugger active to identify pages that used fingerprinting as part of the authentication process. In this step, we set breakpoints to ensure the scripts containing the potential fingerprinting were indeed executed and not only loaded.

Bot detection. To find pages that contain bot-detection the following method was used. First, a collection of candidate pages was created based on known customers of bot protection, booking pages, and pages mentioned by Jonker et al. in their paper on the detection of bot detection [55]. Second, the candidate pages were filtered using a specialized crawler that visits web pages, waits for 30 seconds, and then takes a screenshot. If the collected screenshot showed a (re)CAPTCHA or that access was denied, the web page was deemed to perform bot detection. Third, the remaining pages were manually analyzed by visiting the page with the debugger active, again setting breakpoints to ensure suspicious scripts were executed, retaining the pages that used fingerprinting as part of the bot-detection process.

Alexa. To find web pages that perform fingerprinting which do not match any of the previous categories, we used two different approaches. First, we used information from a crawl of Alexa top 100,000 performed with OpenWPM, which recorded blocked fingerprinting resources based on the Disconnect list [9]. We also visited pages in the Alexa top 1,000 and did the same analysis as for authentication pages. Based on these results, we visited a random set of web pages which had fingerprinting resources and ensured these resources were executed.

5.1. Experiment Setup

To visit the different web pages, we used a crawler implemented with Puppeteer [25] to control EssentialFP. In order to decrease the overhead of collecting the labels, we only collected labels that were part of the baseline API imprint. Using the baseline, each of the selected pages was visited by the crawler for up to 12 hours. This is to ensure that the fingerprinting script was not prevented from running due to JSFlow performance issues. As an example, fingerprinting scripts usually compound the values into a hash, and executing FingerprintJS alone with all flags active would take around 45 minutes with JSFlow. These performance issues come from JSFlow being an information-flow aware deep-embedding of ECMA-262 v5. When executing, JSFlow performs every execution step mandated by ECMA-262 v5 without any optimization. The execution speed of JSFlow does not reflect on the feasibility of using dynamic IFC but it limits the number of pages that can be analyzed with the current setup. Indeed, only around 5% of the execution time is spent on handling the security labels.

5.2. API Endpoints

Each of the selected web pages was visited while recording created and exfiltrated labels from the baseline API imprint. For each page and each API endpoint, every label reaching the endpoint was collected in addition to collecting all created labels for each page and script. This allows us to infer what parts of the API imprint did flow to network sinks as well as if the application accumulated fingerprinting information internally.

To define potential network sinks we analyzed the used API endpoints to identify uses that could trigger a network or storage request (which would or could be used to send the information later). For this experiment, the identified network sinks were:

- `navigator.sendBeacon`
- `XMLHttpRequest`
- `fetch`

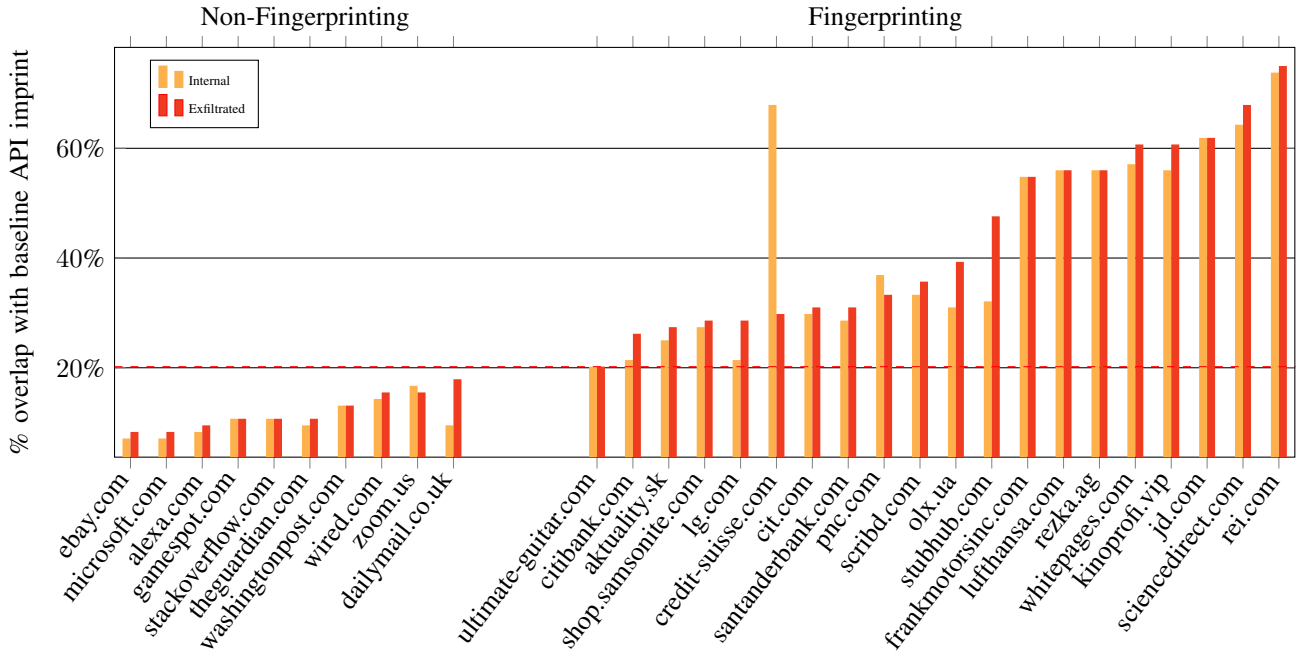


Figure 1: Breakdown of the maximum overlap for a script for each web page against the baseline API imprint consisting of the libraries FingerprintJS, ImprintJS, and ClientJS. *Internal* labels are features accessed by the web page code, and *Exfiltrated* are labels reaching a network sink. The web pages are sorted based on the exfiltrated label within the two categories (Non-fingerprinting and Fingerprinting). The y-axis is the percentage overlap for a web page with the baseline API imprint. The dashed red line is the lowest Fingerprinting overlap.

- `window.postMessage`
- setting `src` attributes of HTML elements such as images and scripts
- setting attributes on `HTMLFormElement`
- `document.cookie`

5.3. Analysis

The complete overlap for the sinks against the baseline API imprint for each web page can be found in Figure 1 and the exfiltration method for each web page can be seen in Table 1.

A key takeaway is that our results confirm our intuition: pages that access a wide surface of sensitive APIs and send consolidated information to the network represent the essence of fingerprinting and are clearly marked as such by their high overlaps with the baseline. Further, the majority of the visited web pages send sensitive information via network sinks, as can be seen in Table 1. This is an indication that (partial) compounded data is being sent to an external server. As expected, this occurs on pages belonging to both categories. The difference is the amount of sensitive information being transmitted. This is a strong indicator that it does not suffice to look at the API imprint of the application and whether the application uses the network or not. To detect fingerprinting we must track what information reaches the network sink.

We can also see that the majority of the web pages have close to equivalent internal fingerprints and labels that reach the network sink. Although our experiments over-approximate the labels reaching the network sink as the labels are compounded based on the API used and not the source of the request, it still indicates that current fingerprinting scripts accumulate and compute a fingerprint before transmitting it. As we do not see a much smaller overlap of internal labels we see no evidence of piece by piece fingerprinting, where the fingerprint is being sent gradually to an external server. This indicates it may be enough to only look at the internally created fingerprints, but a larger study must be conducted to evaluate this.

The only potential local use of the fingerprint is creditsuisse.com, as we can see a large drop between the overlap of the largest internally created label and the exfiltrated label. After manually analyzing the source code of creditsuisse.com it is clear the largest internal label comes from performing fingerprinting, which writes the result to a global variable called `fp2murmur`. This global variable is never used, which may indicate the result from that specific fingerprinting method is not used. However, we could also see that several fingerprinting attributes are being written gradually to a form element, which is the 29.8% overlap which can be seen in Figure 1.

When looking at the two categories, we can see there is a potential cut-off that allows us to distinguish between non-fingerprinting and fingerprinting. A larger than 20% overlap with the full baseline indicates the presence of fingerprinting. Indeed, the *maximum* matching overlap for the non-fingerprinting category came from dailymail.co.uk, with an overlap of 17.9%. This can be compared to the *minimum* matching overlap for the fingerprinting category, namely ultimate-guitar.com with 20.2%.

When looking at the individual classes in the fingerprinting category, the situation is more subtle. Pages within the authentication class have an overlap with the full baseline of between 26.2% and 33.3%, and the bot detection class has an overlap between 28.6% and 60.7%, compared to the alexa class which has five pages at or above 56%. This is not surprising since authentication and

TABLE 1: Table showing the sinks used by the scripts on the domains to exfiltrate the maximum overlap of (potential) fingerprint value. Only sinks that can be used to send data (e.g. `document.cookie`, network requests, etc.) is shown. The entries are sorted by the two categories.

Domain	Class	Sink
alexa.com	—	`Image.src`
dailymail.co.uk	—	`document.querySelectorAll.src`
ebay.com	—	`HTMLFormElement.setAttribute`
gamespot.com	—	`document.cookie`
microsoft.com	—	`navigator.sendBeacon`
stackoverflow.com	—	`navigator.sendBeacon`
theguardian.com	—	`Image.src`
washingtonpost.com	—	`navigator.sendBeacon`
wired.com	—	`document.createElement.src`
zoom.us	—	`HTMLFormElement.setAttribute`
aktuality.sk	Alexa list	`document.createElement.src`, `XMLHttpRequest.open`
cit.com	Authentication	`HTMLFormElement.setAttribute`
citibank.com	Authentication	`XMLHttpRequest.send`
credit-suisse.com	Authentication	`HTMLFormElement.appendChild`
frankmotorsinc.com	Bot detection	`XMLHttpRequest.send`
jd.com	Alexa list	`XMLHttpRequest.send`
kinoprofi.vip	Alexa list	`HTMLFormElement.setAttribute`
lg.com	Alexa list	`HTMLFormElement.setAttribute`
lufthansa.com	Bot detection	`XMLHttpRequest.send`
olx.ua	Alexa list	`document.cookie`
pnc.com	Authentication	`document.createElement.src`
rei.com	Alexa list	`document.cookie`
rezka.ag	Alexa list	`HTMLFormElement.setAttribute`
santanderbank.com	Authentication	`XMLHttpRequest.send`
sciencedirect.com	Alexa list	`document.cookie`
scribd.com	Alexa list	`document.createElement.src`
shop.samsonite.com	Bot detection	`HTMLFormElement.setAttribute`
stubhub.com	Bot detection	`XMLHttpRequest.send`
ultimate-guitar.com	Alexa list	`document.createElement.src`, `navigator.sendBeacon`
whitepages.com	Bot detection	`XMLHttpRequest.send`

bot detection may have different goals compared to the fingerprinting in the alexa class.

We note that the overlap of the bot detection pages may have been higher if we tried to hide the fact we are a crawler by using stealth libraries for Puppeteer (e.g. puppeteer-extra-plugin-stealth [26]). We decided against using a stealth library as this can make properties used for fingerprinting non-accessible and with that, the amount of fingerprinting information is less.

Aside from the pages in the alexa class of the fingerprinting category using `document.cookie` as the exfiltration method (only one non-fingerprinting web page used it), there is no real distinction between the exfiltration methods between the two categories.

To further analyze the two categories, we have created heatmaps for the different features of FingerprintJS and ImprintJS. The heatmaps for the two categories are found in Figure 2 for FingerprintJS and in Appendix A for ImprintJS. As ImprintJS is not as updated as FingerprintJS there are some keys not found on web pages as they refer to outdated API calls. However, the heatmaps for ImprintJS entail the same story as FingerprintJS and we will therefore focus on FingerprintJS in this section.

When generating the heatmaps, we took the labels from the highest overlap with the baseline API imprint for each web page (i.e., the same overlap as in Figure 1). This set of labels is then cross-referenced against what labels each feature of the fingerprinting libraries would produce, to discover which keys could have been used when generating the set of labels. Each row in the heatmaps shows the conditional probabilities of the features of each column given the feature of the row. The probabilities are interpreted as a gradient between yellow (0% probability) and red (100% probability). Thus, the diagonal shows which features were used for the largest overlaps by the pages in the category.

Looking at the heatmaps we can see that pages from the non-fingerprinting category (Figure 2a) are not that intrusive when it comes to data collection compared to the fingerprinting category (Figure 2b). One could argue non-fingerprinting web pages should have no overlap with the baseline API imprint, but we can see in Figure 2a that the collected features are mainly screen information and general browser information. Although these features are also used by fingerprinting web pages, more features are usually used when conducting fingerprinting.

We have also added heatmaps for each class in the fingerprinting category in Appendix B for FingerprintJS and Appendix C for ImprintJS. These heatmaps show an increased intensity in the use of fingerprinting features from the authentication class and the bot detection class. The alexa class clearly shows that the more general, traditional fingerprinting is the most intrusive with many fingerprinting features being used. This supports the success in detecting fingerprinting by looking at the overlap between the baseline API imprint and the exfiltrated information.

When looking at the heatmap for the fingerprinting category for FingerprintJS, we can see the use of `enumerateDevices`, which probes the list of all available media input and output devices. Interestingly enough, this almost exclusively comes from the bot detection class. It is worth pointing out that `enumerateDevices` is disabled by default in FingerprintJS, which may explain why it is not used in the alexa class. Similarly, the `webdriver` feature, which checks for the property

`navigator.webdriver`, is overrepresented in the bot detection class. Although it can be faked, this property indicates if the user agent is controlled by an automatic tool such as Puppeteer or Selenium, making it an easy check to detect potential bots, given that they are not trying to hide their presence.

6. Discussion

While our initial experiments show that it is possible to detect fingerprinting by tracking how information flows from a baseline API imprint to network sinks, our work opens up a few interesting avenues of future work.

6.1. Fingerprinting Metrics

Our solution compares the overlap between the baseline API imprint and the script API access pattern to detect fingerprinting. The results show that this approach is already effective. For the pages that make up our empirical study, it is possible to use this to perform a precise classification without false positives or false negatives. For a more extensive study, however, more precise metrics are likely to be needed. For instance, a source of potential false negatives would be web pages which only use a specific fingerprinting method, such as canvas fingerprinting. In the experiments, we manually verified each of the non-fingerprinting web pages to ensure this was not the case, but for these situations, another fingerprinting metric may be more suitable.

Weighted overlap. The unweighted overlap used in this paper does not distinguish between common and uncommon features. For example, the API pattern of canvas fingerprinting is implicitly assigned the same importance as the API pattern of querying the user agent or the screen width of the browser, while the canvas fingerprinting is arguably more indicative of fingerprinting than probing the user agent or screen width. Instead of implicitly giving all features the same importance weighted overlap assigns weights to each source and compares the weighted sum. One interesting starting point to the challenge of deciding on the relative weights could be to use the entropy reported by Panopticlick [22] to generate the weights for each API pattern in the fingerprinting library.

Conditional overlap. The weighted overlap assigns more importance to features that are more indicative of fingerprinting, but does not take that some combinations of features may be rarer than others, as indicated by our heatmaps, into account. Thus, such combinations should probably be given more weight than the sum of their parts. The conditional probability computed to generate the heatmaps could be a good starting point for finding clusters that identify the various categories.

6.2. Further Aid Anti-Fingerprinting

If a more effective version of EssentialFP is implemented directly into V8, which is a huge undertaking due to V8 being a fast-moving target, there are two natural uses. On the one hand, it could be used to help generate filter lists that help users block unwanted content. This could remove the currently heavy manual burden of creating and maintaining these filter lists. On the other hand, it could be used to collect more labels than just the API imprint, and with that find new potential fingerprinting vectors when they arise.

7. Related Work

Much work has been done, regarding both conducting and combating device fingerprinting. This section aims to provide an overview of combating fingerprinting. For a full description, we refer the reader to a timely survey by Laperdrix et al. [59].

Englehardt and Narayanan developed OpenWPM [21], [46], a web privacy measurement framework. OpenWPM instruments the access to the JavaScript API and links every access to the corresponding script. This is then stored for offline analysis, and Englehardt and Narayanan found new fingerprinting vectors by crawling with OpenWPM on the Alexa top 1,000,000. However, different from OpenWPM, EssentialFP does not implicitly assume that access equates to exfiltration and can instead focus purely on the flows that reach potential network sinks.

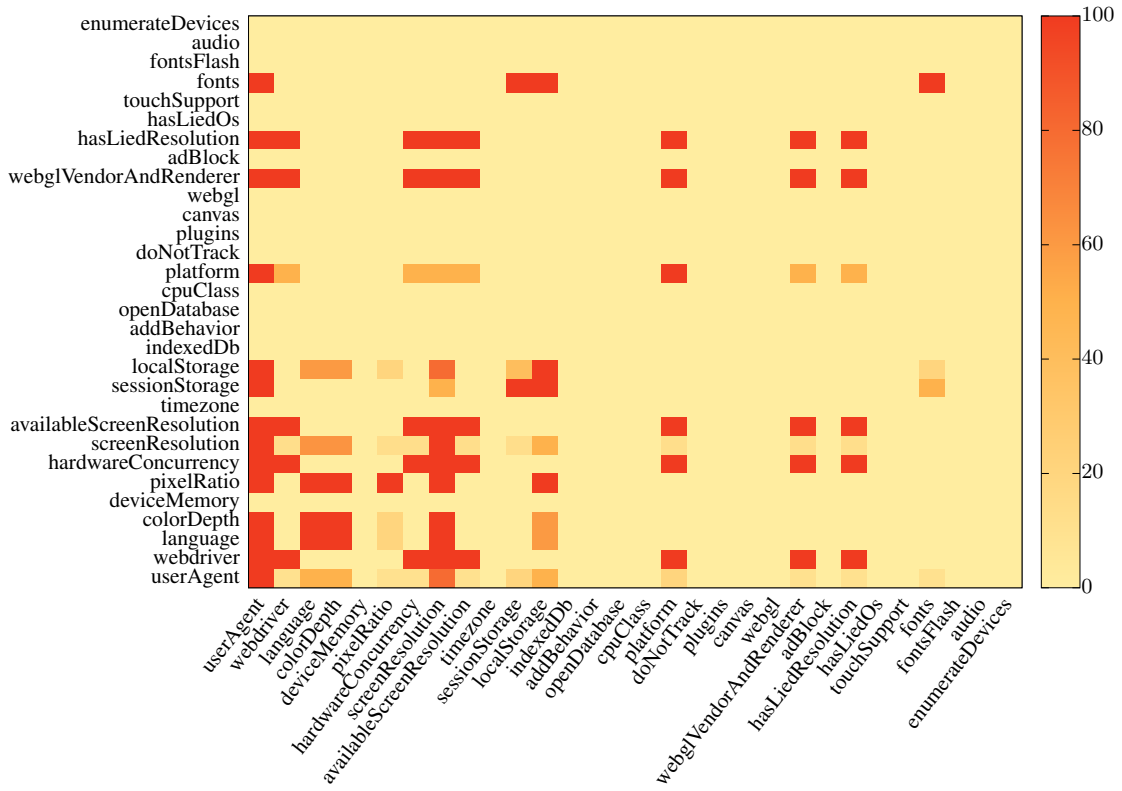
A promising approach to help detect browser fingerprinting is machine learning. Iqbal et al. presented FP-INSPECTOR [53] and detected fingerprinting with a 99.9% accuracy and 26% more fingerprinting scripts than manually designed heuristics. Rizzo et al. combine analysis of JavaScript code with machine learning to detect web fingerprinters [65], achieving 94% accuracy and finding more than 840 fingerprinting services, 695 of which were unknown to popular tracker blockers. By taking advantage that fingerprinting scripts have similar API access patterns, Bird et al. presented a semi-supervised machine learning approach to detect fingerprinting [41], identifying $\geq$94.9% of resources that current heuristic techniques identified. This is an approach that can be combined with EssentialFP and potentially strengthen the approach by only focusing on observable values that reach a network sink.

Another approach is to use browser extensions to randomize, e.g., the user agent, but as Nikiforakis et al. showed [64], this can create inconsistencies between the user agent and other publicly available APIs (e.g. `navigator.platform`). Vastel et al. developed FP-SCANNER [71], a test-suite that explores browser fingerprint inconsistencies to detect potential alterations done by fingerprinting countermeasures tools. FP-SCANNER could not only find these inconsistencies but also reveal the original values, which in turn could be exploited by fingerprinters to more accurately target browsers with fingerprinting countermeasures.

Both Laperdrix et al. [60] and Gómez-Boix et al. [48] proposed virtualization and modular architectures to randomly assemble a coherent set of components whenever a user wanted to browse the web. This would break the linkability of fingerprints between sessions while not having any inconsistencies between attributes, but the user comfort may go down.

Besson et al. formalized a privacy enforcement based on a randomization defense using quantitative information-flow [39]. They synthesize a randomization mechanism that defines the configurations for each user and found that more efficient privacy enforcement often

(a) Non-fingerprinting



(b) Fingerprinting

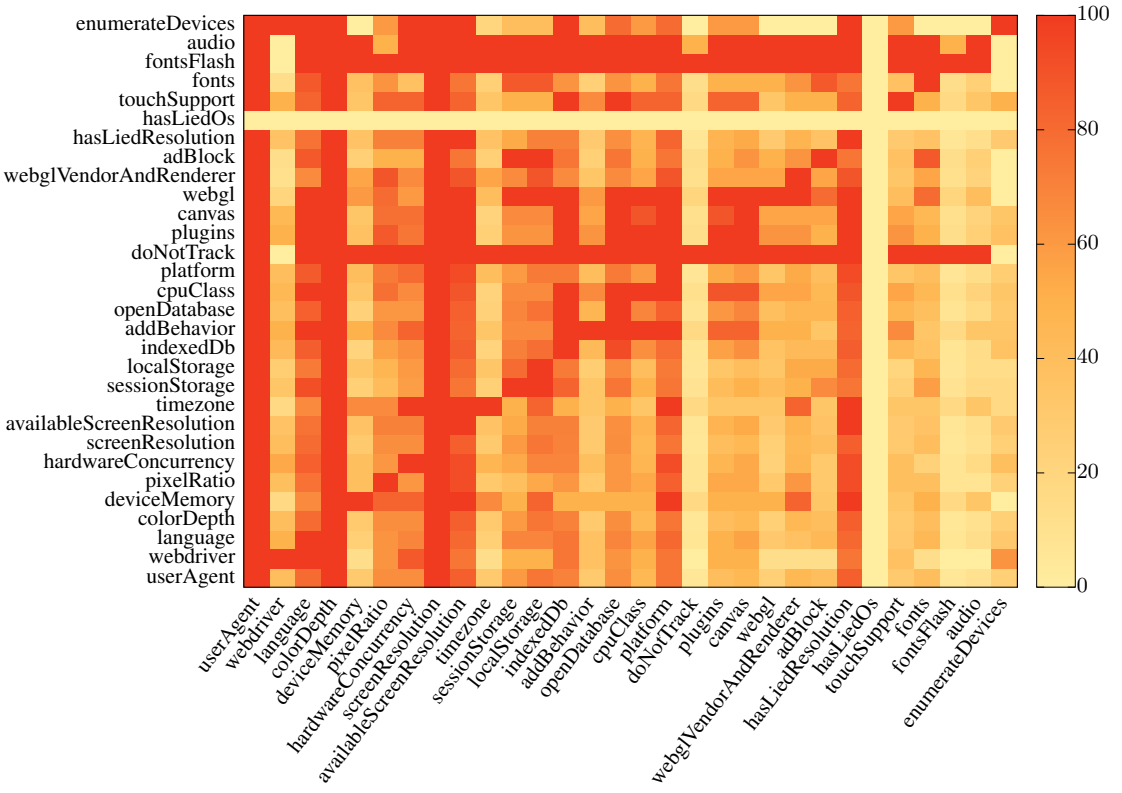


Figure 2: Heatmaps for the two categories when compared against FingerprintJS. For each row, the column identifies the conditional probability of the feature of the column given the feature of the row. The probability is interpreted as a gradient between yellow (0% probability) and red (100% probability).

leads to lower usability, i.e., users have to switch to other configurations often. Jang et al. used a rewriting-based IFC approach to detect four privacy-violating flows in web applications: cookie stealing, location hijacking, history sniffing, and behavior tracking, finding 46 cases of history sniffing in the Alexa top 50,000 [54]. Ferreira Torres et al. [69] proposed generating unique fingerprints to be used on each visited web page, making it more difficult for third parties to track the same user over multiple web pages. FaizKhademi et al. [47] proposed the detection of fingerprinting by monitoring and recording the activities by a web page from the time it started loading. Based on the recording, they were able to extract metrics related to fingerprinting methods to build a signature of the web page to distinguish normal web pages from fingerprinting web pages. If the web page is deemed to be fingerprinting, the access to the browser is limited e.g. by limiting the number of fonts allowed to be enumerated, by adding randomness to attribute values of the `navigator` and `screen` objects, and noise to canvas images that are generated.

Nikiforakis et al. [63] proposed using randomization policies, which are protection strategies that can be activated when certain criteria are met. Similarly, Laperdrix et al. [58] proposed adding randomness to some more complex parts of the DOM API: canvas, web audio API, and the order of JavaScript object properties.

Browser vendors are also implementing anti-fingerprinting measures. Brave [34] added randomness to mitigate canvas, WebGL, and AudioContext fingerprinting, adding to their already implemented fingerprinting protections [11]. Firefox introduced Enhanced Tracking Protection [17], which would allow third-party cookies to be blocked. This has later been expanded to also block all third-party requests to companies that are known to participate in fingerprinting [13]; a feature that is also found in Microsoft Edge [18]. Safari applies similar restrictions on cookies as Firefox, and also presents a simplified version of the system configuration to trackers, making more devices look identical [28]. The Tor browser aims to make all users look identical to resist fingerprinting [31]. Unfortunately, this means that as soon as a user maximizes the browser window or installs a plugin, their fingerprint will divert from the unified Tor browser fingerprint [57]. Similarly, as all Tor browsers aim to look identical, Khattak et al. showed they can be a target for blocking, showing 3.67% of the Alexa top 1,000 pages blocked access to Tor users [56]. Lastly, of the well-known browser vendors, Chrome has announced "The Privacy Sandbox" [30], where they are planning to combat fingerprinting by implementing a privacy budget [23].

8. Conclusion

We have presented EssentialFP, a principled approach to detecting fingerprinting on the web using observable information-flow control. Answering RQ1 on the essence of fingerprinting: EssentialFP identifies the essence of fingerprinting based on the pattern of (i) gathering information from a wide browser API surface (multiple browser-specific sources), and (ii) communicating the information to the network via a network sink. Using this pattern, EssentialFP can clearly distinguish fingerprinting from similar types of scripts such as analytical scripts and polyfills. For RQ2 on exposing fingerprinting: EssentialFP exposes the pattern from RQ1 by monitoring based on observable information flow tracking. To implement EssentialFP, we have leveraged, extended, and deployed JSFlow, a state-of-the-art information flow tracker for JavaScript, in a browser. We have demonstrated the effectiveness to spot fingerprinting on the web by EssentialFP by evaluating it on two categories of web pages: non-fingerprinting (pages that perform analytics, use polyfills, and show ads), and fingerprinting (which are divided into the classes of authentication, bot detection, and fingerprinting-enhanced pages from the Alexa list). Our results reveal different extent of fingerprinting in the web pages, ranging from no evidence of fingerprinting in the non-fingerprinting pages, to some evidence of fingerprinting in the authentication and bot detection pages, to full-blown evidence in the fingerprinting-enhanced pages from the Alexa list.

Acknowledgments. Thanks are due to Pete Snyder for the stimulating discussions on the topic of browser fingerprinting. This work was partially supported by the Swedish Foundation for Strategic Research (SSF) and the Swedish Research Council (VR).

References

[1] https://github.com/disconnectme/disconnect-tracking-protection/issues.

[2] https://forums.lanik.us/viewforum.php?f=64&sid=3d7d9fed66ba36b96c4b18f3142d0e43.

[3] https://github.com/easylist/easylist/issues.

[4] https://github.com/brave/brave-browser/issues/10000.

[5] Babel. https://babeljs.io/.

[6] ClientJS. https://clientjs.org/.

[7] Combating Fingerprinting with a Privacy Budget. https://github.com/bslassey/privacy-budget.

[8] core-js. https://www.npmjs.com/package/core-js.

[9] Disconnect. https://github.com/disconnectme/disconnect-tracking-protection.

[10] EssentialFP code and benchmarks. https://www.cse.chalmers.se/research/group/security/essential-fp.

[11] Fingerprinting Protections. https://github.com/brave/brave-browser/wiki/Fingerprinting-Protections.

[12] FingerprintJS. https://fingerprintjs.com/.

[13] Firefox 72 blocks third-party fingerprinting resources. https://blog.mozilla.org/security/2020/01/07/firefox-72-fingerprinting/.

[14] General Data Protection Regulation GDPR. https://gdpr-info.eu/.

[15] ImprintJS. https://github.com/mattbrailsford/imprintjs.

[16] Jsflow. http://www.jsflow.net/.

[17] Latest Firefox Rolls Out Enhanced Tracking Protection. https://blog.mozilla.org/blog/2018/10/23/latest-firefox-rolls-out-enhanced-tracking-protection/.

[18] Learn about tracking prevention in Microsoft Edge. https://support.microsoft.com/en-us/help/4533959/microsoft-edge-learn-about-tracking-prevention.

[19] Mitigating Browser Fingerprinting in Web Specifications. https://w3c.github.io/fingerprinting-guidance/.

[20] Modernizr. https://modernizr.com/.

[21] OpenWPM. https://github.com/mozilla/OpenWPM.

[22] Panopticlick. https://panopticlick.eff.org.

[23] Potential uses for the Privacy Sandbox. https://blog.chromium.org/2019/08/potential-uses-for-privacy-sandbox.html.

[24] Proxy. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy.

[25] Puppeteer. https://pptr.dev/.

[26] puppeteer-extra-plugin-stealth. https://github.com/berstend/puppeteer-extra/tree/master/packages/puppeteer-extra-plugin-stealth.

[27] regenerator-runtime. https://www.npmjs.com/package/regenerator-runtime.

[28] Safari Privacy Overview. https://www.apple.com/safari/docs/Safari_White_Paper_Nov_2019.pdf.

[29] Standard ECMA-262 6th Edition / June 2015. https://www.ecma-international.org/ecma-262/6.0/.

[30] The Privacy Sandbox. https://www.chromium.org/Home/chromium-privacy/privacy-sandbox.

[31] Tor. https://www.torproject.org/.

[32] Tracking Preference Expression (DNT). https://www.w3.org/TR/tracking-dnt/.

[33] Unsanctioned Web Tracking. https://w3ctag.github.io/unsanctioned-tracking/.

[34] What's Brave Done For My Privacy Lately? Episode #3: Fingerprint Randomization. https://brave.com/whats-brave-done-for-my-privacy-lately-episode3/.

[35] window-crypto. https://www.npmjs.com/package/window-crypto.

[36] G. Acar, C. Eubank, S. Englehardt, M. Juárez, A. Narayanan, and C. Díaz. The Web Never Forgets: Persistent Tracking Mechanisms in the Wild. In *CCS*, 2014.

[37] G. Acar, M. Juárez, N. Nikiforakis, C. Díaz, S. F. Gürses, F. Piessens, and B. Preneel. FPDetective: dusting the web for fingerprinters. In *CCS*, 2013.

[38] M. Balliu, D. Schoepe, and A. Sabelfeld. We Are Family: Relating Information-Flow Trackers. In *ESORICS*, 2017.

[39] F. Besson, N. Bielova, and T. P. Jensen. Browser Randomisation against Fingerprinting: A Quantitative Information Flow Approach. In *NordSec*, 2014.

[40] A. Bichhawat, V. Rajani, D. Garg, and C. Hammer. Information Flow Control in WebKit's JavaScript Bytecode. In *POST*, 2014.

[41] S. Bird, V. Mishra, S. Englehardt, R. Willoughby, D. Zeber, W. Rudametkin, and M. Lopatka. Actions speak louder than words: Semi-supervised learning for browser fingerprinting detection. 2020.

[42] Y. Cao, S. Li, and E. Wijmans. (Cross-)Browser Fingerprinting via OS and Hardware Level Features. In *NDSS*, 2017.

[43] C. Cimpanu. Ad Network Uses DGA Algorithm to Bypass Ad Blockers and Deploy In-Browser Miners. https://www.bleepingcomputer.com/news/security/ad-network-uses-dga-algorithm-to-bypass-ad-blockers-and-deploy-in-browser-miners/.

[44] D. E. Denning and P. J. Denning. Certification of Programs for Secure Information Flow. 1977.

[45] D. Devriese and F. Piessens. Noninterference through Secure Multi-execution. In *S&P*, 2010.

[46] S. Englehardt and A. Narayanan. Online Tracking: A 1-million-site Measurement and Analysis. In *CCS*, 2016.

[47] A. FaizKhademi, M. Zulkernine, and K. Weldemariam. FPGuard: Detection and Prevention of Browser Fingerprinting. In *DBSec*, 2015.

[48] A. Gómez-Boix, D. Frey, Y. Bromberg, and B. Baudry. A Collaborative Strategy for Mitigating Tracking through Browser Fingerprinting. In *MTD@CCS*, 2019.

[49] A. Gómez-Boix, P. Laperdrix, and B. Baudry. Hiding in the Crowd: an Analysis of the Effectiveness of Browser Fingerprinting at Large Scale. In *WWW*, 2018.

[50] D. Hedin, L. Bello, and A. Sabelfeld. Information-flow security for JavaScript and its APIs. 2016.

[51] D. Hedin and A. Sabelfeld. A Perspective on Information-Flow Control. In *Software Safety and Security*. 2012.

[52] D. Hedin and A. Sabelfeld. Information-Flow Security for a Core of JavaScript. In *CSF*, 2012.

[53] U. Iqbal, S. Englehardt, and Z. Shafiq. Fingerprinting the Fingerprinters: Learning to Detect Browser Fingerprinting Behaviors. In *S&P*, 2021.

[54] D. Jang, R. Jhala, S. Lerner, and H. Shacham. An empirical study of privacy-violating information flows in JavaScript web applications. In *CCS*, 2010.

[55] H. Jonker, B. Krumnow, and G. Vlot. Fingerprint Surface-Based Detection of Web Bot Detectors. In *ESORICS*, 2019.

[56] S. Khattak, D. Fifield, S. Afroz, M. Javed, S. Sundaresan, D. McCoy, V. Paxson, and S. J. Murdoch. Do You See What I See? Differential Treatment of Anonymous Users. In *NDSS*, 2016.

[57] P. Laperdrix. Browser Fingerprinting: An Introduction and the Challenges Ahead. https://blog.torproject.org/browser-fingerprinting-introduction-and-challenges-ahead.

[58] P. Laperdrix, B. Baudry, and V. Mishra. FPRandom: Randomizing Core Browser Objects to Break Advanced Device Fingerprinting Techniques. In *ESSoS*, 2017.

[59] P. Laperdrix, N. Bielova, B. Baudry, and G. Avoine. Browser Fingerprinting: A Survey. 2020.

[60] P. Laperdrix, W. Rudametkin, and B. Baudry. Mitigating Browser Fingerprint Tracking: Multi-level Reconfiguration and Diversification. In *SEAMS*, 2015.

[61] M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, 2006.

[62] K. Mowery and H. Shacham. Pixel Perfect: Fingerprinting Canvas in HTML5. In *W2SP*, 2012.

[63] N. Nikiforakis, W. Joosen, and B. Livshits. PriVaricator: Deceiving Fingerprinters with Little White Lies. In *WWW*, 2015.

[64] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. Cookieless Monster: Exploring the Ecosystem of Web-Based Device Fingerprinting. In *S&P*, 2013.

[65] V. Rizzo, S. Traverso, and M. Mellia. Unveiling Web Fingerprinting in the Wild Via Code Mining and Machine Learning. In *PETS*, 2020.

[66] A. Sabelfeld and A. C. Myers. Language-based information-flow security. 2003.

[67] A. Sjösten, D. Hedin, and A. Sabelfeld. Information Flow Tracking for Side-Effectful Libraries. In *FORTE*, 2018.

[68] C. Staicu, D. Schoepe, M. Balliu, M. Pradel, and A. Sabelfeld. An Empirical Study of Information Flows in Real-World JavaScript. In *PLAS*, 2019.

[69] C. F. Torres, H. L. Jonker, and S. Mauw. FP-Block: Usable Web Privacy by Controlling Browser Fingerprinting. In *ESORICS*, 2015.

[70] E. Trickel, O. Starov, A. Kapravelos, N. Nikiforakis, and A. Doupé. Everyone is Different: Client-side Diversification for Defending Against Extension Fingerprinting. In *USENIX Security*, 2019.

[71] A. Vastel, P. Laperdrix, W. Rudametkin, and R. Rouvoy. FP-Scanner: The Privacy Implications of Browser Fingerprint Inconsistencies. In *USENIX Security*, 2018.

[72] A. Vastel, P. Laperdrix, W. Rudametkin, and R. Rouvoy. FP-STALKER: Tracking Browser Fingerprint Evolutions. In *S&P*, 2018.

[73] T. Yen, Y. Xie, F. Yu, R. P. Yu, and M. Abadi. Host Fingerprinting and Tracking on the Web: Privacy and Security Implications. In *NDSS*, 2012.

Appendix A.
Heatmaps of the Two Categories Against ImprintJS

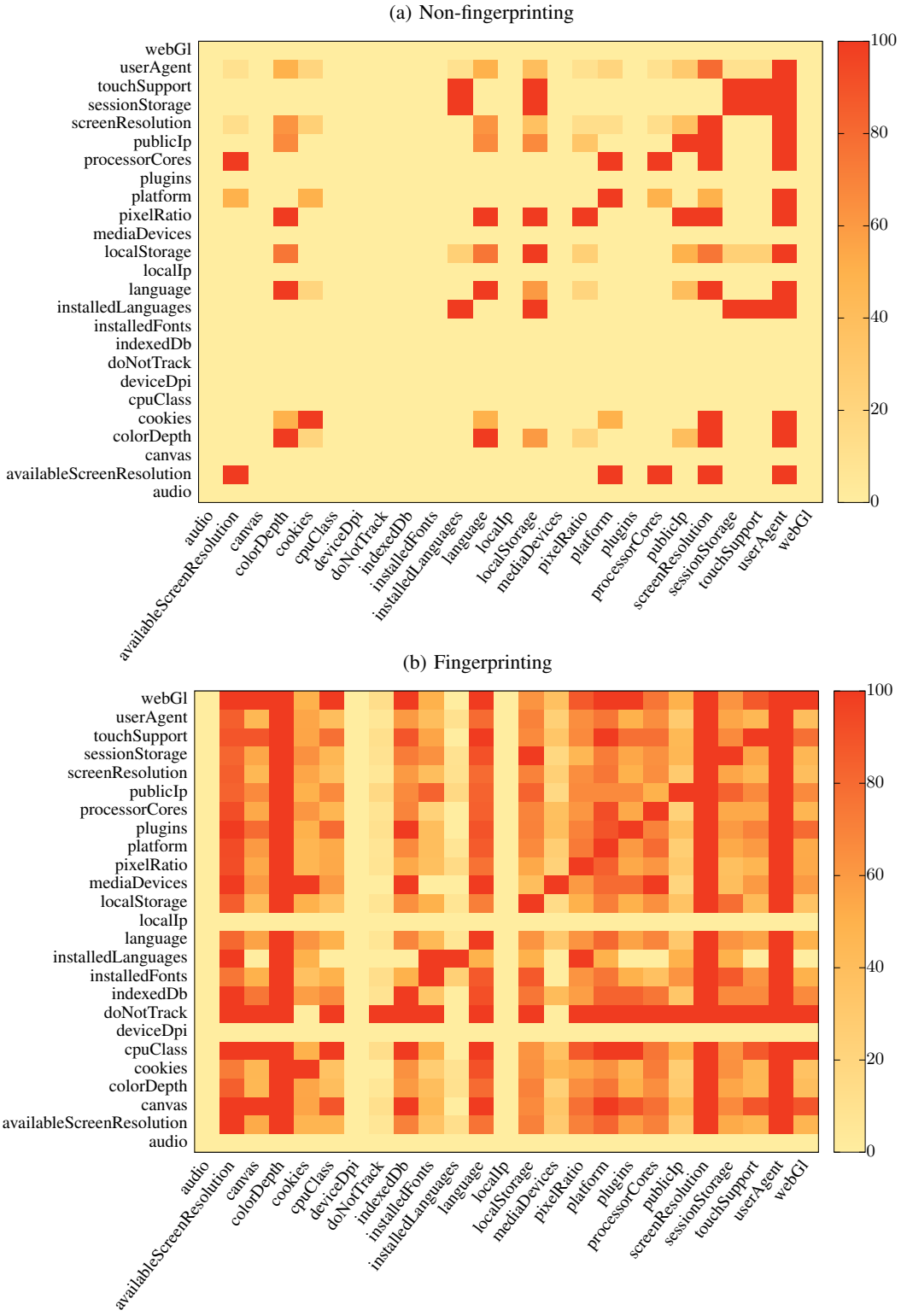
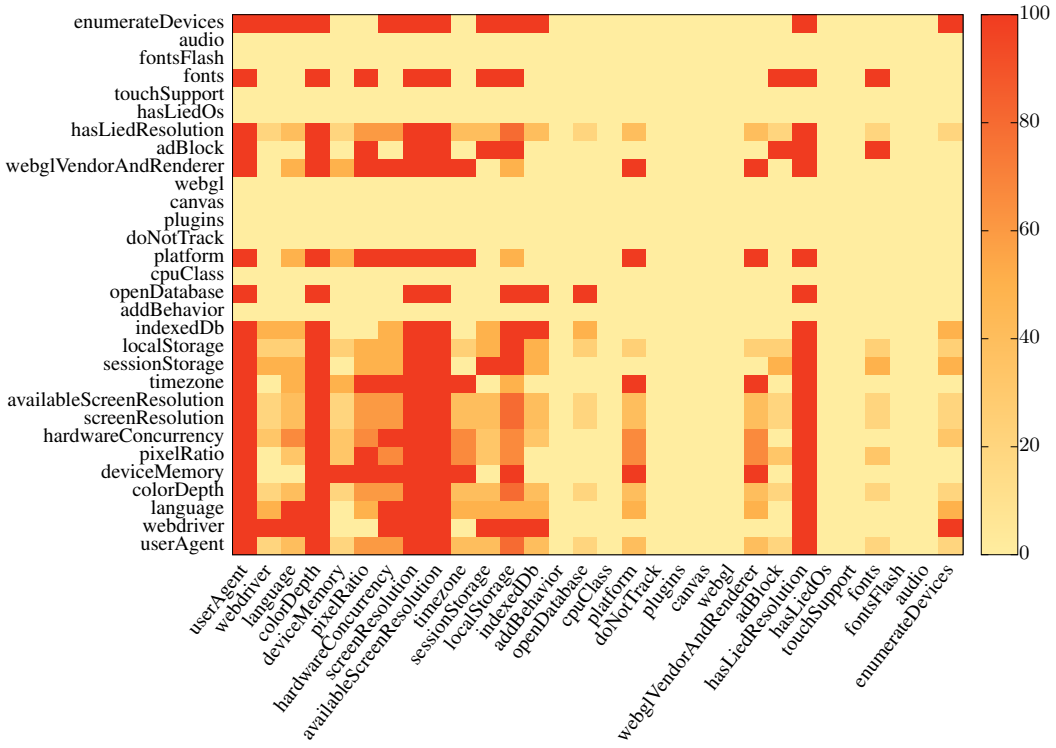


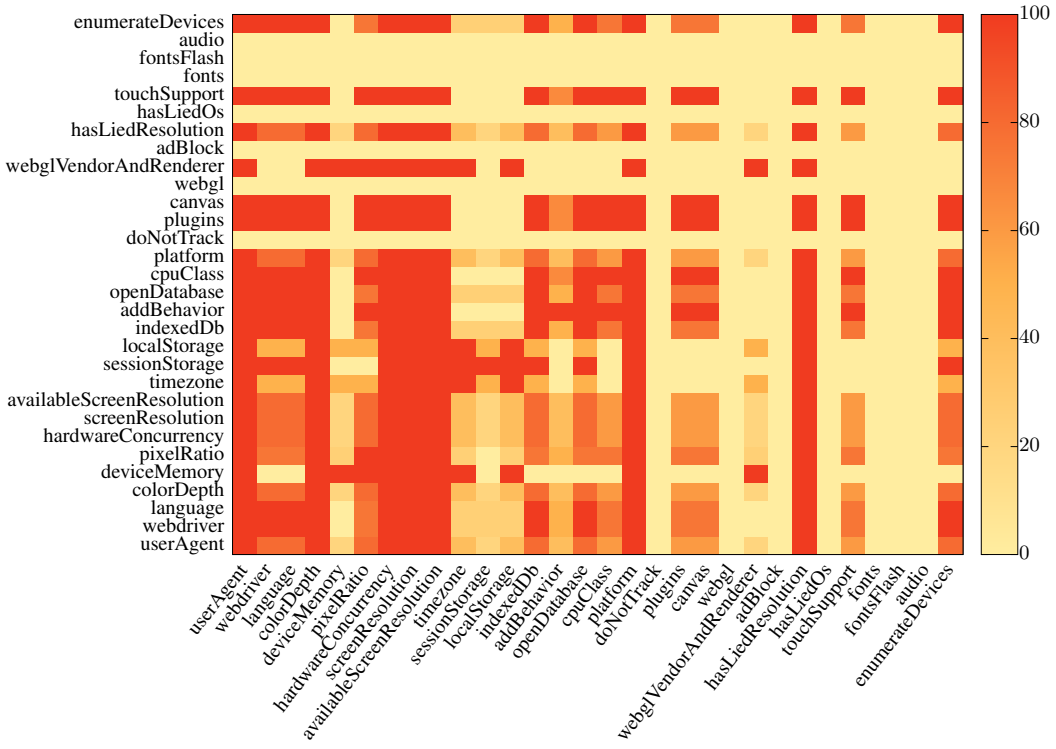
Figure 3: Heatmaps for the two categories when compared against ImprintJS. For each row, the column identifies the conditional probability of the feature of the column given the feature of the row. The probability is interpreted as a gradient between yellow (0% probability) and red (100% probability).

Appendix B.
Heatmaps of Fingerprinting Classes Against FingerprintJS

(a) Authentication



(b) Bot detection



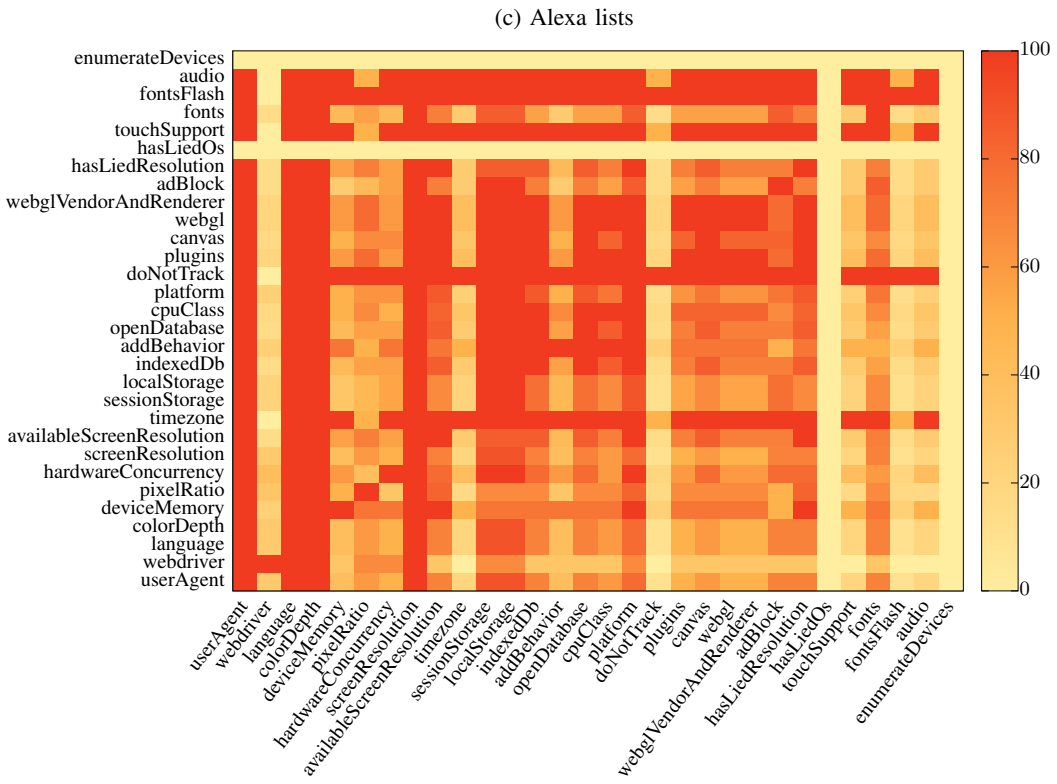
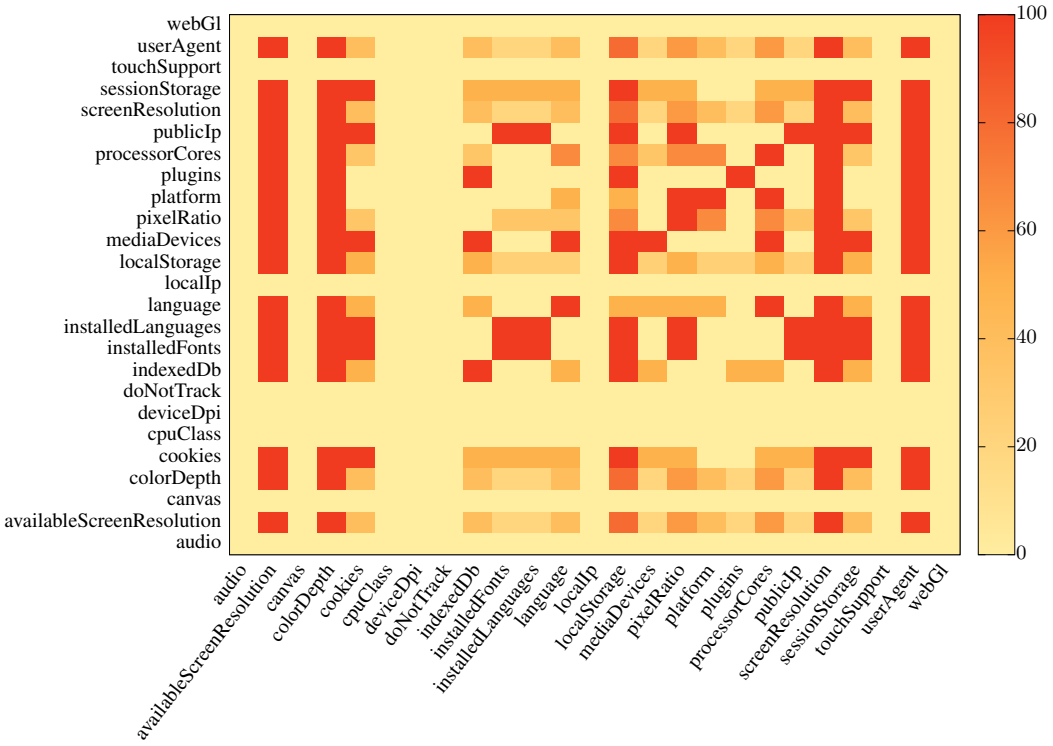


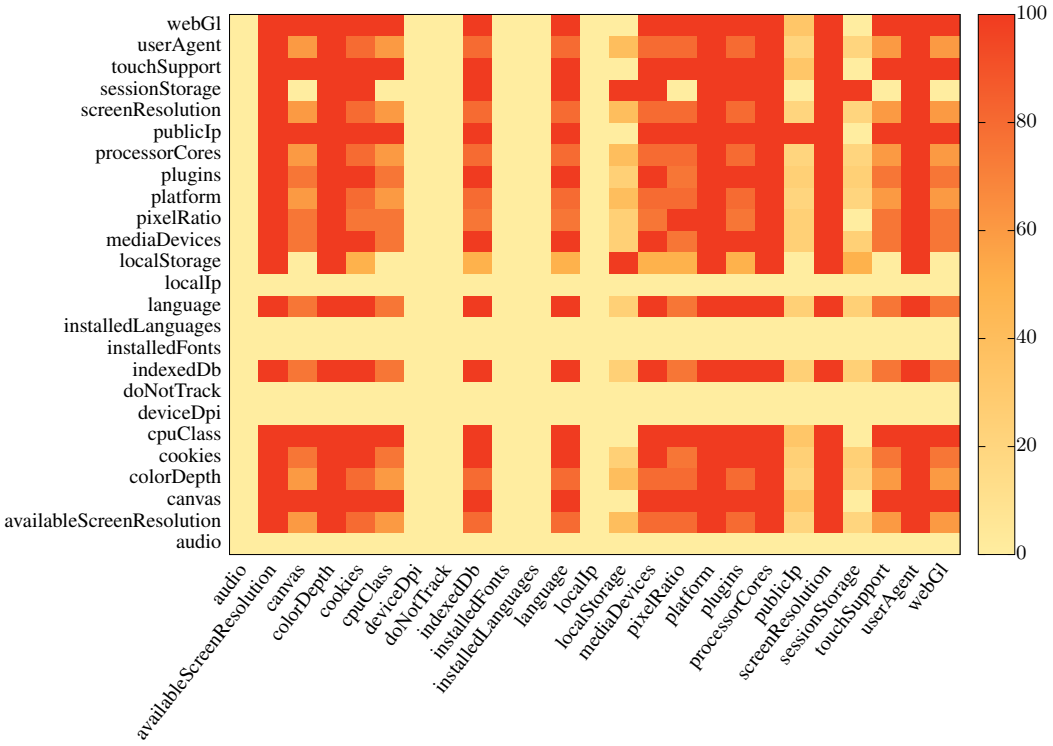
Figure 4: Heatmaps for the different fingerprinting classes when compared against FingerprintJS. For each row, the column identifies the conditional probability of the feature of the column given the feature of the row. The probability is interpreted as a gradient between yellow (0% probability) and red (100% probability).

Appendix C.
Heatmaps of Fingerprinting Classes Against ImprintJS

(a) Authentication



(b) Bot detection



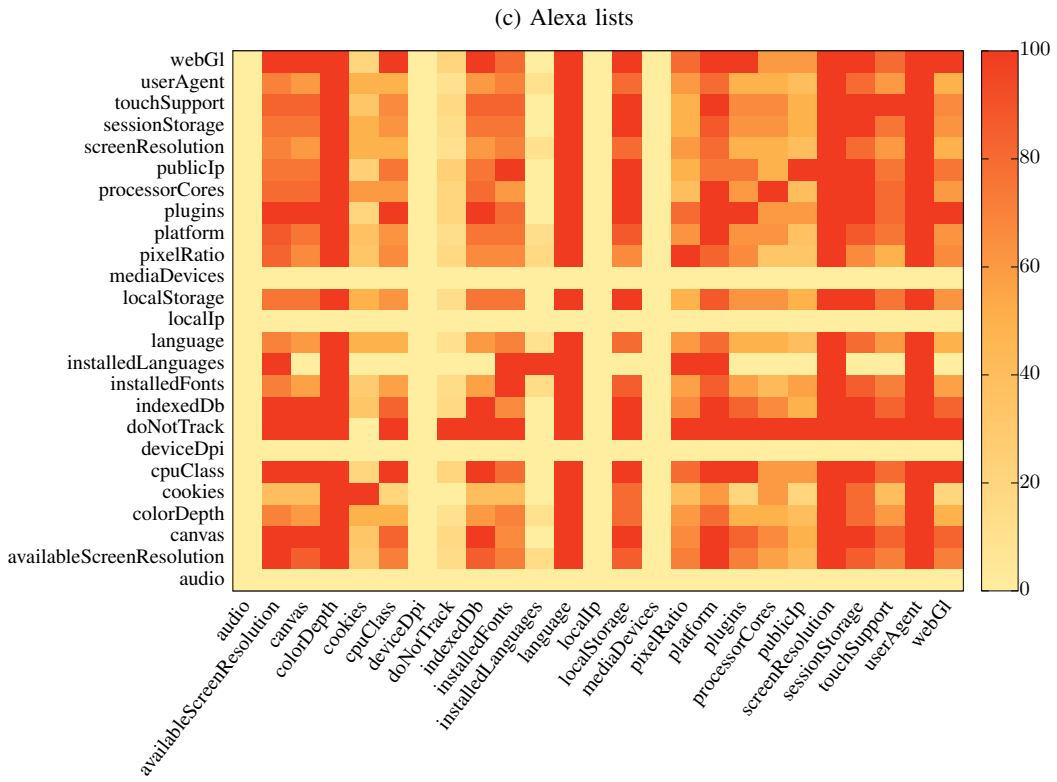


Figure 5: Heatmaps for the different fingerprinting classes when compared against ImprintJS. For each row, the column identifies the conditional probability of the feature of the column given the feature of the row. The probability is interpreted as a gradient between yellow (0% probability) and red (100% probability).