Nathan Reitinger* and Michelle L. Mazurek

ML-CB: Machine Learning Canvas Block

Abstract: With the aim of increasing online privacy, we present a novel, machine-learning based approach to blocking one of the three main ways website visitors are tracked online—canvas fingerprinting. Because the act of canvas fingerprinting uses, at its core, a JavaScript program, and because many of these programs are reused across the web, we are able to fit several machine learning models around a semantic representation of a potentially offending program, achieving accurate and robust classifiers. Our supervised learning approach is trained on a dataset we created by scraping roughly half a million websites using a custom Google Chrome extension storing information related to the canvas. Classification leverages our key insight that the images drawn by canvas fingerprinting programs have a facially distinct appearance, allowing us to manually classify files based on the images drawn; we take this approach one step further and train our classifiers not on the malleable images themselves, but on the more-difficult-to-change, underlying source code generating the images. As a result, ML-CB allows for more accurate tracker blocking.

Keywords: Privacy, Device Fingerprinting, Web Measurement, Machine Learning

DOI 10.2478/popets-2021-0056
Received 2020-11-30; revised 2021-03-15; accepted 2021-03-16.

1 Introduction

Preventing online tracking is particularly problematic when considering dual use technologies [1–4]. If a tool maintains a functional purpose, even if it is not a mainstream staple of the web, the case for blocking the tool outright suffers from diminishing returns. The cost of privacy is functionality, a trade-off which few users are willing to take [5, 6].

HTML5's canvas suffers from this trade-off. Originally developed to allow low-level control over a bit-mapped area of a browser window [7], the canvas element was soon identified a boon for fingerprinting, given its ability to tease out uniqueness between devices while appearing no different than any other image rendered on a webpage [8]. This occurs because the same image, when drawn on the canvas rather than served as a single .jpg or .png image file, will render uniquely given a machine's idiosyncratic software and hardware characteristics (e.g., variations in font rasterization techniques like anti-aliasing or hinting, which try to improve rendering quality) [3, 9].

While methods exist to block or spoof use of the canvas [10], the current state of affairs leaves much to be desired—over-blocking (e.g., prohibiting javascript entirely [11, 12]) or under-blocking (e.g., blocking web requests based on *a priori* knowledge of a tracking purpose [13]) is the norm. This leaves us with the following grim outlook: "[I]n the case of canvas fingerprinting, there is little that can be done to block the practice, since the element is part of the HTML5 specification and a feature of many interactive, responsive websites [14]."

This paper introduces ML-CB, a means of increasing online privacy by blocking *only adverse* canvas-based actions. ML-CB leverages the key insight that the images drawn to the canvas, when used for fingerprinting, have a distinct and repetitive appearance, in part due to the highly cloned nature of fingerprinting programs on the web [15]. What is more, by labeling programs based on the images drawn—but training our machine learning models on the text generating those images—we are able to take advantage of a simple labeling process combined with a nuanced understanding of program text. This novel approach provides models that are less overfit to particular images and more robust—though not perfect—against obfuscation and minification.

In summary, we:

1. Present the **canvas fingerprinting dataset**, containing over 3,000 distinct canvas images related to over 150,000 websites which used HTML5's canvas element at the time of our roughly half-a-million-website scrape.[1]

***Corresponding Author: Nathan Reitinger:** University of Maryland, E-mail: nlr@umd.edu
Michelle L. Mazurek: University of Maryland, E-mail: mmazurek@umd.edu

1 Accompanying material available at: https://osf.io/shbe7/.

2. **Measure** canvas fingerprinting across the web.
3. Introduce **ML-CB**—a means of using distinguishable pictorial information combined with underlying website source code to produce accurate and robust classifiers able to discern fingerprinting from non-fingerprinting canvas-based actions.

2 Background and Related Work

Prominent, in-practice tools driving online tracking can be categorized at a high level into several buckets, including: (1) stateful tracking such as cookies [4]; (2) browser-based configurations [16, 17]; and (3) the canvas [3, 10, 18]. Cookies have been studied for a long time, are a form of stateful tracking (i.e., in the typical case, information must be stored client-side), and create problems related to consent (i.e., notice and choice has a long history of well-documented failures) [19–23]. Browser-based configurations, though a form of stateless tracking (i.e., no information needs to be stored client-side, allowing the tools to operate covertly), have also received a lot of attention from the privacy community [24]; more importantly, browser vendors are in a good position to address this area through default configurations, like Tor, Brave, Firefox, and Safari have done over the years [25–28]. Canvas fingerprinting, on the other hand, is a stateless form of tracking and is a dual-use tracking vector, making it difficult for a browser vendor to block without sacrificing functionality. Further, relatively few efforts to describe fingerprinting focus specifically on the canvas. For these reasons, we exclusively focus on the canvas.

2.1 The Canvas

Canvas fingerprinting originates from a 2012 paper explaining how images drawn to HTML5's canvas produce a high amount of Shannon Entropy [29] given the unique software and hardware characteristics of user machines (e.g., browser version, operating system, graphics hardware, and anti-aliasing techniques) [8, 30]. Researchers had Amazon Mechanical Turk (M-Turk) workers visit a website surreptitiously hosting a canvas fingerprinting script, in order to assess participant identifiability. The following piece of JavaScript code is an updated version of what the researchers used, illustrating the core of canvas fingerprinting.

```
1  var canvas = document.createElement('canvas');
2  var ctx = canvas.getContext('2d');
3  var txt = 'Cwm fjordbank glyphs vext quiz';
4  ctx.textBaseline = "top";
5  ctx.font = "16px 'Arial'";
6  ctx.textBaseline = "alphabetic";
7  ctx.fillStyle = "#f60";
8  ctx.fillRect(125,1,62,20);
9  ctx.fillStyle = "#069";
10 ctx.fillText(txt, 2, 15);
11 ctx.fillStyle = "rgba(102, 200, 0, 0.7)";
12 ctx.fillText(txt, 4, 17);
13 var strng = canvas.toDataURL();
```

Lines 1-2 create the canvas element; lines 3-12 add color and text (`fillStyle` and `fillText`, respectively); and line 13 converts the image into a character string using `toDataURL`. In fact, `toDataURL` is a lynchpin method for the fingerprinter [31]. This function allows the image drawn to be turned into a base64 encoded string, which may be hashed and compared with other strings. If a fingerprinting-compatible image, such as a mix of colored shapes and a pangram [8], is drawn, the resulting hash will have high entropy, leading to identification of a user (or more accurately, a device, assumed to be associated with a user [32]) among a set of users [33]. Notably, the amount of entropy varies per drawing, but colored images and text have been shown to be most effective [8, 33].

Following this work, a series of repositories for plug-and-play fingerprinting popped up [33–35], but it remained unclear whether these mechanisms were being adopted in the wild. Then, in 2016, Englehardt and Narayanan [36] conducted a large-scale measurement study with OpenWPM [37], finding that these types of fingerprints were indeed widely used. The catch was that the canvas did not have a solely malicious purpose, it was used to both track the user and benefit the user's web experience. And this posed the seminal question: How can canvas actions used *only* for device fingerprinting be distinguished and blocked [11, 38, 39]?

2.2 Approaches to Blocking Canvas Fingerprinting

Generally, three solutions to blocking "bad" canvas actions exist: (1) block or spoof *all* canvas actions; (2) prompt the user for a block–no-block decision; and (3) use blocklists to block or permit particular attributes, most commonly specific URLs [40].

Rote Blocking. The first option is most common among anti-tracker canvas-blocking techniques. Tools like `canvasfingerprintblock` [41] simply return an empty images for all canvas drawings. A similar approach may be seen in FP-Block [42] where a spoof,

including the FP-Block logo, was added to all canvas images. Though the researchers did distinguish between tracking on a main page and cross-domain tracking (preventing third parties from linking users between sites), the spoofing was nonetheless rote, failing to distinguish between "fingerprinting" or "non-fingerprinting" canvas actions. Likewise, PriVaricator [43] adds random noise to all canvas image output returned from the `toDataURL` call (see also [44]). Although the noise is only added to `toDataURL` output, this approached received critique for its identifiability by fingerprinters [45]. Baumann et al. argue that their DCB tool "transparently" modifies all canvas images and therefore achieves a similar spoof without sacrificing image quality or identifiability [45]. However, DCB also assumes a canvas image of 1,806 pixels; our scrape, discussed below in Section 3, identified images used for both fingerprinting and non-fingerprinting that were represented at 16x22 pixels, making these transparent changes likely noticeable to the user or adversary. FPRandom [46] approaches the problem like DCB and PriVaricator, but modifies the canvas by manipulating underlying browser source code to return slightly different values on subsequent function calls. A similar approach is taken by Blink [47], which uses a variety of configuration options to create the same type of image inconsistency. UniGL [9] follows suit by modifying all renderings made with WebGL (interactive 2D and 3D images) to make images uniform [16]. These last three efforts garner the same criticism for identifiability in image manipulation [48, 49], and, by modifying all images, this may lead to a cost-benefit ratio disfavoring adoption via adulterated functionality [5, 6].

User Prompt. The second option for blocking canvas actions centers on user-focused control. The canonical example here is Tor's prompt on all canvas actions, disabling these images by default and asking the user for permission to render [50]. A problem with this approach is the requisite knowledge required to correctly make a block–no-block decision. Many users, even experienced ones, would have no basis from which to decide whether to block or permit a particular canvas action.

Finally, there is a third option: choosing specfici instances to block/permit, generally using either heuristics or machine-learning classifiers.

Blocklist—Heuristic. For heuristics, in the simplest case (e.g., Disconnect [51]), a predefined list of domain names flagged as block-worthy is used. These lists are difficult to maintain, and do not always accommodate changes. For example, the company ForeSee, at one point in time, used fingerprinting techniques on .gov TLDs like ftc.gov and state.gov [52, 53]. After receiving scrutiny for the practice [54], the fingerprinting scripts were removed, but the company remains on Disconnect's blocklist [55]. A more advanced tool, FP-Guard [56], considers a canvas action "suspicious" if it reads and writes to the canvas, and more suspicious if the canvas drawing is dynamically created. Yet, in our crawl, we found examples of these actions used for both beneficial canvas images as well as tracking-based canvas images.

Perhaps the best example of where heuristic-based blocklists fall short is the popular false positive triggered on Wordpress's script meant to test emoji settings [57]. Although the script has a benevolent, user-focused purpose, it is often flagged by heuristics because it acts like a fingerprinting script, creating the canvas element, filling it with color and text, and reading the element back with a call to `toDataURL` [14, 49, 58, 59]. As we discuss in our results (Section 4.2), although blocklist-based heuristics for canvas fingerprinting, like the current state of the art from Englehardt and Narayanan (see Appendix A) [36], these heuristics are often highly accurate overall in part because they lean toward labeling instances as non-fingerprinting, which is the majority class, but may perform less well is correctly identifying instances of fingerprinting [36, 60]. Further, an adversary may adapt to these heuristics and purposefully avoid or include certain functions to escape classification.

Blocklist—Machine Learning. Seeking robustness, another class of research [15, 60–62] opts instead to use machine learning to make the block–no-block decision. Researchers here start with a set of ground truth (i.e., labeling programs through manual inspection or applying heuristic-based rules), but then build on the ground truth by training machine learning models to detect fingerprinting programs.

Researchers in [15], on which our work is based, attempted to solve the false positive problem by leveraging the fact that most tracking scripts are functionally and structurally similar. Using this key insight, researchers expert-labeled a set of Selenium-scraped programs, used a semantic representation of these programs via the "canonical form" (i.e., a string representation of the program which accounts for tf-idf ranked n-grams based on the program's data and control flows [63, 64]), and trained one-class and two-class support vector machines (SVMs) [65] on the programs. The result, on originally-scraped data, in the best case, had an accuracy of 99%. Though impressive, researchers took their model and applied it to an updated set of Selenium-scraped

programs. Here, the model's accuracy dropped significantly, down to 75% when labeling tracking programs and 81% when labeling functional programs. Moreover, the researchers acknowledge that this method would only work with a continually trained and updated SVM, because unseen programs (either due to obfuscation or novel functionality) would likely be inaccurately classified.

Overcoming some of these issues, FPInspector [60] (see also [61] which takes a similar approach, but uses an SVM and random forest as models) uses machine learning built on a heuristic-based understanding [36] of fingerprinting programs. Researchers applied static analysis (i.e., text-based abstract syntax trees [66, 67] mined for features using keywords commonly associated with fingerprinting APIs) and dynamic analysis (i.e., statistics-based features like function call counts and attributes of particular APIs like height and width of a canvas image) to generate source-code based features, and then used a decision tree [68] to split features on highest information gain [69] for classification. FPInspector proved accurate (i.e., ~99% accuracy, with ~93% precision and recall) on a manually created dataset[2] with manual-inspection retraining for improved ground truth. Interestingly, researchers in FPInspector noted how `toDataURL` was a watershed function for overall fingerprinting classification, citing it as one of two features with the most information gain (`getSupportedExtensions` was the second). FPInspector uses Englehardt and Narayanan's heuristic list (Appendix A) as ground truth to classify canvas fingerprinting [36], which as we note above works well for non-fingerprinting examples, but not as well on fingerprinting examples (see Section 4.2).

Our work uses a similar approach to [15, 60], but leverages manual classification of images instead of a heuristic-based ground truth. We also take advantage of program representation specifically aimed at JavaScript through our use of jsNice, helping alleviate problems related to classifying minified or obfuscated text. The following section outlines the architecture behind ML-CB.

3 Architecture

Our goal is to build a classifier that can distinguish between "non-fingerprinting" and "fingerprinting" canvas actions. To that end, we first generate a labeled dataset to be used for training and testing. The following sections (see Figure 1 for an overview) describe our nearly half-million-website scrape (3.1), resulting dataset (3.2), and labeling process (3.3). We then discuss canvas's use in the wild (3.4) before describing our fetching of website source code (3.5) and the machine learning models we used to train our classifiers (3.6).

3.1 Scrape

In order to classify the underlying programs driving canvas fingerprinting with supervised machine learning, a dataset of programs and labels is needed. The programs should be those used for both fingerprinting and non-fingerprinting purposes. To gather this data, we scraped the web in August, 2018. The scrape took nearly one week to complete.

We used Selenium, a popular Python web-scraping tool, and crawled 484,463 websites in total (limited due to budget constraints) [70]. We visited websites listed on Alexa Top Sites, using the Alexa Top Sites API, ordered by Alexa Traffic Rank [71]. To maintain efficiency, we parallelize this process with 24 non-headless (for capturing screenshots) Chrome browsers.

Each Chrome browser was driven to a targeted website's landing page. No additional gestures, such as scrolling down or moving the mouse, were used. Additionally, no sub-pages were visited,[3] and we set no "pause" between websites. As soon as the browser was finished loading the landing page, it could move on to the next website in the targeted URLs list.[4]

The browsers included a custom extension which pre-loaded the targeted website's landing page and

2 As is the case with our research, manual labeling is necessary given the lack of an existing "fingerprinting" dataset (e.g., Disconnect's list is by domain and not by program, and although other datasets have been released with fingerprinting examples [36], these are not updated frequently).

3 This design choice was made to identify a lower bound on canvas fingerprinting, when even minimal interaction with the website would garner a fingerprint for the user.

4 Although other researchers have noticed an increase in the number of non-functional files (i.e., JavaScript programs) initiated after waiting for a few seconds on each page [15], we did not notice a change in the files themselves when comparing calls to `toDataURL`. Waiting a few seconds may have increased the number of times a file was called, but typically no 'new' files were called within the waiting period. Therefore, we did not add a pause on each landing page.

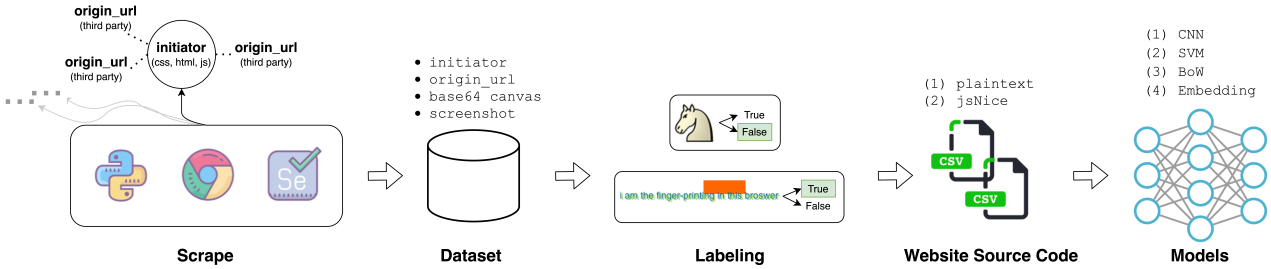

Fig. 1. ML-CB. We **scrape** (3.1) the web with a custom Chrome extension, adding a hook on the `toDataURL` function. The hook updates the **database** (3.2), storing canvas-related information. Canvas images, along with supporting material, are visually inspected for **labeling** (3.3). **Website source code** (3.5) associated with the images are then fetched and stored in either plaintext or jsNiceified form. Machine learning **models** (3.6) are trained on both images and text.

its associated HTTP links by adding the listener `onBeforeRequest.addListener` [72]. All requests (i.e., HTTP or HTTPS links) were parsed using `XMLHttpRequest` [73]. We term the linked content of a request to be a 'file' (i.e., either an HTML document with in-DOM JavaScript or a standalone JavaScript program with a .js file extension).

We scanned each file for the `toDataURL` function[5] based on a plaintext string match with the term *toDataURL*.[6] Upon a successful match, we added the file's information to our database of `toDataURL` occurrences. We also altered the file itself by appending a JavaScript prototype that modified the default `toDataURL` function, by requiring it to capture additional data about how it was used: the initiating domain (i.e., top-level initiator associated with the function's use), the requested URL (i.e., origin_url), and a screenshot of the page. When triggered by the execution of `toDataURL`, the prototype updated the entry in our database to show that `toDataURL` had actually been used, and to store the additional data. All database updates were formatted as SQL query strings, automatically downloaded by the browser as a .sql file. In this way, our scrape analyzed the use of `toDataURL` dynamically, unlike other studies which analyzed webpage text post-scrape, in a static fashion [36]. As the crawl evolved, our Postgres database [74] filled with canvas information.

3.2 Dataset

We structured the resulting dataset around `<initiator, origin_url>` primary key pairs. This is because a single domain (i.e., initiator) may call several files (i.e., origin_urls) which serve various purposes, and, ultimately, we want to block or permit a specific file from engaging in canvas fingerprinting. Notably, a single file may also draw several canvas images. We handle this case by storing each image (in base64 encoded form) in an array associated with the initiator and origin_url pair. If any of the images in the array are flagged as canvas fingerprinting, the origin_url will be labeled as fingerprinting. Finally, it is notable that the dataset may include more than JavaScript programs. This is because an initiator may interleave script tags with HTML and CSS.

Out of the 484,463 websites targeted, 197,918 unique initiators (i.e., domains) were inserted into the dataset, representing those websites with the string `toDataURL` located either in the DOM or in a standalone JavaScript file. This means that almost half of the URLs targeted called a file that used the function `toDataURL`, although this number is imprecise because some of the targeted websites resulted in time-outs or, for example, experienced implementation bugs such that `toDataURL` did not execute (meaning that our extension did not log the image to be drawn by the canvas, but did log the static use of `toDataURL`).

In total, 402,540 canvas images were captured by the scrape; these included 3,517 distinct images with a unique base64 encoding. These images occurred in the DOM of 103,040 files and in 177,108 standalone

5 Although several functions may be used to obtain a base-64 encoded string of the canvas image (e.g., `toDataURL` or `getImageData`), we focused exclusively on `toDataURL`. `toDataURL` is the most common function used in canvas fingerprinting [34] and this limitation is a common practice in the literature [60]. A similar procedure could be followed with other functions.

6 If the function's name were obfuscated, our hook would be unsuccessful, giving us a lower bound on use of `toDataURL`. Anecdotally, we often found that even when a file was noticeably obfuscated, `toDataURL` could be found as a plaintext string.

JavaScript files. This demonstrates that many canvas images are repeated across several domains. A further analysis of the canvas's use in the wild follows our discussion of labeling these images.

3.3 Labeling

We labeled each image according to its use for canvas fingerprinting, true or false. Our key insight is that labeling may be done with high accuracy by looking at the images themselves, rather than the corresponding program text. For example, here is a drawing from the dataset appearing to have no fingerprinting use:

And here is a drawing typically occurring in fingerprinting programs:

To confirm this insight, we ran a mini-scrape and reviewed the images produced by `toDataURL`. We found that most variations of the images used for fingerprinting conform to a similar pattern (e.g., the use of text, one or more colored shapes, and the use of emojis). In fact, many of these patterns come directly from popular repositories providing examples of canvas fingerprinting [34, 75] and are embarrassingly identifiable.

From the mini-scrape, we also found close cases, where we were unsure of the appropriate label based only on the image drawn. For example, Facebook's use of layered emojis (see Figure C (A.6) in the Appendix) did not, at the time, appear visually similar to other images used for fingerprinting. Although we might hypothesize a purpose based on the domain alone, we had some manner of doubt regarding the image's purpose—we would later learn that Facebook was an early adopter of Picasso-styled canvas fingerprints [33], which were frequently seen in our 2020 follow-up scrape (discussed in Section 4.2).

To resolve close cases like this at scale, we follow three steps. *First*, we inspect the image for recognizable similarity to known canvas fingerprinting images. *Second*, if we have any doubt regarding the purpose of the image, we next look at the screenshot associated with the initiator. From this, we can generally tell if the image drawn is present on the website's landing page; a common use-case here is using the canvas to draw a website's logo. *Third*, if we are unable to see the image on the website's landing page, we next look at the file (i.e., origin_url) that drew the image. This approach is telling because website operators, we found, did not commonly take steps to hide the purpose of certain functions, instead using names like `function generateFingerprint()` or headers like `Fingerprintjs2 1.5.0 - Modern & flexible browser fingerprint library v2`. Finally, if we were still unsure as to the purpose of the image, we classify the image as non-fingerprinting. We erred on the side of non-fingerprinting to ensure that our classifiers are using the most-likely, best-possible evidence—images that are most commonly associated with fingerprinting.

With these three steps in mind, we set out to label the entire dataset. A single researcher reviewed 3,517 distinct canvas images for classification. After the labels had been made, the same researcher re-reviewed all images labeled false (16 images reclassified) and all images labeled true (four images were reclassified from true to false). In the end, 285 distinct, positive fingerprinting images were stored in the dataset, along with 3,252 distinct, negative, non-fingerprinting images.

We note that we needed to make a design decision for mixed-purpose files. These occur in the dataset because we generated labels based on images, but primarily train at the granularity of a file—and a single file (i.e., origin_url) could produce multiple images. We consider a file "true" if it drew *any* image used for fingerprinting and "false" if it did not draw *any* images used for fingerprinting. In this way, the total, distinct images labeled as true or false in our dataset is 3,537, which is 20 more than the number reviewed by the expert. These 20 images relate to the case where the image has a ground-truth label of false, but because it is *sometimes* shared with a script that draws a "true" image, it is also labeled true in our dataset.

3.4 Canvas in the Wild

Combining labels with the data from our main scrape allows us to assess the state of canvas fingerprinting on the web (circa 2018). Using this perspective, we learn that canvas images using `toDataURL` are: (1) not predominantly used for fingerprinting, though their use for fingerprinting is rising; (2) repetitive and shared across domains; and (3) used consistently across website pop-

(A) All Images		(B) Fingerprinting	
🇺🇳	95,509 (23.7%)		8,946 (2.2%)
🇺🇳	75,044 (18.6%)		5,288 (1.3%)
🏴	71,688 (17.8%)		4,434 (1.1%)
🤦‍♀️	62,867 (15.6%)		2,214 (0.6%)
☐♀	19,983 (5.0%)		1,226 (0.3%)

Table 1. Most common canvas images (by highest count) out of the sum of all canvas images drawn by initiators. The left (A) considers the full dataset, while the right (B) considers only those images labeled as fingerprinting.

Origin URL	(/FP)
cdn.doubleverify.com/dvbs_src_internal62.js	2,301 (10.7%)
rtbcdn.doubleverify.com/bsredirect5_internal40.js	1,746 (8.1%)
cdn.justuno.com/mwgt_4.1.js?v=1.56	1,165 (5.4%)
d9.flashtalking.com/d9core	741 (3.4%)
cdn.siftscience.com/s.js	724 (3.4%)

Table 2. Top five fingerprinting files representing the most commonly occurring origin_urls in the dataset. Counts show how often the file was executed by an initiator (% of all 21,395 files used for fingerprinting). Since the scrape, some of the URLs have been modified; prior versions can often be found at the Internet Archive Wayback Machine (see, e.g., [76]).

Rank Interval	`toDataURL`	Fingerprinting
[1,100)	56%	27%
[100,1K)	40%	18%
[1K,10K)	36%	11%
[10K,100K)	36%	5%
[100K,400K]	39%	3%

Table 3. Percentage of websites in different rank intervals, ordered by Alexa Top Rank, using `toDataURL` in any way or for canvas fingerprinting. Both uses become less common as website rank decreases. `toDataURL` is commonly used for purposes other than fingerprinting.

ularity, but pooled, in terms of fingerprinting, around content related to news, businesses, and cooking. We group these impressions around *how* the canvas is used and *who* is using it.

Similar results may be shown for the files responsible for drawing these images. Websites relied on a total of 280,148 files to draw canvas images (i.e., sum of the distinct origin_urls used per website). Of these, 21,395 leveraged canvas fingerprinting. Table 2 shows the most common files used for canvas fingerprinting, and the count of how many times each file was used by a website. As shown in Englehardt and Narayanan's work [36], the number one provider of files relying on canvas fingerprinting is doubleverify, though the rest of the providers represent a mixed bag, with newer players like justuno and siftscience being introduced.

From this view, we can verify the highly-cloned nature of canvas images. The most common image in our dataset is used on more than 95,000 websites. And the average number of times a single image is re-used across websites was 114; that number raises slightly when considering only fingerprinting images, to 125. What this means is that use of the canvas is becoming more common on the web, though the most likely use-case is to draw emojis rather than a bespoke logo or other image.

Who Uses the Canvas. When considering website popularity, based on Alexa Top Rank at the time of our scrape, we see that use of the canvas is not isolated to top-ranked websites. Figure 2 provides a close-up view of the top 50 websites and their use of the canvas. More than half of these websites (30) used `toDataURL`, and over a quarter (16) used `toDataURL` for canvas fingerprinting.[7] This view may be expanded to the entire dataset (Table 3), showing relative consistency despite rank. These results show that use of `toDataURL` does not always equate with a fingerprinting purpose. Less than half of the websites that used `toDataURL` used it for fingerprinting. Similar to previous work, we also find that use of the canvas for fingerprinting is increasing, from 4.93% in 2014, to 7% in 2016 (with sub-pages included), to 11% at the time of our scrape [3, 36].

Finally, we wanted to assess the category of websites fingerprinting programs may be associated with. To do this, we fetched URLs found in our dataset and used the website's landing page text as input to Google's Natural Language Processing API, which assigns text into categories roughly mirroring Google's AdWords buckets (620 categories in total) [78]. The fetch occurred in early September, 2020. We note that the process of assigning website text into categories likely creates mis-

7 We consider an initiator (e.g., google.com) to be "in" the dataset based on a loose string match (i.e., domain *or* subdomain, using the Python package `tld` [77]) between the Alexa Top Ranked URL and the URL found in our dataset. For example, using the URL from Alexa, `herokuapp.com`, we would consider the following URL from our dataset to be a match, `word-to-markdown.herokuapp.com`.

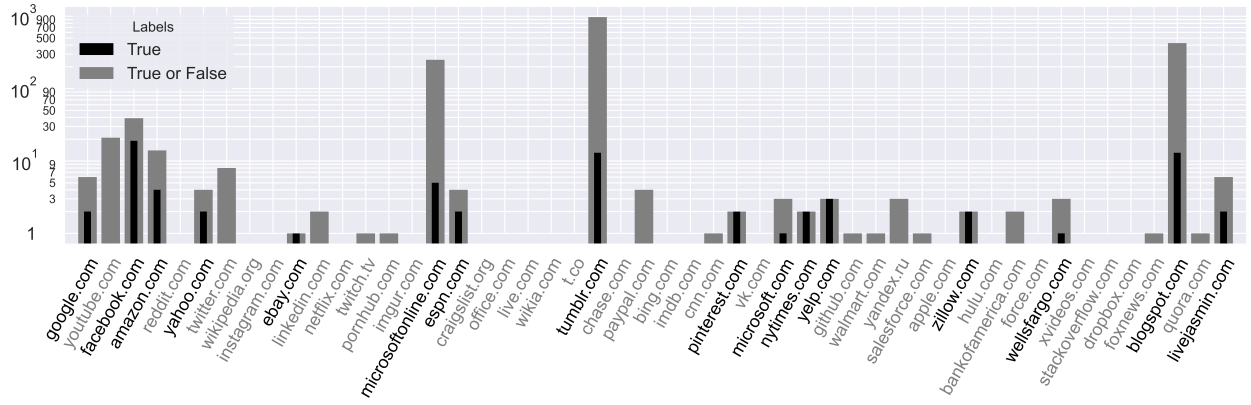

Fig. 2. How the top 50 websites (according to Alexa Top Sites at the time of the scrape) use `toDataURL` with the canvas: Counts (y axis) describe to the number of distinct images drawn by associated domains (i.e., including domain and subdomain matching) in files captured by our script, both total (gray) and labeled as fingerprinting (black). The domain is noted along the x axis. Websites are ranked from left (highest) to right (lowest).

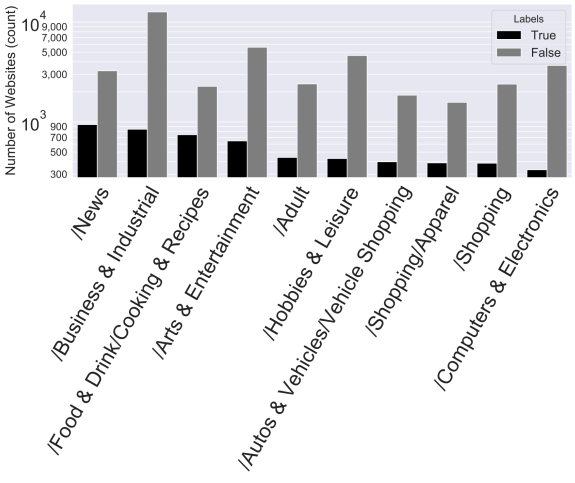

Fig. 3. Distinct initiators from the canvas fingerprinting dataset categorized using Google Natural Language categories. Counts show the number of websites using `toDataURL` for fingerprinting (black) and not for fingerprinting (gray). Raw counts are provided because a single website can pertain to multiple categories (on average 1.5 categories per website). Counts are influenced by overall distribution of Alexa Top Site rankings.

classifications, and it is likely that website landing page text changed from the time of our original scrape to the time of this categorization. On the whole, however, the top ten categories we identified (Figure 3) reinforce findings from prior work [36, 60].

We found that a large number of fingerprinting programs were associated with News and Shopping websites (shopping shows up three separate times, including subcategories like vehicles and apparel), food recipe websites like therealfoodrds.com or ambitiouskitchen.com, and business-related sites, like retreaver.com or ironmartonline.com. This reinforces prior work showing that news websites were more likely to use fingerprinting, with additional fine-grained categories provided by Google Natural Language.

3.5 Website Source Code

At this point in ML-CB's architecture, we have collected a dataset of hyperlinks (i.e., origin_urls), their associated canvas drawings, and their fingerprinting–non-fingerprinting labels. If we intended to train classifiers with these images alone, then we could move on to training. Instead, we obtain JavaScript source code for each canvas drawing, and use that as the basis for our classifier. In the next sections, we explain why images alone are not sufficient, describe how we pre-process source code used in our classifiers, and provide an overview of the models we used.

3.5.1 Why Not Use Images

Models trained only on canvas drawings will be susceptible to unseen or easily-modified images, similar to the limitations expressed in [15]. For instance, suppose fingerprinting scripts started to use the cat image shown in Section 3.3 instead of the common overlapping circles plus the pangram `Cwm fjordbank glyphs vext quiz` (i.e., the `fjord` pangram). It is plausible that the difference in entropy would be small enough that the

image may still be useful for fingerprinting, but this would significantly impede an image-based classifier. Indeed, essentially all of the fingerprinting examples in our dataset include text, so images without text would be very likely to fool a classifier trained with this sample [8].

Using the fingerprinting program's text circumvents this problem because, although it may be somewhat easy to change a fingerprinting image drawn to the canvas, it will be harder to change the textual properties generating that image [79]. Programs fingerprinting with the canvas element, in mostly the same fashion, create the canvas element, fill it with colors and text, and rely on `toDataURL` as a final step—an identifiable pattern used over and over. This can be seen in Section 2.2's example fingerprinting technique.

In short, using images alone for classification leaves too much room for an adversary to easily swap images, while using program text alone is too cumbersome when generating labels at scale. We take a best-of-both-worlds approach, leveraging a fast labeling process—labelling thousands of canvas actions resulting in the same image, though appearing textually heterogeneous, in one step—while still training on less malleable source code.

The following subsection discusses how we obtained program text from our original dataset, and how we then processed this text with jsNice [80]. For an example of how the resulting text appears in our text corpora, see Figure 7 in Appendix D. Following this subsection, we discuss the models we used and their respective architectures, before turning to our results.

3.5.2 Text Corpora

We fetched program text associated with each of the origin_urls in our dataset using the Python `requests` package [81]. To mitigate the effects of potential JavaScript obfuscation, we apply jsNice to achieve a semantic representation of program text (using the command line tool interface to jsNice [82]). We offer a brief background on jsNice and then discuss our resulting text-based corpora.

jsNice. jsNice deobfuscates JavaScript text by using machine learning to predict names of identifiers and annotations of variables (see Appendix, Figure 7 (B) for an example). The tool works by representing programs in a dependency network which allows for the use of probabilistic graphical models (e.g., a conditional random field) to make structured predictions on program properties [80, 82, 83]. This works well for our purpose because jsNice adds natural language annotations, regardless of obfuscation and minification, which may be leveraged by our classifiers.

Corpora. We created two program-text corpora, one plaintext and one jsNiceified.[8] These corpora inherited labels from our image labeling phase, creating `<program, label>` rows.

We first pre-processed the plaintext corpus, dropping 20 rows for empty values, 1,502 rows as duplicates (e.g., dropping programs based on an exact string match, with 1,316 negative duplicates and 186 positive duplicates), and 124 programs for having a length of less than 100 characters (i.e., 113 negatively labeled programs and 11 positively labeled programs). The final plaintext corpus included 84,855 total programs, with 2,601 positively labeled programs and 82,254 negatively labeled programs. We picked the character limit of 100 based on manual inspection of these programs, most of which were HTML error codes (500 or 404). The average character length per program was 106,036, while the maximum character length was 11,838,887.

In pre-processing the jsNiceified corpus, we removed 72 empty rows, 1,806 duplicates (1,467 positively labeled programs and 339 negatively labeled programs), and 115 programs with less than 100 characters (103 negatively labeled programs and 12 positively labeled programs), resulting in a final corpus of 84,735 programs. This included 82,359 negative examples and 2,376 positive examples. The average character length per program was 111,041, with a maximum of 11,819,497.

To quickly assess our resulting dataset, we took all positive examples from our plaintext corpus, tokenized each program, and compared the Jaccard similiarty $\left(\frac{|program1 \cap program2|}{|program1 \cup program2|}\right)$ using pairwise matching [84, 85]. The mean similarity score was .40, with 25, 50, and 75% of the data represented as similarity scores of .22, .38, and .58, respectively (Figure 4). This validates prior work suggesting that a large amount of canvas fingerprinting programs are cloned across the web [15].

Lastly, it is worth noting a possible discrepancy in our text-based data. Our original scrape did not store program text, only labels and origin_url links to program text, and we first labeled data prior to fetching

8 As stated in Section 3.3, websites may either call `toDataURL` in a standalone JavaScript file or use interleaved script tags in an HTML file. This allows for the possibility of classification at the per-script level, breaking each script tag into its own mini-example for training and testing. We tested this approach, but found it had a high tendency to overfit our data. Therefore, we did not move forward with this approach.

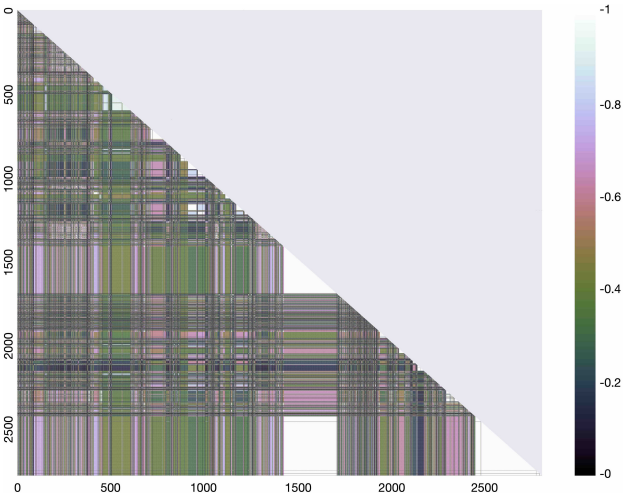

Fig. 4. Jaccard similarity between all tokenized programs (i.e., files) manually labeled as fingerprinting. Programs along the diagonal are compared against themselves (i.e., a Jaccard score of 1). Lighter colors suggest more similarity.

website source code. As a result, almost two months had passed from the time of our original scrape to the time our fetching was complete. This means that a website could have changed a particular JavaScript program from the time when the scrape occurred to the time the program text was downloaded. Although possible, this is unlikely, as previous research has shown that website source code has a half-life of two or more years, and JavaScript tends to change even more slowly [86–90].

3.6 Machine Learning Models

We trained four types of machine learning models: a convolutional neural network (CNN), a support vector machine (SVM), a Bag-of-Words (BoW) model, and an embedding model using a pre-trained GloVe embedding [91].

We use the CNN for an initial reasonableness check on our classifications; this model was trained only on the images drawn to the canvas. While the CNN did produce accurate classifications, its robustness is questionable for the reasons stated in Section 3.5.1, and confirmed in our tests conducted with follow-up data (Section 4.2). The SVM was picked for its efficiency, and because it was shown to be effective in [15]. We hypothesized, based on the limitations of [15], that this model would not adapt to the slowly changing methods used for canvas fingerprinting and would need to be updated frequently. Finally, we used a BoW and embedding model to assess the differences between our plaintext corpus (HTML, CSS, and Javascript) and jsNiceified corpus (more likely to include natural language generated by jsNice). We trained the SVM, BoW, and embedding models on both the plaintext and jsNiceified corpora. Each of the models' architecture is discussed in turn.

3.6.1 Images

A standard CNN was used to classify images from our dataset. The CNN relied on ResNet50, a 50-layer CNN pretrained on the ImageNet dataset [92, 93]. Considering raw images, we used a 20% test-train split for hold-out cross validation (i.e., training with 2,823 examples, 2,691 false and 132 true, and then testing on 707 examples (674 false and 33 true). All images were reduced to a size of `(3, 150, 150)`. Six non-fingerprinting (false) images were unreadable by Pillow [94] and one false image erred during conversion, resulting in a slight difference in total images between our dataset and this model. Because of the highly one-sided distribution of positive examples to negative examples, we also used data aggregation in the form of image manipulation (horizontal, vertical, and 90-degree rotations, along with image "squishing" [95]). We trained the model at three epochs, using a one-cycle policy [12] and discriminative layer training, with the initial layers trained at a learning rate of 0.0001 and the ending layers trained at 0.001 [96].

3.6.2 Text

For our three text-based models (i.e., SVM, BoW, and embedding), we followed the preprocessing steps outlined in Section 3.5.2. Given the skewed distribution of negative to positive examples, we also downsampled the majority class to meet the minority class's example count. To do this, we used all positive fingerprinting examples and randomly selected negative examples with `sklearn`'s `resample` function, which generates n random samples from a provided set. We used Stratified K-Fold (SKF) cross-validation (ten folds) on each model to ensure an accurate balance between testing and training [60, 97]. For a performance metric, we rely on the $F1$ score in each of the models, though we report the accuracy, precision, and recall as well. The *F1* score (harmonic mean of precision and recall) is a more realistic measure of performance than accuracy, and is often used in natural language processing [98]. Finally, to en-

sure consistent performance despite downsampling, we repeated the above process five times, storing the average of the ten folds' scores per loop and averaging the five loops to produce a final score.

Regarding the architecture of specific models, the SVM's feature extraction used a pipeline of count vectorization, to tokenize and count occurrence values; term frequency times inverse document frequency, to turn the occurrence matrix into a normalized frequency distribution; and stochastic gradient descent with a hinge loss parameter, for a linear classification. We used the $l2$ penalty for regularization, with an alpha value of 0.001 to offset the $l2$ penalty. This helps reduce overfitting and increase generalization [99].

For the BoW model [100], we tokenized text using the `keras` tokenizer, which vectorizes a text corpus; in this case, the set of programs from our training class. This produces a set of integers indexed to a dictionary. We used a limit of the 100,000 most common words and then transformed the tokenized programs into a matrix. Our `keras` sequential model contained a dense layer with a rectified linear unit as an activation, a dropout layer of 10%, and a final dense layer using the Softmax function as an activation. The model was compiled with the Adam optimizer (for simple and efficient optimization [101]), used categorical cross-entropy for loss (given the principled approach in its use of maximum likelihood estimates [102]), was fit on the training data for ten epochs, and evaluated with our $F1$ performance metric on testing data.

The embedding model used the same tokenization technique as the BoW model, but created a sequence of integers instead of a matrix for text program representation. The sequences were prepended with 0's to ensure each integer-as-program sequence was the same length as the longest sequence (maximum length set to 1,000). A `keras` sequential model was used, starting with an embedding layer. We used the popular, predefined GloVe embedding matrix (`glove.42B.300d`) because we wanted to assess the contrast between plaintext and jsNiceified programs [91]. We allowed the embedding's weights to be updated during training. The model then used a one dimensional convolution, followed by a maxpooling layer, to reduce overfitting and generalize the model [103]. The next layer in the model was a Long Short Term Memory (LSTM) recurrent neural network [104]. An LSTM will be particularly suited for this purpose given the long memory of the network, as opposed to a typical recurrent neural network. Our `keras` model ended with a dense layer using the Softmax activation. We compiled with the Adam optimizer, used binary cross-entropy for loss, and trained for ten epochs.

4 Results

To assess our machine learning models, we evaluate them using: (1) data produced from the original scrape; (2) a follow-up, up-to-date test suite; and (3) an adversarial perspective, where heavy minification and obfuscation are applied to the test suite. Table 4 provides an overview. It is useful to keep in mind that error here means broken website functionality (i.e., false positive) or permitted tracking (i.e., false negative).

Overall, we can see that all of the text-based models maintain high accuracy and relatively high $F1$ scores on both the original data (test-train split) and the test suite. This is true despite our lack of heavy optimization (i.e., GPU support was not used) and hyper-parameter tuning. Also, as we hypothesized, the CNN did fare well with original data, but had problems when considering new images found in the test suite. Overall accuracy for all models also decreased in the adversarial setting, though both the BoW and embedding models nonetheless report moderately high accuracy scores. A full analysis of each of the three categories follows, along with summaries regarding three hypotheses:

H1 ML-CB's use of text is more effective than images or heuristics
H2 jsNice transformations are useful
H3 ML-CB is robust against an adversary who obfuscates and minifies website source code

4.1 Original Scrape

The data here comes from our original scrape capturing canvas images, using one researcher to label those images, and then fetching program text associated with each image. For the plaintext corpus, the model observed a total of 5,202 examples; 4,682 of those examples were used for training and 520 for testing (10% per fold). For the jsNiceified data, the model observed a total of 4,752 examples, training on 4,277 examples and testing on 475 examples (10% per fold).

As may be expected given the nature of a train-test split, all models perform well ($F1$ scores $\geq 95\%$); in fact, there is almost no difference between the models' performance. This suggests that more complex transformations like jsNice or more complex models like the BoW

	Original Scrape						Test Suite						Adversarial Perspective					
	SVM		*BoW*		*Embedding*		*SVM*		*BoW*		*Embedding*		*SVM*		*BoW*		*Embedding*	
	P	jsN	P	jsN	P	jsN	P	jsN	P	jsN	P	jsN	P	jsN	P	jsN	P	jsN
F1	97	97	98	98	95	95	83	84	86	87	84	86	53	50	55	80	60	74
Accuracy	97	97	98	98	95	95	91	91	93	93	92	93	59	59	85	90	69	84
Precision	98	98	99	99	98	99	72	73	77	78	75	81	37	34	79	73	43	60
Recall	97	97	96	96	92	91	100	100	96	98	96	92	99	89	56	91	99	98

Table 4. Showing the average of the ten folds' $F1$ performance (per the SKF cross validation method used), averaged again over five separate executions of the models, to help generalize performance. The highest $F1$ scores out of all models are noted with a gray box.

or embedding are not needed. It is also worth pointing out that this result is on par with previous work using machine learning to distinguish text-based fingerprinting programs (e.g., [15, 60] had 99% accuracy scores on an original set of scraped programs).

CNN. On original image data, averaging ten separate test-train runs, the CNN was also fairly accurate (98.9%) and performed slightly worse when classifying positive examples ($F1$ score of 89.2%, with recall at 86.1% and precision at 92.7%). We likely could have improved these numbers with additional tuning of the model (e.g., using downsampling or something like SMOTE [105]) or solidified this finding with more robust cross validation, but we can already see the model's shortcomings when comparing images alone (see also Section 3.5.1). Figure 5 (A) illustrates an example weakness—images labeled as fingerprinting can appear very similar to those used for non-fingerprinting, leading to errors.

Summary. When considering the original data, ML-CB is not appreciably better than classification based on images (H1); all models perform well. Likewise, adding jsNice (H2) does not improve performance with this original dataset. While a fraction of files (i.e., scripts) in this dataset are obfuscated (see [60, 106]), this evaluation does not provide sufficient evidence regarding the models' robustness to adversarial obfuscation (H3).

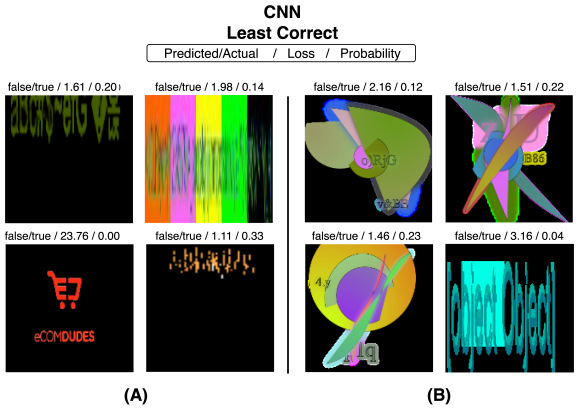

Fig. 5. Most likely incorrect predictions made by the CNN. Images from the original scrape are shown on the left (A), while images from the test suite are shown on the right (B). For further examples of the variety in fingerprinting images, see Figure 6 in Appendix C.

4.2 Test Suite

In order to more accurately evaluate these models, we followed the approach used in [15] and created a second set of scraped programs, using the same methods discussed in Section 3.1. Initially, we targeted 2,200 websites, using our original dataset, original Alexa Top Rank list, and the Tranco list of the one million top sites (pulled on September 10, 2020) as sources [107].

To obtain a wide variety of test cases, we aimed for a set of websites categorized in the following buckets:

- last 100 URLs in Tranco
- 200 random URLs labeled false in original dataset
- 300 random URLs labeled true in original dataset
- 100 random URLs from last 500,000 Tranco URLs
- 300 random URLs outside top 100 Tranco URLs
- first 100 URLs in original Alexa Top Rank list
- first 100 URLs in Tranco
- first 1,000 URLs in Tranco but not original dataset

We ran our scraper on these 2,200 URLs, hooking the `toDataURL` function call and creating a second dataset (hereafter, the test suite). Our scrape occurred in early October, 2020. The scrape took less than one day to complete.

The resulting test suite contained 906 distinct domains identified as using `toDataURL` and 181 distinct canvas images. The same researcher labeled the images,

returning a set of 90 true images and 95 false images. As described in Section 3.3, a small number of images were labeled both true and false depending on the origin_url the image was associated with. This aligns with our design preference to label as "true" when handling multiple image labels found within a single program (i.e., a program drawing several benevolent images and one fingerprinting image is labeled as fingerprinting).

We then fetched origin_urls to generate the plaintext corpus and applied the jsNice transformation to receive the jsNiceified corpus. Because we wanted a one-to-one comparison on the test suite, we dropped two negative examples from the plaintext corpus which did not transform due to a jsNice error, resulting in 318 negative examples and 90 positive examples.

We train one model on the original plaintext corpus and one on the original jsNiceified corpus, then test these models on the test suite's plaintext and jsNiceified data, respectively. We assess performance on the test suite for each fold in our SKF cross validation, taking the average for all folds, and repeating this process five times, reporting the average of the five runs in Table 4.

Overall, most models perform well on the new data. The average $F1$ score among all models is in the mid 80s, with accuracy scores in the low 90s. Notably, precision scores are low for most models, with high recall scores. We can also start to see the case for why jsNice is needed. Although most models perform better across all metrics when using jsNice, the difference is slight, but begins to show higher gains in the embedding model, which better balances precision and recall.

Heuristic Comparison. We next want to compare our models against state-of-the-art heuristic approaches. One possible approximation is to compare results: do we identify the same fingerprinting files as prior work? This comparison, however, is not straightforward; different datasets use different strategies to decide which and how many websites to test, and were collected at different times, meaning tracking companies may have moved their fingerprinting scripts to new files, or the content of the same URL may have changed. Datasets like Englehardt and Narayanan's original work (2016) [36] and the more recent FPInspector (2021) [60] also focus on general fingerprinting rather than canvas fingerprinting specifically and contain URLs rather than the contents of files that may be fingerprinting. Acar et al. (2014) [3] released domain names of fingerprinting scripts rather than full paths. All three publicly released datasets only include examples of fingerprinting, not examples of non-fingerprinting.

As such, we calculate overlap between our dataset and these datasets as follows: considering only origin_urls we identify as fingerprinting, how much overlap is there between fingerprinting URLs in our data (our original and test suite datasets combined) and prior datasets. We find that the overlap with Englehardt and Narayanan [36] is 1% of their dataset (0.7% of ours). Our overlap with FPInspector [60] is 14% of their dataset (9.8% of ours). And the overlap with Acar et al. [3] (truncating our dataset to only domains to match theirs) is 20% of their data (.8% of ours).

For a more meaningful heuristic comparison, we used openWPM [36, 37] on the URLs in the test suite, filtering for "canvas fingerprinting" according to the heuristic used in [36] (see Appendix A). To make the comparison one to one, we considered only those origin_urls (i.e., script_urls) captured by both our scraper and openWPM, 306 urls in total. On this subset, the heuristic maintains a high accuracy rate, at 93.8%, but low $F1$ score, at 77.6% (precision at 80.5% and recall at 75%). Considering these origin_urls only, ML-CB's BoW model with the jsNice corpus has an accuracy of 97.6% and an $F1$ score of 91.4% (with precision at 94.7% and recall at 88.9%). We hypothesize that this occurs because openWPM performs best on typical canvas fingerprints (i.e., the `fjord` pangram), as does our classifier. Finally, we modify the heuristic to be more accurate on the test suite by dropping the requirement that at least two colors are used in the fingerprint. The heuristic finds increased performance, but still less than the ML-CB BoW model, achieving an accuracy of 94.8%, but an $F1$ score of 81.8% (81.8% precision and recall).

CNN. We also used the test suite to assess our image-based CNN model trained on original scrape data and tested on renderable, distinct images associated with the test suite (181 examples in total, 108 negative and 73 positive). We followed the same procedures stated in Section 3.6.1 (e.g, image manipulations, training at three epochs with a one-cycle policy and discriminative layer training). The model achieved an average accuracy of 82%, with an $F1$ score of 72.1% (100% precision and 57.4% recall) on ten separate test-train runs. Notably, images most likely to be incorrectly labeled by the CNN were Picasso-styled images, most of which the model predicted as negative. Figure 5 (B) demonstrates another weakness of the CNN: new or different images can easily trick the classifier, as the Picasso-styled fingerprinters in the test suite did.

Summary. As shown in Table 5, ML-CB handles the up-to-date test suite better than the CNN ($F1$ scores of 72% versus 91% when using the BoW model) and bet-

Test Suite (Subset)				
	F1	*Accuracy*	*Precision*	*Recall*
ML-CB (BoW jsNice)	91	98	95	89
Heuristic (improved)	82	95	81	75
Heuristic (original)	78	94	82	82
CNN*	72	82	100	57

Table 5. Comparing ML-CB against the heuristic and CNN, percentages rounded up. All models (except the CNN* which reports results on the full test suite) use a subset of programs found in the test suit which were also identified by openWPM, for a one to one comparison. Although the heuristic shows high accuracy, ML-CB offers a better balance of precision and recall.

ter than heuristics ($F1$ score, in the improved heuristic, of 82% versus 91%); therefore, H1 holds at this point. However, although there is a performance boost by using jsNice in the full test suite, the difference is not substantial, meaning that H2, whether jsNice is useful, only partially holds. Finally, although we found examples of obfuscation in the test suite, we still do not have enough information to assess whether ML-CB resists adversarial techniques (H3).

4.3 Adversarial Considerations

A canvas fingerprinting detection system must also consider the ad- and tracker-blocking ecosystem it would possibly be deployed in [108]. If straightforward attempts to fool the classifier via obfuscation are successful, then ML-CB will be less useful.

To approximate this type of adversary, we applied heavy, but standard, obfuscation and minification to our test suite.[9] We accomplished this by iterating over the plaintext corpus from our test suite and obfuscating all JavaScript programs with the JavaScript Obfuscator Tool [109] (see Figure 7 (C) in the Appendix for an example). We also minified all HTML and CSS statements using a `numpy` package `html-minifier` [110] or a `Python` tool `htmlmin` [111] (if the `html-minifier` failed).

For minification, we used typical techniques, such as the removal of whitespace, the removal of optional tags, and the reduction of boolean attributes. For the JavaScript Obfuscator Tool, a full list of flags may be found in Appendix B, but the notable flags include: control-flow-flattening, to alter code structure [112]; dead-code-injection, to thwart language-based classifications; unicode-escape-sequence, to make natural language classification harder; numbers-to-expressions, to further change control flow; string-array, to replace natural language strings with hexadecimal arrays; and transform-object-keys, to turn objects into functions.

The obfuscated and minified results became our "plaintext" adversarial corpus. We then applied jsNice to the obfuscated and minified programs to create a "jsNiceified" adversarial corpus. This process is illustrated in Appendix D. Due to errors in obfuscation, the final dataset held 289 negative examples and 86 positive examples. As described in 3.5.2, we train on our original corpora (using ten-fold SKF) and test on the new "plaintext" adversarial corpus and "jsNiceified" adversarial corpus, separately.

The classifiers perform poorly against the "plaintext" adversarial corpus, but adequately against the "jsNiceified" adversarial corpus, with accuracy scores as high as 90%. It is unsurprising that a model trained on natural-language plaintext is ill-equipped to assess non-natural-language code (e.g., hexadecimal strings) in the obfuscated corpus. On the other hand, applying jsNice restores some of the natural-language features, including predicting names and types. A classifier trained using jsNice-ification is able to take advantage of these features. Appendix E further illustrates this point, showing the contrast between obfuscated code (unreadable) and obfuscated-but-jsNiceified code (more readable).

The improvement on the adversarial corpus when using jsNice validates our hypothesis that jsNice adds natural-language meaning to otherwise difficult-to-parse source code, motivating its use in this setting. Moreover, we can see that the SVM—which we expected to need more continual updating—does not fare well, even with the use of jsNice. The same is true for the embedding model, possibly due to its use of predefined weights taken from website text (i.e., the stock GloVe weights) rather than website source code. We would likely see a performance boost if the embedding model used its own set of weights tuned to this particular environment.

Lastly, we note that although the models in the adversarial case do not perform nearly as well as in the original scrape or test suite, to some extent, this outcome correctly aligns incentives. Because obfuscation creates many false positives (low precision scores), our classifier may incentivize website owners who are not conducting fingerprinting to avoid obfuscation. On the

9 We did not apply other, less out-of-the-box kinds of adversarial perturbations, such as substantive code changes. We also did not consider use of alternative functions, such as `getImageData` to avoid a classifier's detection of `toDataURL`, but we could easily adjust our pipeline to recognize such functions.

other hand, trackers who try to use obfuscation to enable fingerprinting are reasonably likely to be identified.

Summary. One of the advantages of images and heuristics is that these methods are not as heavily affected by obfuscation—the same image is eventually drawn, and openWPM uses dynamic analysis to measure features in part for this reason (H3) [36]. The trade-off here is that these methods ossify easily, with Picasso-styled images being missed by both (i.e., short, same-color strings are missed by the heuristic's second requirement, see Appendix A). At the same time, the heavy obfuscation used in our adversarial corpora, which weakens ML-CB, is hypothetical, and not consistently or heavily applied by website owners, even though these methods have existed for years. This suggests that it may be best to prioritize generalizable models in canvas fingerprint blocking tools, something achieved by ML-CB's use of text, rather than resilience from obfuscation (H1). Finally, from this vantage point, we can see that jsNice is only partially needed now (H2) but may provide some protection against future, heavier obfuscation.

5 Discussion

HTML5's canvas is a dual-use technology, currently used by 37% of the top 10K websites (Table 3). At the same time, using the canvas for device fingerprinting is rising, with websites ranked in the 1K to 10K interval engaging in canvas fingerprinting at a rate of 4.93% in 2014, 7% in 2016 (sub-pages included), and 11% in 2018 [3, 36]. Moreover, the current state of the art for blocking canvas fingerprinting is either over-aggressive, in a block-all manner, or under-aggressive, with rigid heuristics.

ML-CB presents an alternative approach. Using humans to make judgments on easily distinguishable canvas fingerprinting images to quickly and easily create a set of ground truth, allowing classifiers to be trained on website source code. This approach makes it feasible to use supervised machine learning on continuously trained or easily updatable classifiers. ML-CB may also be combined with other classifiers in a wisdom of the crowd approach—e.g., using a bitwise OR operation on the outputs from the CNN and text-based models from above—or used as a plug-in to identify canvas fingerprinting in systems that might otherwise use a heuristic-based list to define canvas fingerprinting (Section 2) [60]. What is more, this approach may be extended to other forms of fingerprinting like AudioContext, mobile device orientation, or touch-focused fingerprinting, which are likely heavily cloned across the web and may rely on identifiable hallmarks when used for fingerprinting [34, 113, 114]. The difficulty here would be finding a way to relax the dual-use problem with an easy-to-identify proxy, which we leave to future work.

As a final recommendation, we urge policymakers to take a more active role in the ad- and tracker-blocking ecosystem. The use of surreptitious tracking mechanisms, like canvas fingerprinting, is rising, and, currently, nothing stops a website from using these stateless tools—a barely legible statement in a privacy policy may be required in some jurisdictions, but is unlikely to bring awareness to the practice [115]. To change the Panopticon-styled Internet we have today, tools like ML-CB are necessary, but not sufficient, and must be accompanied by non-technical solutions like legislation [116].

5.1 Limitations

A primary limitation to ML-CB is the self-labeled nature of supervised learning, which may have biased the classifier. However, although it may have been possible to outsource the image-based labeling to crowdworkers, training a crowdworker to distinguish fingerprinting from non-fingerprinting would require supplying straightforward guidelines (i.e., distinguishing *meaningful* content like the cat from *meaningless* content like the `fjord` pangram), and this requires the crowdworker to also have context about the website on which the image was found. Further, most of the canvas images in the dataset were both illogical and non-fingerprinting. This occurs because many of these images are used as small background icons or pieced together given some user interaction. As such, non-fingerprinting examples would have likely been labeled fingerprinting by workers. Overcoming false positives by providing more training about true positives would inherit the same bias as relying on an expert for labeling. For this reason, we opted to label the images ourselves.

Second, ML-CB produces more false positives than false negatives. This likely occurs because of our design decision to aggressively downsample, based on the respective difficulty of classifying positive versus negative examples. Given that the vast majority of examples in our dataset are negative (non-fingerprinting), the classifier's strength would be oversold if we did not downsample and let the classifier prioritize a prediction of non-

fingerprinting—as a ground truth label is, by default, more likely negative. This approach enables a better performing classifier, but one that potentially "blocks" benevolent files, worsening the user's experience. Depending on context, it may therefore be preferable to alter this judgment and use less downsampling; however, in order to better assess classification decisions, we did not take that approach in this paper.

Finally, although ML-CB inferred tracking based on "typical" canvas fingerprinting images, there are potentially beneficial uses for these images and underlying methods, such as authentication schemes or fraud detection. For example, the last URL found in Table 2 shows siftscience, an anti-fraud company, drawing the popular `fjord` pangram. Because the same image may be used for either purpose, and because ML-CB's pipeline takes ground truth from images, altering the label on those images would inappropriately change the label for all images. Instead, a deployed fingerprinting blocker could opt to allowlist specified `<script, domain>` pairs, allowing specific, approved websites to engage in canvas fingerprinting. We note that what constitutes appropriate or inappropriate fingerprinting requires human judgment.

6 Conclusion

In this paper, we presented ML-CB, a process and implementation for generating trained machine-learning models which may be integrated into browsers to provide "smart" tracking blocking on HTML5 canvas actions. To achieve this, we crawled roughly half a million websites and created a dataset of canvas-based actions. We analyze this dataset and discuss the web's overall use of the canvas, finding that rote blocking of the canvas would disproportionately block a substantial number of non-harmful canvas actions. We then apply a key insight—the images drawn to the canvas may be used as a proxy for "good" or "bad" canvas actions, allowing us to label hundreds of files (i.e., programs), related to thousands of webpages, with a single image. We use these labels to train supervised machine-learning classifiers, which perform adequately even in the adversarial case where website source code is heavily obfuscated and minified. In this way, ML-CB may be used to increase privacy online by thwarting one way devices are surreptitiously tracked across the web.

Acknowledgements.

This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors. We thank Steven M. Bellovin for comments on an earlier version of this paper.

References

[1] E. Zuckerman, "The internet's original sin," *The Atlantic*, vol. 14, August 2014. [Online]. Available: https://www.theatlantic.com/technology/archive/2014/08/advertising-is-the-internets-original-sin/376041/

[2] N. Bielova, "Web tracking technologies and protection mechanisms," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[3] G. Acar, C. Eubank, S. Englehardt, M. Juarez, A. Narayanan, and C. Diaz, "The web never forgets: Persistent tracking mechanisms in the wild," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014.

[4] J. R. Mayer and J. C. Mitchell, "Third-party web tracking: Policy and technology," in *2012 IEEE Symposium on Security and Privacy*, 2012.

[5] N. F. Awad and M. S. Krishnan, "The personalization privacy paradox: An empirical evaluation of information transparency and the willingness to be profiled online for personalization," *MIS Quarterly*, vol. 30, no. 1, pp. 13–28, March 2006.

[6] M. Taddicken, "The 'privacy paradox' in the social web: The impact of privacy concerns, individual characteristics, and the perceived social relevance on different forms of self-disclosure," *Journal of Computer-Mediated Communication*, vol. 19, no. 2, p. 248–273, January 2014.

[7] S. Fulton and J. Fulton, *HTML5 Canvas*. O'Reilly Media, Inc., 2011.

[8] K. Mowery and H. Shacham, "Pixel perfect: Fingerprinting canvas in HTML5," in *Proceedings of W2SP*, 2012.

[9] S. Wu, S. Li, Y. Cao, and N. Wang, "Rendered private: Making GLSL execution uniform to prevent WebGL-based browser fingerprinting," in *28th USENIX Security Symposium*, 2019.

[10] P. Laperdrix, N. Bielova, B. Baudry, and G. Avoine, "Browser fingerprinting: A survey," *ACM Transactions on the Web (TWEB)*, vol. 14, no. 2, pp. 1–33, 2020.

[11] S. Luangmaneerote, E. Zaluska, and L. Carr, "Survey of existing fingerprint countermeasures," in *2016 International Conference on Information Society (i-Society)*, 2016.

[12] L. N. Smith, "A disciplined approach to neural network hyper-parameters: Part 1 – learning rate, batch size, momentum, and weight decay," *US Naval Research Laboratory*, Technical Report 5510-026, 2018.

[13] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code," in *2010 17th Working*

Conference on Reverse Engineering, 2010.

[14] S. Clark, M. Blaze, and J. M. Smith, "Smearing fingerprints: Changing the game of web tracking with composite privacy," in *Cambridge International Workshop on Security Protocols*, 2015.

[15] M. Ikram, H. J. Asghar, M. A. Kaafar, A. Mahanti, and B. Krishnamurthy, "Towards seamless tracking-free web: Improved detection of trackers via one-class learning," *Proceedings on Privacy Enhancing Technologies*, vol. 2017, no. 1, pp. 79–99, 2017.

[16] T. Bujlow, V. Carela-Español, J. Solé-Pareta, and P. Barlet-Ros, "A survey on web tracking: Mechanisms, implications, and defenses," *Proceedings of the IEEE*, vol. 105, no. 8, pp. 1476–1510, August 2017.

[17] Electronic Frontier Foundation, "Panopticlcick," https://panopticlick.eff.org/.

[18] R. Upathilake, Y. Li, and A. Matrawy, "A classification of web browser fingerprinting techniques," in *2015 7th International Conference on New Technologies, Mobility and Security (NTMS)*, 2015.

[19] L. I. Millett, B. Friedman, and E. Felten, "Cookies and web browser design: Toward realizing informed consent online," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, 2001.

[20] O. Kulyk, A. Hilt, N. Gerber, and M. Volkamer, "'this website uses cookies': Users' perceptions and reactions to the cookie disclaimer," in *European Workshop on Usable Security (EuroUSEC) 2018*, April 2018.

[21] F. Marotta-Wurgler, "Does "notice and choice" disclosure regulation work? an empirical study of privacy policies," in *Michigan Law: Law and Economics Workshop*, 2015. [Online]. Available: https://perma.cc/GYN4-3YFA

[22] J. R. Reidenberg, N. C. Russell, A. Callen, S. Qasir, and T. Norton, "Privacy harms and the effectiveness of the notice and choice framework," *I/S: A Journal of Law and Policy for the Information Society*, pp. 485–524, 2014.

[23] N. Richards and W. Hartzog, "The pathologies of digital consent," *Washington University Law Review*, vol. 96, pp. 1461–1503, 2019.

[24] S. Englehardt, D. Reisman, C. Eubank, P. Zimmerman, J. Mayer, A. Narayanan, and E. W. Felten, "Cookies that give you away: The surveillance implications of web tracking," in *Proceedings of the 24th International Conference on World Wide Web*, 2015.

[25] M. Perry, E. Clark, S. Murdoch, and G. Koppen, "The design and implementation of the Tor Browser [draft]," June 2018, https://2019.www.torproject.org/projects/torbrowser/design/.

[26] Inform Action, "noscript," https://noscript.net.

[27] A. Macrina and E. Phetteplace, "The Tor Browser and intellectual freedom in the digital age," *Reference and User Services Quarterly*, vol. 54, no. 4, pp. 17–20, 2015.

[28] M. Piekarska, Y. Zhou, D. Strohmeier, and A. Raake, "Because we care: Privacy dashboard on FirefoxOS," *arXiv*, 2015. [Online]. Available: https://arxiv.org/abs/1506.04105

[29] C. E. Shannon, "A mathematical theory of communication," *Bell Syst. Tech.*, vol. 27, pp. 379–423, 1948. [Online]. Available: https://arxiv.org/abs/1506.04105

[30] Y. Cao, S. Li, E. Wijmans *et al.*, "(Cross-) browser fingerprinting via OS and hardware level features." in *Proceedings of the 2017 Network and Distributed System Security Symposium*, 2017.

[31] Mozilla, "HTMLCanvasElement.toDataURL()," https://developer.mozilla.org/en-US/docs/Web/API/HTMLCanvasElement/toDataURL.

[32] N. Reitinger, "Faces and fingers: Authentication," *Journal of High Technology Law*, vol. 20, no. 1, pp. 61–81, 2020.

[33] E. Bursztein, A. Malyshev, T. Pietraszek, and K. Thomas, "Picasso: Lightweight device class fingerprinting for web clients," in *Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2016.

[34] fingerprintJS, "FPJS - Valve," https://github.com/Valve/fingerprintjs2.

[35] antoinevastel, "Picasso based canvas fingerprinting," https://github.com/antoinevastel/picasso-like-canvas-fingerprinting.

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

[37] Mozilla, "OpenWPM," https://github.com/mozilla/OpenWPM.

[38] P. Laperdrix, "Browser fingerprinting: Exploring device diversity to augment authentification and build client-side countermeasures," *Cryptography and Security [cs.CR]. INSA de Rennes*, 2017. [Online]. Available: https://tel.archives-ouvertes.fr/tel-01729126/document

[39] N. Bielova, F. Besson, and T. Jensen, "Using JavaScript monitoring to prevent device fingerprinting," *ERCIM News*, vol. 106, July 2016. [Online]. Available: https://ercim-news.ercim.eu/images/stories/EN106/EN106-web.pdf

[40] G. Merzdovnik, M. Huber, D. Buhov, N. Nikiforakis, S. Neuner, M. Schmiedecker, and E. Weippl, "Block me if you can: A large-scale study of tracker-blocking tools," in *2017 IEEE Symposium on Security and Privacy*, 2017.

[41] Appodrome, "CanvasFingerprintBlock," https://chrome.google.com/webstore/detail/canvasfingerprintblock/ipmjngkmngdcdpmgmiebdmfbkcecdndc.

[42] C. F. Torres, H. Jonker, and S. Mauw, "FP-Block: Usable web privacy by controlling browser fingerprinting," in *European Symposium on Research in Computer Security*, 2015.

[43] N. Nikiforakis, W. Joosen, and B. Livshits, "Privaricator," in *Proceedings of the 24th International Conference on the World Wide Web*, 2015.

[44] A. ElBanna and N. Abdelbaki, "NONYM! ZER: Mitigation framework for browser fingerprinting," in *2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion*, 2019.

[45] P. Baumann, S. Katzenbeisser, M. Stopczynski, and E. Tews, "Disguised Chromium browser: Robust browser, flash and canvas fingerprinting protection," in *Proceedings of the 2016 ACM on Workshop on Privacy in the Electronic Society*, 2016.

[46] P. Laperdrix, B. Baudry, and V. Mishra, "FPRandom: Randomizing core browser objects to break advanced device fingerprinting techniques," in *9th International Symposium on Engineering Secure Software and Systems*, 2017.

[47] P. Laperdrix, W. Rudametkin, and B. Baudry, "Mitigating browser fingerprint tracking: Multi-level reconfiguration

and diversification," in *Proceedings of the IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2015.

[48] A. Datta, J. Lu, and M. C. Tschantz, "Evaluating anti-fingerprinting privacy enhancing technologies," in *The Web Conference*, 2019.

[49] A. Vastel, P. Laperdrix, W. Rudametkin, and R. Rouvoy, "Fp-Scanner: The privacy implications of browser fingerprint inconsistencies," in *27th USENIX Security Symposium*, 2018.

[50] The Tor Project, "Bug 6253: Add canvas image extraction prompt," https://gitweb.torproject.org/tor-browser.git/commit/?h=tor-browser-52.5.2esr-7.0-2&id=196354d7951a48b4e6f5309d2a8e46962fff9d5f.

[51] Disconnect, "Disconnect tracker protection," https://github.com/disconnectme/disconnect-tracking-protection.

[52] Wayback Machine, "ftc.gov, february 1, 2019," https://web.archive.org/web/20190201065632/https://www.ftc.gov/ and https://web.archive.org/web/20190201050345js_/https://gateway.foresee.com/code/19.6.6/fs.utils.js.

[53] N. Reitinger, "Strange bedfellows: Fingerprinting phenomena...or state.gov versus facebook.com," https://medium.freecodecamp.org/strange-bedfellows-fingerprinting-phenomena-or-state-gov-versus-facebook-com-8d123866e7df.

[54] A. Narayanan, January 2019, https://twitter.com/random_walker/status/1089897867458867200.

[55] Disconnect, "services.json," https://github.com/disconnectme/disconnect-tracking-protection/blob/master/services.json and https://perma.cc/C4GQ-XSUY.

[56] A. FaizKhademi, M. Zulkernine, and K. Weldemariam, "FPGuard: Detection and prevention of browser fingerprinting," in *IFIP Annual Conference on Data and Applications Security and Privacy*, 2015.

[57] WordPress.org, "#43264: WordPress emojis show up as browser fingerprinting and will be blocked in new versions of FireFox," https://core.trac.wordpress.org/ticket/43264.

[58] K. Boda, Á. M. Földes, G. G. Gulyás, and S. Imre, "User tracking on the web via cross-browser fingerprinting," in *Nordic Conference on Secure IT Systems*, 2011.

[59] A. Gómez-Boix, D. Frey, Y.-D. Bromberg, and B. Baudry, "A collaborative strategy for mitigating tracking through browser fingerprinting," in *Proceedings of the 6th ACM Workshop on Moving Target Defense*, 2019.

[60] U. Iqbal, S. Englehardt, and Z. Shafiq, "Fingerprinting the fingerprinters: Learning to detect browser fingerprinting behaviors," in *2021 IEEE Symposium on Security & Privacy*, 2021.

[61] V. Rizzo, S. Traverso, and M. Mellia, "Unveiling web fingerprinting in the wild via code mining and machine learning," *Proceedings on Privacy Enhancing Technologies*, vol. 1, pp. 43–63, 2021.

[62] 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, https://arxiv.org/pdf/2003.04463.pdf.

[63] C.-H. Hsiao, M. Cafarella, and S. Narayanasamy, "Using web corpus statistics for program analysis," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, 2014.

[64] S. Robertson, "Understanding inverse document frequency: On theoretical arguments for IDF," *Journal of Documentation*, vol. 60, no. 5, p. 503–520, 2004.

[65] W. S. Noble, "What is a support vector machine?" *Nature Biotechnology*, vol. 24, no. 12, p. 1565–1567, 2006.

[66] D. E. Knuth, "Semantics of context-free languages. mathematical systems theory," *Mathematical systems theory*, vol. 2, no. 2, pp. 127–145, 1968.

[67] J. McCarthy, "A formal description of a subset of ALGOL," Stanford University Department of Computer Science, Tech. Rep., 1964.

[68] J. R. Quinlan, *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1994.

[69] ——, "Induction of decision trees," *Machine learning*, vol. 1, no. 1, pp. 81–106, 1986.

[70] Selenium, "Selenium WebDriver," https://www.selenium.dev.

[71] Amazon Web Services, "Top sites in united states," https://www.alexa.com/topsites/countries/US.

[72] Mozilla, "webRequest," https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/webRequest.

[73] MDN Web Docs, "webRequest.onBeforeRequest," https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/webRequest/onBeforeRequest.

[74] PostgreSQL, "Postgresql: The world's most advanced open source relational database," https://www.postgresql.org.

[75] Browserleaks.com, "HTML5 Canvas Fingerprinting," https://browserleaks.com/canvas.

[76] Wayback Machine, "justuno.com, october 31, 2018," 2018, https://web.archive.org/web/20181031094820/cdn.justuno.com/mwgt_4.1.js?v=1.56.

[77] tld, "tld 0.12.2," https://pypi.org/project/tld/.

[78] Google Cloud, "Natural language," https://cloud.google.com/natural-language.

[79] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection," in *in Proceedings of the 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, 2014.

[80] V. Raychev, M. Vechev, and A. Krause, "Predicting program properties from 'Big Code'," in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2015.

[81] Requests, "Requests: HTTP for humans," https://requests.readthedocs.io/en/master/.

[82] brettlangdon, "Command line interface to http://jsnice.org," https://github.com/brettlangdon/jsnice.

[83] D. Koller and N. Friedman, *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009.

[84] M. Chahal, "Information retrieval using Jaccard similarity coefficient," *International Journal of Computer Trends and Technology (IJCTT)*, vol. 36, no. 3, 2016.

[85] textdistance, "textdistance 4.2.0," https://pypi.org/project/textdistance/.

[86] W. Koehler, "Web page change and persistence—Ai'm not four-year longitudinal study," *Journal of the American Society for Information Science and Technology*, vol. 53, no. 2, pp. 162–171, 2002.

[87] P. Warren, C. Boldyreff, and M. Munro, "The evolution of websites," in *Proceedings of the Seventh International Workshop on Program Comprehension*, 1999.

[88] W. Koehler, "An analysis of web page and web site constancy and permanence," *Journal of the American Society for Information Science*, vol. 50, no. 2, pp. 162–180, 1999.

[89] D. Herrmann, R. Wendolsky, and H. Federrath, "Website fingerprinting: Attacking popular privacy enhancing technologies with the multinomial naïve-Bayes classifier," in *Proceedings of the ACM Workshop on Cloud Computing Security*, 2009.

[90] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. V. Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, "You are what you include," in *Proceedings of the 2012 ACM SIGSAC Conference on Computer and Communications Security*, 2012.

[91] J. Pennington, R. Socher, and C. D. Manning, "GloVe: Global vectors for word representation," in *Empirical Methods in Natural Language Processing*, 2014.

[92] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[93] Stanford Vision Lab, Stanford University, and Princeton University, "ImageNet," http://www.image-net.org.

[94] Alex Clark and Contributors, "Pillow," https://pillow.readthedocs.io/en/stable/.

[95] fastai, "Data augmentation in computer vision," 2021, https://docs.fast.ai/vision.augment.html.

[96] X. Jin, Y. Chen, J. Dong, J. Feng, and S. Yan, "Collaborative layer-wise discriminative learning in deep neural networks," in *European Conference on Computer Vision*, 2016.

[97] E. G. Adagbasa, S. A. Adelabu, and T. W. Okello, "Application of deep learning with stratified k-fold for vegetation species discrimation in a protected mountainous region using sentinel-2 image," *Geocarto International*, pp. 1–21, 2019.

[98] N. Ghamrawi and A. McCallum, "Collective multi-label classification," in *Proceedings of the 14th ACM International Conference on Information and Knowledge Management*, 2005.

[99] M. Jaggi, "An equivalence between the lasso and support vector machines," in *Regularization, Optimization, Kernels, and Support Vector Machines*, J. A. Suykens, M. Signoretto, and A. Argyriou, Eds. CRC Press, 2014. [Online]. Available: https://arxiv.org/pdf/1303.1152.pdf

[100] L. Wu, S. C. Hoi, and N. Yu, "Semantics-preserving bag-of-words models and applications," *IEEE Transactions on Image Processing*, vol. 19, no. 7, pp. 1908–1920, 2010.

[101] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proceedings of the 3rd International Conference on Learning Representations*, 2014.

[102] P.-T. de Boer, D. P. Kroese, S. Mannor, and R. Y. Rubinstein, "A tutorial on the cross-entropy method," *Annals of Operations Research*, vol. 134, pp. 19–67, 2005.

[103] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. [Online]. Available: http://www.deeplearningbook.org

[104] X. Shi, Z. Chen, H. Wang, D.-Y. Yeung, W.-K. Wong, and W.-C. Woo, "Convolutional LSTM network: A machine learning approach for precipitation nowcasting," in *Advances in Neural Information Processing Systems*, 2015.

[105] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: Synthetic minority over-sampling technique," *Journal of Artificial Intelligence Research*, vol. 16, pp. 321–357, 2002.

[106] P. Skolka, C.-A. Staicu, and M. Pradel, "Anything to hide? Studying minified and obfuscated code in the web," in *The Web Conference*, 2019.

[107] V. Le Pochat, T. Van Goethem, S. Tajalizadehkhoob, M. Korczynski, and W. Joosen, "Tranco: A research-oriented top sites ranking hardened against manipulation," in *Proceedings of the 2019 Network and Distributed System Security Symposium*, 2019. [Online]. Available: https://tranco-list.eu

[108] C. Baraniuk, "Where will the ad versus ad blocker arms race end?" *Scientific American*, May 2018. [Online]. Available: https://www.scientificamerican.com/article/where-will-the-ad-versus-ad-blocker-arms-race-end

[109] JavaScript Obfuscator, https://obfuscator.io.

[110] kangax, "HTML minifier," https://kangax.github.io/html-minifier/.

[111] mankyd, "htmlmin," https://htmlmin.readthedocs.io/en/latest/.

[112] T. László and Ákos Kiss, "Obfuscating C++ programs via control flow flattening„" *Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica*, vol. 30, no. 1, pp. 3–19, 2009.

[113] A. Das, N. Borisov, G. Acar, and A. Pradeep, "The web's sixth sense: A study of scripts accessing smartphone sensors," *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[114] R. Masood, B. Z. H. Zhao, H. J. Asghar, and M. A. Kaafar, "Touch and you're trapp(ck)ed: Quantifying the uniqueness of touch gestures for tracking," *Proceedings on Privacy Enhancing Technologies*, vol. 2018, no. 2, pp. 122–142, 2018.

[115] R. Amos, G. Acar, E. Lucherini, M. Kshirsagar, A. Narayanan, and J. Mayer, "Privacy policies over time: Curation and analysis of a million-document dataset," 2020, https://arxiv.org/pdf/2008.09159.pdf.

[116] S. M. Bellovin, P. K. Dutta, and N. Reitinger, "Privacy and synthetic datasets," *Stanford Technology Law Review*, vol. 22, no. 1, pp. 1–52, 2019.

A Heuristics

1. The canvas element's height and width properties must not be set below 16 px.
2. Text must be written to [the] canvas with at least two colors or at least 10 distinct characters.
3. The script should not call the save, restore, or `addEventListener` methods of the rendering context.
4. The script extracts an image with `toDataURL` or with a single call to `getImageData` that specifies an area with a minimum size of 16 px × 16 px.

B JavaScript Obfuscator Tool Flags

- **control-flow-flattening** (i.e., a series of nested functions alter code structure [112])
- **dead-code-injection** (i.e., dead code is randomly placed throughout the JavaScript program)
- **compact** (i.e., one-line output code)
- **unicode-escape-sequence** (i.e., string converted to unicode escape sequence)
- **identifier-names-generator in hexadecimal** (i.e., renaming identifiers to hexadecmial, e.g., _0xabc123)
- **numbers-to-expressions** (i.e., converts numbers to functions, e.g., $1234 == -0xd93 + -0x10b4 + 0x41 * 0x67 + 0x84e * 0x3 + -0xff8;$)
- **rename-globals** (i.e., rename global variable names)
- **rename-properties** (i.e., rename property names)
- **rotate-string-array** (i.e., putting stings in array and rotating the array)
- **self-defending** (i.e., defeating measures to JavaScript beautification tools)
- **shuffle-string-array** (i.e., putting stings in array and shuffling the array)
- **split-strings** (i.e., splitting strings into separate chunks)
- **string-array** (i.e., removes string literals and puts them in an array, e.g., var m = "Hello World" becomes var m = _0x12c456[0x1])
- **transform-object-keys** (i.e., takes object keys and transforms them into functions)

C Representative Fingerprinting Images

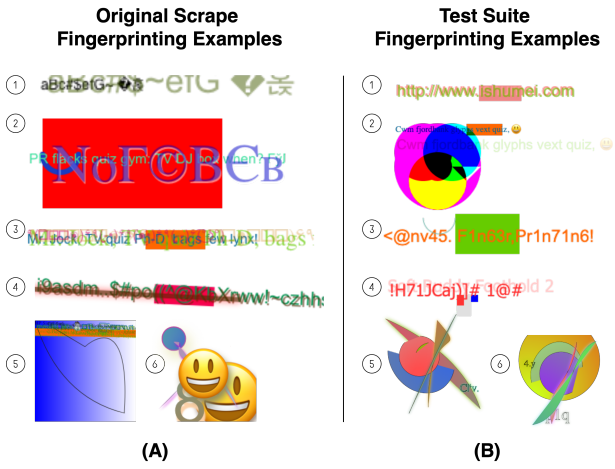

Fig. 6. Common canvas fingerprinting images found in the dataset. Images on the left (A) come from our original scrape, while images on the right (B) are from our test suite. Notably, the third image on the left (original scrape) was found on 68 websites (i.e., initiators), including: wsj.com, lenscrafters.com, and fnlondon.com. On the right (test suite), the sixth image was found on 48 websites in the test suite, including humanesocietyofyorkcounty.org, bloomberg.com/businessweek, and heart.org.

D Example Program Text

Fig. 7. Representative program text taken from the test suite (https://colorscheme.ru/js/canvas.min.js?e828175732). (A) shows the original plaintext version of the program in truncated form, focusing on the part of the program drawing to the canvas; (B) shows the plaintext snippet processed with jsNice, using the method discussed in Section 3.5.2; (C) shows a truncated version of the original plaintext, obfuscated with the Javascript Obfuscator Tool, using the settings mentioned in Section 4.3—as best as we could tell (with the help of jsNice), this sub-snippet represents the actions related to creating and filling the canvas with text, the full program was transformed into a 12,457 character, single-line string; and (D) takes the full obfuscated program from (C), processes it with jsNice, and shows only the truncated part provided in part (C), the full jsNice program is 483 lines long.

E Obfuscated Versus Obfuscated-then-jsNiceified Example

(A)

```
var
_0x28db=['\x63\x6e\x50\x74\x75','\x79\x63\x48\x44\x67','\x46\x47\x43\x45\x4e','\x63\x68\x61\x72\x43\x6f\x64\x65\x41\x74','\x54\x78\x7a\x45\x75','\x74\x79\x4c\x71\x4e','\x74\x6e\x4d\x47\x56','\x47\x72\x65\x73\x4c','\x41\x4a\x4f\x79\x4f','\x64\x6f\x66\x73\x76','\x50\x6c\x79\x76\x66','\x6d\x6c\x6d\x48\x51','\x5a\x6f\x54\x51\x52','\x51\x58\x43\x4f\x53','\x71\x76\x6e\x4e\x42','\x45\x76\x4b\x44\x68','\x73\x6c\x69\x63\x65','\x6a\x48\x59\x4c\x55','\x68\x70\x55\x75\x4b','\x77\x69\x64\x74\x68','\x65\x6e\x4f\x72\x69\x65\x6e\x74\x61\x74','\x42\x73\x78\x6a\x50','\x65\x5a\x41\x4b\x41','\x6e\x61\x6d\x65', ...]
```

(B)

```
'use strict';
/** @type {!Array} */
var _0x28db = ["... "canvas win", "Rbsdb", "evenodd", "alphabetic", "kfjle", "11pt Arial", "TjBop", "11pt no-re", "al-font-12", "rgba(102, ", "vguft",
"18pt Arial", "xTrWN", "Cwm fjordb", "ank glyphs", " vext quiz", "multiply", "prBsn", "rgb(255,0,", "255)", "dxAOz", "RrtdA", "rgb(255,25", "5,0)",
"NCxpH", "cTcFz", "wJngE", "lTTHH", "EXT_textur", "e_filter_a", ...]

...

  /**
   * @param {?} lagOffset
   * @return {?}
   */
  var render = function(lagOffset) {
    if (match[_0x1cec("0xbc")](match[_0x1cec("0x782")], match[_0x1cec("0x782")])) {
      /** @type {!Array} */
      var PL$86 = [];
      var c = document[_0x1cec("0x5da") + _0x1cec("0x5db")](match["WITNA"]);
      /** @type {number} */
      c[_0x1cec("0x751")] = 2E3;
      /** @type {number} */
      c[_0x1cec("0x783")] = 200;
      c[_0x1cec("0x5dd")][_0x1cec("0x784")] = match[_0x1cec("0xcd")];
      var urlParams = c[_0x1cec("0x785")]("2d");
      return . . .  urlParams["fillStyle"] = match[_0x1cec("0x78c")], urlParams["fillRect"](125, 1, 62, 20),  . . .

      urlParams["arc"](50, 50, 50, 0, match[_0x1cec("0x798")](2, Math["PI"]), true), urlParams[_0x1cec("0x799")](), urlParams[_0x1cec("0x79a")](),
      urlParams[_0x1cec("0x78d")] = match[_0x1cec("0xe4")], urlParams[_0x1cec("0x797")](), urlParams[_0x1cec("0x79b")](100, 50, 50, 0,
      match[_0x1cec("0x798")](2, Math["PI"]), true), . . .

      c["toDataURL"] && PL$86[_0x1cec("0x5cd")](match[_0x1cec("0x7b")](match[_0x1cec("0xe9")],  c["toDataURL"]())), PL$86; . . .
```

Fig. 8. Illustration of obfuscated plaintext (A) versus obfuscated-then-jsNiceified text (B). The example in (A) continues in hexadecimal form, while (B), with jsNice, re-introduces words like `Cwm fjord` and `toDataURL` in an instantiation section and, in a function named lagOffset, shows a similar structure to fingerprinting programs (Figure 7 (A)) and includes properties like `fillStyle` and `fillRect`.