FP-Fed: Privacy-Preserving Federated Detection of Browser Fingerprinting

Meenatchi Sundaram Muthu Selva Annamalai
University College London
meenatchi.annamalai.22@ucl.ac.uk

Igor Bilogrevic
Google
ibilogrevic@google.com

Emiliano De Cristofaro
University of California, Riverside
emilianodc@cs.ucr.edu

***Abstract*—Browser fingerprinting often provides an attractive alternative to third-party cookies for tracking users across the web. In fact, the increasing restrictions on third-party cookies placed by common web browsers and recent regulations like the GDPR may accelerate the transition. To counter browser fingerprinting, previous work proposed several techniques to detect its prevalence and severity. However, these rely on 1) centralized web crawls and/or 2) computationally intensive operations to extract and process signals (e.g., information-flow and static analysis).**

To address these limitations, we present FP-Fed, the first distributed system for browser fingerprinting detection. Using FP-Fed, users can collaboratively train on-device models based on their real browsing patterns, without sharing their training data with a central entity, by relying on Differentially Private Federated Learning (DP-FL). To demonstrate its feasibility and effectiveness, we evaluate FP-Fed's performance on a set of 18.3k popular websites with different privacy levels, numbers of participants, and features extracted from the scripts. Our experiments show that FP-Fed achieves reasonably high detection performance and can perform both training and inference efficiently, on-device, by only relying on runtime signals extracted from the execution trace, without requiring any resource-intensive operation.

I. Introduction

As users browse the web, they are often tracked across multiple unrelated websites by means of third-party cookies. Although this type of tracking can have valid use cases (e.g., ad conversion [47]), it is generally considered privacy invasive [79]. To protect users, browsers like Safari [83], Firefox [84], and Brave [11] have blocked or restricted third-party cookies by default. Chrome, the most popular web browser, will also start deprecating them in 2024 [50].

To circumvent these restrictions, trackers are turning to alternative methods like bounce tracking [82] and browser fingerprinting [75]. The latter uses client-side information to build unique user identifiers, typically via Javascript programs gathering device information, e.g., screen resolution, installed fonts, etc. [32]. This is then combined and hashed to generate a unique identifier for the user's browser, which remains stable over time regardless of the websites visited [70].

While browser fingerprinting can be used for acceptable purposes, e.g., web authentication [8, 49], bot [18, 33, 85] or fraud detection [38, 53], it largely represents a threat to user privacy [19, 31, 78, 81]. In fact, it can be even more intrusive than third-party cookies: the latter are easily detectable and can be cleared at any time, whereas browser fingerprinting is less transparent, and countermeasures often result in significant website breakage [10, 39]. Moreover, it can be effective even in incognito mode [7] and potentially track users for months [70].

Domains using browser fingerprinting have increased in recent years – from 519 in the top 1M websites in 2016 to 2,349 in the top 100k websites in 2019 [39]. Also considering that large-scale cross-site tracking will likely move away from cookie-based tracking, it is fair to assume fingerprinting will likely continue to grow in prevalence and severity.

Early research on browser fingerprinting facilitated the creation and manual curation of blocklists [25, 27, 28, 30] and basic heuristics [2, 32]. More recently, machine learning has been used to build fingerprinting detectors with high precision and recall [22, 37, 39]. These methods rely on one entity performing a large-scale crawl of the web (typically, top-ranked websites) to collect scripts, which are then labeled and used to train detection models. These techniques showcase the feasibility of detecting browser fingerprinting with machine learning-based approaches.

Limitations of Centralized Approaches. Alas, crawlers can rarely replicate human-like browsing behavior and interactions with a site, e.g., they are often identified by bot detectors, cannot operate beyond login and paywalls, or solve CAPTCHAs [3]. Moreover, the scripts' behavior may differ compared to real-world interactions, depending on the type of device, OS, etc. While it is possible in principle for a crawler to simulate these attributes (e.g., using different user-agent headers), in practice, it might be hard to do it correctly and, perhaps more importantly, to extensively cover the range of different devices, OS, etc. To this end, we also conduct a small-scale study on the top 300 domains from the Tranco ranking list and find that crawls involving real users (logging in, solving CAPTCHAs, etc.) can capture 3 times more fingerprinting scripts than automated/centralized ones proposed in previous work [32, 39]. In Appendix A, we provide an example of a script missed by an automated crawler but captured by user interaction.

Overall, training data built from centralized crawlers may fail to visit a non-negligible number of (potentially fingerprinting) websites, including top-ranked ones [39]. One other option could be gathering real-world observations from different users as they browse various websites; however, data collected from

Network and Distributed System Security (NDSS) Symposium 2024
26 February - 1 March 2024, San Diego, CA, USA
ISBN 1-891562-93-2
https://dx.doi.org/10.14722/ndss.2024.24360
www.ndss-symposium.org

websites might reveal sensitive information such as medical conditions [23], which could affect users' privacy. Alternatively, each user could collect their own data and train a detection model locally; while providing optimal privacy, this approach is unlikely to provide meaningful accuracy.

Federated Learning and its Challenges. As a result, we opt to build on Federated Learning (FL) [56], a collaborative learning approach seeking a reasonable privacy-utility compromise between local-only and fully centralized training. With FL, users train models locally but collaborate to build a global model, which lets them acquire knowledge from other users' data by only sharing (less sensitive) model updates.

However, it is not trivial to federate high-precision, high-recall classification algorithms, and in particular, to do so in an efficient, scalable manner, e.g., adapting to the federated setting existing classifiers [39] that rely on thousands of complex, hand-crafted, features extracted from each script. Also, executing complex algorithms may severely impact the browser's performance, especially when deploying to clients with a wide range of computational power and resource constraints.

Moreover, although sharing model updates rather than raw data is inherently less privacy-invasive, inference attacks are still possible that allow adversaries to learn sensitive information about users' training data [58, 65]. Therefore, it is imperative to provide rigorous privacy guarantees through the Differential Privacy framework [26]. Alas, this is likely to introduce a loss in model performance and thus has to be done in a way that preserves a reasonable privacy-utility tradeoff.

Contributions. This paper introduces FP-Fed (Browser Fingerprint Detection via Federated Learning) – to the best of our knowledge, the first distributed system for detecting fingerprinting in the wild. FP-Fed relies on Differentially Private Federated Learning (DP-FL) and achieves reasonably high accuracy, with minimal false positives, while providing formal privacy guarantees. To assess its feasibility and explore its deployment challenges, we analyze the performance of FP-Fed along several axes, including different levels of privacy, number of participants, and feature sets.

We evaluate FP-Fed on a dataset of 18.3k popular websites, finding that, with 1M participants, we achieve 0.86 Area Under the Precision-Recall Curve (AUPRC) while providing strong (central) differential privacy guarantees ($\varepsilon = 1$). With $\varepsilon = 10$, FP-Fed achieves a comparable performance to a fully centralized approach (0.95 vs. 0.97 AUPRC), and overall offers a significant improvement compared to each client only training on their local dataset (0.78 AUPRC).

Our experiments shed light on the optimal configurations balancing the practicality of deployment and detection performance. For instance, we show that we do not necessarily need extensive instrumentation of Javascript APIs or thousands of features, as done in previous work. In fact, a small set of 149 features is enough, even in the DP-FL setting.

Overall, while centralized techniques might miss websites and scripts due to bot detection techniques and user login requirements, the federated architecture of FP-Fed captures real-world browsing behavior and can detect fingerprinting more robustly while providing rigorous privacy guarantees.

II. Background

This section reviews background topics, namely, Federated Learning (FL) and Differential Privacy (DP); readers familiar with them can skip it without loss of continuity.

A. Federated Learning (FL)

Federated Learning (FL) is a decentralized learning approach where participants collaboratively train a machine learning model without sharing (possibly sensitive) training data with a central server [56]. Instead, they train local models on their individual datasets and only share model updates. The central server only sees and aggregates the model updates and propagates the global model to the participants.

There are different ways to instantiate FL; in this paper, we follow prior work [57, 63] and use the Federated Averaging (FedAvg) algorithm [56], which builds and iteratively updates a global model. In each round r, a subset of participants (C) is chosen from all participants by a central server. The server sends the aggregated global model parameters from the previous round θ^r_{global} to these participants, which initialize the local model with the global model parameters (i.e., participant i initializes $\theta^r_i = \theta^r_{global}$). Each participant i performs E local updates using a given optimization algorithm (typically Stochastic Gradient Descent) and returns the parameters of the resulting model, θ^{r+1}_i, to the server. Finally, the server averages the local models to get $\theta^{r+1}_{global} = \frac{1}{|C|}\sum_{i \in C} \theta^{r+1}_i$.

B. Differential Privacy (DP)

DP is the established framework to define algorithms resilient to adversarial inferences. It provides an unconditional upper bound on the privacy loss of individual data subjects from the output of an algorithm by introducing statistical noise [26].

Definition 1 (Differential Privacy (DP))**.** A randomized mechanism $\mathcal{M} : \mathcal{D} \to \mathcal{R}$ is (ε, δ)-differentially private if for any two neighboring datasets $d, d' \in \mathcal{D}$ and $S \subseteq \mathcal{R}$

$$\Pr[\mathcal{M}(d) \in S] \leq e^{\varepsilon} \Pr[\mathcal{M}(d') \in S] + \delta$$

The above definition leaves the definition of *neighboring datasets* to possibly depend on the setting, and thus it can vary [57]. The ε parameter (aka privacy budget) is a numerical value ranging from 0 to ∞ (lower values imply better privacy), representing the privacy loss due to the mechanism. The additional parameter δ is referred to as the "failure probability," i.e., the probability with which the mechanism fails to provide any privacy guarantees. Therefore, δ is set to be an asymptotically small number ($\approx 10^{-5}$).

C. DP in FL

In the context of FL, DP can be defined and applied in various ways, also depending on the trust assumptions in place.

Record-level vs. Participant-level DP. In FL, DP guarantees can hold at either record or participant level, depending on the definition of *neighboring datasets*. When each user contributes one record (aka a sample or a row), the dataset is defined as a collection of records. Therefore, neighboring datasets either add or remove a single record corresponding to whether

Algorithm 1 DP-FedAvg

function MAIN(*initial model* θ_0, *# rounds* R, *# participants* W, *sampling probability* q, *noise scale* z, *clipping parameter* S, *optimizer* OPT)
  $\theta^1_{global} \leftarrow \theta_0$
  $\sigma \leftarrow \frac{zS}{qW}$
  for round r from 1 to R **do**
    $P_r \leftarrow$ randomly select participants with probability q
    for participant $k \in P_r$ **do**
      $\Delta^{r+1}_k \leftarrow \text{LOCAL\_UPDATE}(k, \theta^r_{global}, S, \text{OPT})$
    end for
    $\theta^{r+1}_{global} \leftarrow \theta^r_{global} + \frac{1}{qW} \sum_{k \in P_r} \Delta^{r+1}_k + \mathcal{N}(0, \sigma^2 I)$
  end for
  return θ^{R+1}_{global}
end function

function LOCAL_UPDATE($k, \theta^r_{global}, S, \text{OPT}$)
  $\theta \leftarrow \theta^r_{global}$
  for local epoch i from 1 to E **do**
    $\theta \leftarrow \text{OPT}(\theta, \mathcal{D}_k)$ ▷ local update with OPT
    $\Delta \leftarrow \theta - \theta^r_{global}$
    $\theta \leftarrow \theta^r_{global} + \min(1, \frac{S}{||\Delta||_2}) \cdot \Delta$
  end for
  return $\theta - \theta^r_{global}$ ▷ already clipped
end function

a single user contributed their record to that dataset. Here, record-level guarantees [26] ensure the difficulty (bound by the privacy parameter ε) for an adversary to determine if a single record was included in the data analysis. Naturally, there are settings where each user contributes multiple records; e.g., in healthcare settings, each patient contributes multiple records corresponding to each hospital visit. Similarly, in FL, each user contributes all the records in their local dataset; here, participant-level guarantees [57] are necessary. Thus, neighboring datasets either add or remove all records that belong to a single user. In this case, participant-level DP ensures it is difficult (up to privacy parameter ε) for an adversary to determine if all the records contributed by a single user were included in the data analysis.

While in both settings, the DP guarantees correspond to whether a single user participated in the data analysis, the resulting definitions of datasets/neighboring datasets vary due to the amount of data contributed by each user.

Central, Local, and Distributed DP. Another difference in how DP can be integrated into FL is based on the trust placed on the server: 1) in Local DP (LDP) [67], each participant adds noise before sending updates to the server; 2) in Central DP (CDP) [35, 57], participants send updates without noise, and the server applies a differentially private aggregation algorithm.

LDP has the advantage that each participant's model update is (ε, δ)-differentially private; thus, not even the server is able to make inferences on them (with probability bounded by ε). At the same time, however, this also means that a large amount of local noise may be required, which could severely affect utility [44]. By contrast, in CDP, individual model updates are sent unperturbed to the server; the DP guarantees are primarily with respect to the aggregate model vis-à-vis the other participants. The server is trusted with the aggregated model coefficients but not the (sensitive) training data. Since noise is only added after aggregation, less noise is typically required, thus resulting in better model utility.

A possible alternative is to combine CDP with *secure aggregation* protocols. This setting, known as Distributed DP [42], uses CDP in that the total amount of noise added to the model updates is the same as with CDP. Each participant also adds a *small* amount of noise to their local model updates and encrypts them, using additively homomorphic encryption or secure multiparty computation, *before* sending them to the server. The server can then only decrypt the aggregate, but not the individual users' model updates. Alas, in practice, the implementation of secure aggregation protocols in production systems is not trivial, as the details of finite precision and modular summation arithmetic are often overlooked [42], noise has to be sampled from special discrete distributions [6, 36, 42], and the computational complexity of secure aggregation protocols tend to scale with the total number of participants (typically very large in FL).

Consequently, our work relies on CDP. Specifically, we follow prior work [57, 63] and use Algorithm 1, which guarantees that the output of the aggregation function at the server is indistinguishable (with probability bounded by ε) regardless of whether a given participant shared their local model updates in training (*participant-level DP*). Although the server is trusted with the model updates and the addition of noise during aggregation, this is a significantly weaker assumption than trusting the server with the raw training data [64]. Real-world deployments of differentially private FL often use CDP, such as Google's next-word prediction [55].

III. Browser Fingerprinting

We now introduce browser fingerprinting and our approach to collecting samples of fingerprinting scripts in the wild.

A. What is Browser Fingerprinting?

Browser fingerprinting is a stateless *online tracking* technique, usually deployed through Javascript, geared to build a unique identifier tied to a user's browser. Typically, fingerprinting scripts collect pieces of high-entropy device information that, combined together, yield unique and stable identifiers.

An example is shown in Script 1: the script builds a list of fonts installed on a device by rendering text in various fonts on a Canvas element and measuring the width of the rendered text—if the font is installed, its width will be different from when the font is not installed, in which case the user's device reverts to rendering the text in some default font. While many fonts are pre-installed on almost all devices, it is rare to have all possible fonts installed, and thus the combination of available fonts becomes virtually unique [29].

Although specific cases of fingerprinting – like the one discussed above – are well-known, there is no consensus on one specific definition of fingerprinting [39]. If anything, it is challenging to identify the *intent* behind the behavior and to determine whether it is truly used for fingerprinting purposes. For example, while the screen resolution can be a source of

```
// Canvas font fingerprinting script.
Fonts = ["monospace" , ... , "sans-serif"];

CanvasElem = document.createElement("canvas");
CanvasElem.width = "100";
CanvasElem.height = "100";
context = CanvasElem.getContext('2d');
FPDict = {};
for (i = 0; i < Fonts.length; i++)
{
  CanvasElem.font = Fonts[i];
  FPDict[Fonts[i]] = context.measureText("example").width;
}
```

Script 1: An example of a fingerprinting script [39].

high-entropy information used to potentially fingerprint users, it can also be used by modern reactive websites to display content using the appropriate layout and aspect ratio.

Thus, in this work, we follow the approach of FP-Inspector [39] to identify and refer to browser fingerprinting, which uses a conservative definition based on a set of well-known heuristics and signatures. While more details are provided in Section IV-C, in a nutshell, FP-Inspector [39] focuses on the four most prevalent forms of fingerprinting – namely, Canvas, Canvas Font, WebRTC, and AudioContext – and exclude the simple collection of properties from the `Navigator` and `Screen` APIs.

Although this might potentially miss some fingerprinting scripts and techniques (i.e., false negatives), it tends to minimize the chance of wrongly flagging scripts as fingerprinting (i.e., false positives). This is an important requirement of fingerprinting detection, as it directly affects its validity; when deployed in the wild, false positives might amount to falsely accusing organizations of doing fingerprinting and potentially not abiding by their stated privacy policies [24].

B. How to Collect Fingerprinting Scripts?

As mentioned, fingerprinting scripts access Javascript APIs, e.g., Canvas, WebRTC, and AudioContext, to retrieve high-entropy information about the device. Therefore, detection typically entails instrumenting browsers and collecting the content and execution traces of scripts (i.e., arguments and return values of APIs called) during web crawls. Our experiments adhere to this approach. While our work introduces a FL approach to train ML models to detect fingerprinting in-the-wild, we collected the execution traces and page content based on a web crawl. Note that this approach, while not fully representing the distributed browsing behavior of individual users, has been adapted to simulate different browsing patterns on the web (see Section V-A).

In contrast to previous work [32, 39], which mainly used extended versions of OpenWPM [32], we use Puppeteer[1] to collect contents and traces from scripts loaded by websites. OpenWPM uses Firefox, while Puppeteer uses Google Chrome; different browsers expose different Javascript APIs, which can, in turn, be exploited by different fingerprinting scripts. For example, although the Battery Status API was removed from Firefox since v52 (released in 2017), it is still supported in the latest version of Chrome M114. Furthermore, as Chrome is more widely used than Firefox, we assume it is more likely that fingerprinting scripts would rely on the API surface provided by Chrome.

Moreover, rather than extracting both static (e.g., Abstract Syntax Trees) and dynamic (e.g., execution traces) features, we only focus on dynamic ones—specifically, the number of times each API is called, the arguments passed to the API, and the return values of the API call. This follows previous work [66] showing that models built using static features are not robust, as obfuscation techniques can easily evade them. Additionally, while Javascript APIs can be easily instrumented in the browser to collect execution traces using extensions, performing more resource-intensive operations (such as parsing Abstract Syntax Trees and running clustering algorithms) is inefficient and impractical on resource-constrained devices. As a result, especially in a distributed setting, it is more practical to deploy classifiers based on dynamic features alone.

IV. The FP-Fed System

In this section, we introduce FP-Fed, a privacy-preserving federated learning system to detect browser fingerprinting collaboratively. We present an overview of the system along with its key components; then, we discuss in detail the individual steps involved in the operation of FP-Fed.

A. Overview

Figure 1 provides a high-level diagram of FP-Fed. The system works as follows:

1) FP-Fed participants visit websites according to their own interests and preferences. The participants run an instrumented browser (e.g., Chrome M114, which supports native API tracing) on a platform that supports FL (e.g., Android).
2) Before federated training begins, each participant's instrumented browser performs the following actions:
   a) Collects execution traces when specific (often high-entropy) monitored APIs are called;
   b) Extracts features for each script loaded on any visited website (see Section IV-B);
   c) Generates seed ground truth labels (fingerprinting/non-fingerprinting) for each script, according to a high-precision ground-truth heuristic (see Section IV-C);
   d) Participates in a pre-processing phase, sharing local summary statistics (mean and variance) of each feature with the server, which aggregates the statistics, adds DP noise, and shares them back with the participants. Finally, the browser scales each feature according to the global summary statistics (see Section IV-D).
3) At each round, the server selects a subset of participants to engage in that round of training. The server then sends the global model parameters from the previous round to these participants.
4) Participants instantiate a local model with the global model's parameters and update their local model with the data from the websites they visited.
5) Participants send the updated model parameters to the server.

[1] https://pptr.dev/

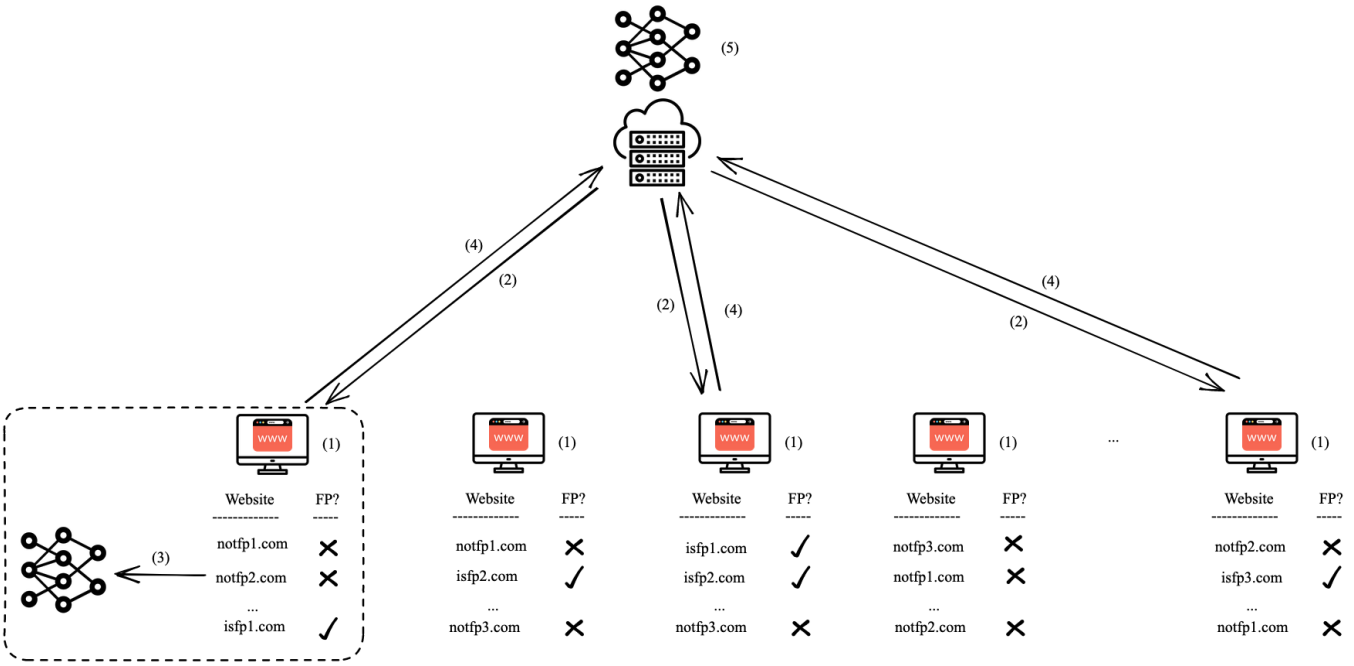

Fig. 1: An overview of the FP-Fed system: (1) Participants collect execution traces from the websites they visit. (2) At each round of training, the server selects a subset of participants and sends the previous round's global model parameters to them. (3) These participants train a local model based on data collected from websites they visited, and (4) send the local model updates to the server, (5) which aggregates them, adds differentially private noise, moves on to the next round, and repeats steps (2) to (5).

6) The server aggregates the updated model parameters, adds noise to satisfy DP, and generates the new global model parameters for the next round.
7) Steps 2) to 5) are repeated over multiple rounds until the global model converges; the global model parameters are then propagated to all participants to be used for on-device browser fingerprinting detection.

As the content served by websites and the scripts they load do not remain static over time, in theory, Steps 1) to 7) can be repeated regularly (e.g., once a month or once sufficient scripts are collected by participants) to ensure that the latest fingerprinting techniques remain detectable by the global model. However, in our experiments, we will only consider a single data collection phase for the sake of simplicity.

As discussed in Section I, the data collected by our system in Steps 2a) to 2c) might reveal sensitive information that needs to be protected. Therefore, FP-Fed needs to provide differential privacy guarantees (satisfying CDP) when aggregating such data. That means adding statistical noise to all model updates and statistics shared with participants, including during the pre-processing phase (namely, in Steps 1d and 5). We use DP's advanced composition theorem [45] and the moments accountant [1] to keep track of the overall privacy budget. In other words, FP-Fed provides strong privacy guarantees to the participants by formally bounding leakage.

Components. We can summarize the three main components involved in FP-Fed as follows:

- ***Participants.*** FP-Fed operates in a distributed setting where participants collaboratively build a browser fingerprinting detection model based on the websites they visit.
- ***Server.*** The aggregation server chooses the participants for each round, propagates the global model, and aggregates the local model updates (satisfying CDP). Although participants do not need to trust the server with their execution traces, etc., they trust it with (significantly less sensitive) model updates and that it will correctly perform differentially private aggregation.
- ***Model.*** While FP-Fed can potentially be used to train any neural network-based model, in this work we instantiate a simple logistic regression model, which can be seen as a neural network with no hidden layers, one output neuron, and the sigmoid activation function. We discuss this choice in detail in Section IV-E.

B. Feature Extraction

The features we extract from the execution traces collected by FP-Fed's instrumented browsers include: 1) the number of times 684 potential fingerprinting Javascript APIs are called, which we refer to as the *API call counts*, and 2) 830 custom features. In total, we consider 1,514 features (684 API call counts + 830 custom features) from each execution trace.

API Call Counts. Some APIs tap into well-known sources of high-entropy data (e.g., `Navigator.userAgent`). Others, such as `CanvasRenderingContext2D.measureText`, might do so only when called multiple times with a specific purpose. These "patterns" can be encoded into fingerprinting "signatures", which can then be used in a heuristic to tag scripts as fingerprinting if they appear to match any such signatures. FP-Inspector collects call counts from 500 APIs; however, since it uses Firefox, its crawler does not capture a number of deprecated and unimplemented APIs that are present in Chrome, e.g., `BatteryManager.level`. Using the same instrumentation

used by FP-Inspector, our Chrome-based crawler detects an additional 75 APIs accessed by scripts.

We also instrument 184 APIs available in Chrome that have been specifically flagged as "High Entropy APIs." These are natively traced by Chrome, instead of being separately instrumented by, e.g., an extension; thus, fingerprinting scripts will not be able to evade the instrumentation. Therefore, signals collected from those APIs are much more robust to fingerprinting evasion in JavaScript, as compared to APIs that are externally instrumented via extensions.

Custom Features. In addition to the API call counts, we instrument and use 830 custom hand-crafted features defined by [39]. Unlike API call counts, which simply involve counting the number of times an API is called, these features are processed from the arguments and return values of the calls. For instance, one such feature encodes whether the `WebGL2RenderingContext.fillStyle` API is set to fill the canvas with a gradient color (as opposed to a solid color). In other words, they often represent signatures that are present across many known fingerprinting scripts and thus are very useful in detecting browser fingerprinting.

Note that FP-Inspector originally extracts 2,128 such features; however, most of them could not be properly encoded when execution traces did not contain a call to the corresponding API. Rather, we focus on the 830 features that could be encoded even when the API was not called. In Table I, we report a sample of these custom features.

Furthermore, note that, even though these features process the raw execution traces heavily, information that might identify sensitive websites might still persist. Consider, for instance, a website that contains information about a sensitive medical condition – e.g., https://www.nhs.uk/conditions/hiv-and-aids/treatment/. This website might load a script that accesses a unique combination of APIs that will be visible from the API call counts. Alternatively, the script might make an API call with unique arguments, such as storing a first-party cookie with a unique string length which will be visible from the custom features. An adversary with access to these features might then be able to determine if a user has visited the sensitive website, thus leaking sensitive information about the user (e.g., the likelihood that the user has HIV).

C. Ground Truth

Unfortunately, there are no definite and readily available labels for Javascript scripts that can be used for fingerprinting detection, and creating them manually and at scale is hard. Therefore, we follow an approach similar to previous work [13, 39], using high-precision heuristics to generate binary labels (fingerprinting/non-fingerprinting). While these heuristics have high precision, they are typically narrowly defined to minimize false positives; as a result, they miss fingerprinting scripts. However, machine learning classifiers trained on high-precision heuristics are known to generalize over fingerprinting behaviors and detect previously unknown fingerprinting techniques. In prior work [39], machine learning classifiers were able to detect 26% more fingerprinting scripts than manually designed heuristics in the wild.

More precisely, our ground-truth labeling heuristic is taken from [39], which uses a high-precision fingerprinting definition that minimizes the false positive rate. This does not consider simple access to device information as fingerprinting; rather, only unwarranted accesses or aggressive calls made to well-known fingerprinting APIs are considered fingerprinting. Below, we review the four main types of fingerprinting identified in [39], along with their "definitions."

API	Operation
`CanvasRenderingContext2D.fillStyle`	is fill gradient?
`CanvasRenderingContext2D.textAlign`	returns `start`?
`CanvasRenderingContext2D.textBaseline`	returns `top`?
`CanvasRenderingContext2D.lineJoin`	returns `round`?
`WebGLRenderingContext.getExtension`	is first argument `EXT_blend_minmax`?
`WebGLRenderingContext.getExtension`	is first argument `WEBGL_draw_buffers`?
`WebGLRenderingContext.getExtension`	is first argument `WEBGL_lose_context`?
`WebGLRenderingContext.pixelStorei`	is second argument `4`?
`WebGLRenderingContext.getAttribLocation`	is second argument `r5`?
`WebGLRenderingContext.depthMask`	is first argument `False`?
`HTMLCanvasElement.getElementsByTagName`	is first argument `script`?
`Node.isConnected`	returns `False`?
`Document.getElementsByTagName`	is first argument `head`?
`HTMLCanvasElement.nodeName`	returns `canvas`?
`AnalyserNode.channelInterpretation`	returns `suspended`?
`AnalyserNode.channelCountMode`	returns `max`?
`OscillatorNode.type`	returns `triangle`?
`AudioContext.state`	returns `suspended`?
`RTCPeerConnection.iceGatheringState`	returns `complete`?
`RTCPeerConnection.signalingState`	returns `stable`?

TABLE I: Sample of custom features extracted from execution traces.

Canvas Fingerprinting: The differences in font rendering across devices are exploited to build a high-entropy identifier. Canvas fingerprinting is considered to be happening if:

1) Text is written to the `canvas` element using the `fillText` or `strokeText` method;
2) Style is applied with `fillStyle` or `strokeStyle` method;
3) `toDataURL` is called to extract the image from the `canvas`; and
4) `save`, `restore` or `addEventListener` methods are not called.

Canvas Font Fingerprinting: Scripts rely on accessing the list of fonts installed on a device. Canvas font fingerprinting is happening if:

1) `font` property of canvas element is set to more than 20 different fonts; and
2) `measureText` is called more than 20 times.

WebRTC Fingerprinting: Scripts use the access to candidate IP addresses of peers used by the WebRTC protocol [61]. The following actions define WebRTC fingerprinting:

1) `createDataChannel` or `createOffer` method is called on a WebRTC peer connection; and
2) `onicecandidate` or `localDescription` method is called.

AudioContext Fingerprinting: Scripts use differences in how audio signals are processed by different hardware. The following actions define AudioContext fingerprinting:

1) Either `createOscillator`, `createDynamicsCompressor`, `destination`, `startRendering`, or `oncomplete` is called.

D. Differentially Private Federated Pre-Processing

Before training, we need to normalize the extracted features to have a mean of 0 and a variance of 1. This is known to

Algorithm 2 DP-FedNorm

function MAIN(*# features* F, *# participants* W, *sampling probability* q, *noise scale* z, *mean clipping parameter* S_μ, *var clipping parameter* S_s)
  $\sigma_\mu \leftarrow \frac{zS_\mu}{qW}$
  $\sigma_s \leftarrow \frac{zS_s}{qW}$
  for feature f from 1 to F **do**
    $P_\mu^f \leftarrow$ randomly select participants with prob. q
    for participant $k \in P_\mu^f$ **do**
      $\mu_k^f \leftarrow \text{LOCAL\_MEAN}(k, f, S_\mu)$
    end for
    $\boldsymbol{\mu}_f \leftarrow \frac{1}{qW} \sum_{k \in P_\mu^f} \mu_k^f + \mathcal{N}(0, \sigma_\mu^2 I)$
    $P_s^f \leftarrow$ randomly select participants with prob. q
    for participant $k \in P_s^f$ **do**
      $s_k^f \leftarrow \text{LOCAL\_VAR}(k, f, S_s)$
    end for
    $\mathbf{s}_f \leftarrow \frac{1}{qW} \sum_{k \in P_s^f} s_k^f + \mathcal{N}(0, \sigma_s^2 I)$
  end for
  return $\boldsymbol{\mu}, \mathbf{s}$
end function

function LOCAL_MEAN(k, f, S_μ)
  $n \leftarrow |\mathcal{D}_k|$ ▷ number of rows
  $\mathbf{v}^f \leftarrow f^{th}$ column from $\mathcal{D}_k$
  $\mu = \frac{1}{n} \sum_{i=0}^{n} \mathbf{v}_i^f$
  return $\min(\mu, S_\mu)$ ▷ clip
end function

function LOCAL_VAR(k, f, S_s)
  $n \leftarrow |\mathcal{D}_k|$ ▷ number of rows
  $\mathbf{v}^f \leftarrow f^{th}$ column from $\mathcal{D}_k$
  $\mu = \frac{1}{n} \sum_{i=0}^{n} \mathbf{v}_i^f$
  $s = \frac{1}{n-1} \sum_{i=0}^{n} (\mathbf{v}_i - \mu)^2$
  return $\min(s, S_s)$ ▷ clip
end function

improve model convergence. In a centralized setting, the mean and variance of each feature can be trivially calculated by the data holder. Whereas in a federated setting, the mean and variance of each feature have to be aggregated in a distributed way from many participants, while satisfying differential privacy. To this end, we use Algorithm 2.

For each feature, f, after the mean ($\boldsymbol{\mu}_f$) and variance ($\mathbf{s}_f$) are calculated, each participant scales the vector of features extracted from each script ($\mathbf{v}^f$) by computing $\frac{\mathbf{v}^f - \boldsymbol{\mu}_f}{\mathbf{s}_f}$. This makes the (normalized) features have a mean of 0 and a variance of 1 throughout.

E. Training

Previous work [37, 39] in the centralized setting used Support Vector Machines (SVM) and Decision Trees. However, [37] focuses on the broader problem of "web tracking," building a balanced dataset where 57% of the scripts are tracking. By contrast, browser fingerprinting is a much more narrow definition: typically, less than 1% of all scripts are fingerprinting. Additionally, since SVMs do not perform well in problems with severe class imbalance [14], they are not a good fit for the problem of browser fingerprinting.

Moreover, decision trees have only been recently adapted to the federated setting. More precisely, Maddock et al. [54] recently introduced a framework to deconstruct the decision tree algorithm into components, with a number of settings that need to be fine-tuned to the specific application, e.g., discretizing continuous features, batching weight updates, etc.

Therefore, as adapting or designing novel DP-FL algorithms is not the focus of our work, we opt for a simple model architecture based on a logistic regression function, although FP-Fed can be instantiated with any algorithm. To ensure fast convergence, we also use the LBFGS optimizer instead of standard first-order gradient descent techniques, such as Stochastic Gradient Descent. To the best of our knowledge, we are the first to empirically show that second-order methods such as LBFGS can be used to train models with differential privacy successfully in the federated setting.

V. Experimental Evaluation

In this section, we discuss the setup of our experimental evaluation to shed light on the key factors impacting the performance of FP-Fed. We present our strategy to visit popular webpages and collect fingerprinting scripts. We then discuss how we distribute these scripts among FP-Fed participants. Finally, we introduce the different subsets of features we experiment with.

A. Dataset

Our first step is to collect execution traces. We opt to *simulate* a distributed setting by performing a single crawl, and then split the traces across different participants using different distributions. We do so for two reasons: 1) arguably, it would be exceedingly difficult to recruit and instrument large numbers of users and browsers, e.g., due to privacy, efficiency, cost, and coverage issues, and 2) simulating a distributed setting enables us to experiment with script and feature distributions to evaluate the impact of different distributions on the performance of the overall model (see Section V-B).

Popular websites. Our crawl strategy follows that of prior work [39], as we visit the homepages of 20k popular websites. We visit the top 10k sites from the Chrome User Experience Report (CrUX) [20] and randomly sample another 10k sites ranking between 10k and 100k. Previous studies have often used the Alexa ranking to sample and visit popular websites; we use the CrUX ranking instead since the Alexa ranking has become deprecated since May 2022 [9]. Moreover, recent research [74] suggests that the CrUX dataset provides a more accurate ranking than the alternatives.

Samples. Out of the 20k websites, we successfully visit 18,300 (91.5%). That is, our crawl fails to collect traces from 1,700 websites, with the overwhelming majority (64.3%) of them due to *HTTP 403 Forbidden* errors. This error is typically returned by websites requiring user login or bot detection scripts. Of the 18,300 websites loaded, we collect 181,633 unique scripts extracting 1,514 features each.

According to our high-precision ground-truth heuristic, 752 out of the 181,633 scripts (0.41%) are fingerprinting. We then

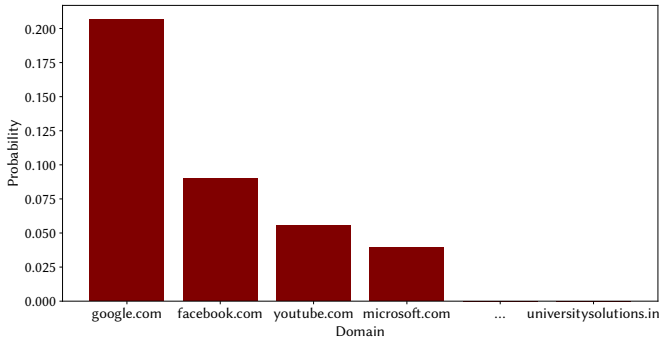

Fig. 2: Distribution of domains among participants

split the dataset into 80% training (145,307 scripts, 602 fingerprinting) and 20% testing (36,326 scripts, 150 fingerprinting).

Data Distribution. Next, we assign the training data to the participants in FP-Fed to simulate a realistic distributed browsing scenario. To do so, we build a distribution among domains based on the Tranco [69] list as showed in Figure 2.

Although the CrUX dataset used for the crawl does provide rankings for websites, these are coarse-grained (1k, 10k, 100k, etc.). Therefore, we assign the fine-grained Tranco ranks to the websites present on the CrUX list. The distribution follows the Zipf's law, which reflects the real-world observation that some websites are visited much more frequently than others [4, 21]. Thus, each participant k samples a fixed set of D URLs from the distribution and constructs her local list $\mathcal{D}_k$ of URLs, for which it stores all scripts that were loaded on each of these URLs from the training dataset.

B. Feature Sets

One key consideration of deploying FL in production is the limited computing capability available to participants [44]. While this is usually a key factor in deciding the size and complexity of models and training algorithms involved, in the context of browser fingerprinting, this consideration additionally impacts the features that the browser can feasibly extract. As described earlier, FP-Fed extracts two types of features: 1) API call counts and 2) custom features. The former simply involves attaching a counter to potential fingerprinting APIs, while the latter involves reading and processing arguments and return values of APIs, which might be computationally intensive and potentially raise privacy concerns.

As a result, we experiment with different smaller subsets of the 1,514 total features, which are more practical to deploy, and investigate their impact on model performance. Next, we define the four main feature sets we experiment with.

All. This feature set comprises all 1,514 features: 1) the 500 features collected in our crawl by instrumenting the full API surface instrumented by FP-Inspector [39], along with the additional 184 APIs not included in their work but currently present in Google Chrome, and 2) 830 custom features.

FP Inspector. This set includes 1,330 features: 1) the 500 API call counts and 2) the 830 custom hand-crafted features. This is the set of features considered by FP-Inspector [39].

JShelter. This set consists of 588 features: 96 API call counts and 492 custom features corresponding to the APIs instrumented by JShelter [40], a browser extension that, among other use cases, includes a fingerprinting detector module. We extract the API surface instrumented by JShelter from its source code using jsrestrictor.[2] As obtaining the exact list of custom features actually extracted by JShelter would require significant reverse engineering, we use all custom features corresponding to the API surface. While this may not be the exact feature set used by the extension to classify fingerprinting scripts, it provides a useful upper bound on the number of features real-world fingerprinting detectors typically have.

High Entropy. This set includes 109 features: no custom features and the 109 API call counts flagged as "high entropy" by Chromium.[3]

C. Metrics

To quantify the model's performance, we use the Area Under the Precision-Recall Curve (AUPRC) statistic. Rather than choosing a single threshold for classification and calculating precision and recall for that threshold, the AUPRC summarizes the model performance across various thresholds.

Thus, AUPRC is a more robust statistic that captures the effectiveness of the model as a whole. Also, we repeat and average all experiments over five runs.

VI. Results

We now present the results of our performance evaluation of FP-Fed across various settings. We start with assessing the impact of federated training – specifically, the number of participants – on model performance, without any differential privacy guarantees. We then measure FP-Fed's performance at various levels of privacy and for different feature sets used. In the process, we also introduce and experiment with an additional feature set called *Ext High Entropy* (Extended High Entropy). Next, we evaluate the performance improvements of using the feature normalization step. Moreover, we assess FP-Fed with respect to Non-IID distributions. Finally, we conduct a small-scale user study evaluating the effectiveness and computational overhead of performing a manual crawl using a prototype browser extension that deploys FP-Fed.

A. Non-DP Federated Training

We first focus on Non-DP training, where no noise is added to the feature normalization or the training algorithm ($z = 0$). Here, we compare the impact of federated training on model performance compared to a fully centralized setting baseline. The fully centralized setting corresponds to when a centralized crawler (e.g., FP-Inspector) can visit all websites present in our dataset. While this baseline is the closest comparison to prior work, we emphasize that this is not necessarily the best comparison as it assumes FP-Fed and prior work encounter the same number of FP scripts, which may not necessarily be true in practice (see Section VI-G). Figure 3 plots the model performance for an increasing number of participants

[2] https://github.com/polcak/jsrestrictor/blob/master/common/fp_config/wrappers-lvl_0_1.json

[3] https://github.com/chromium/chromium/blob/aae7191b27cef1f097b23e7742afb4895ec6a9d3/docs/privacy_budget/privacy_budget_instrumentation.md?plain=1#L196

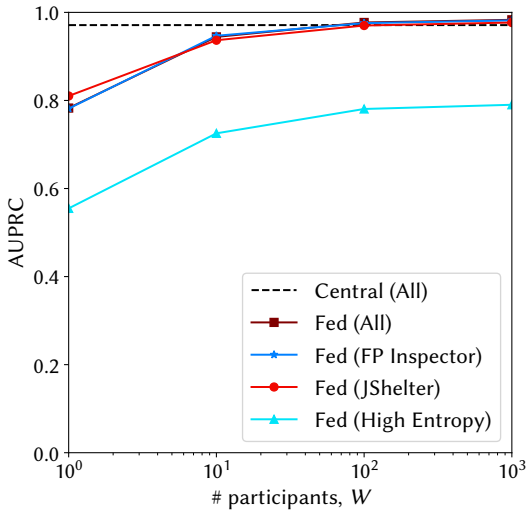

Fig. 3: Model performance (AUPRC) vs. number of participants (W) for various feature sets.

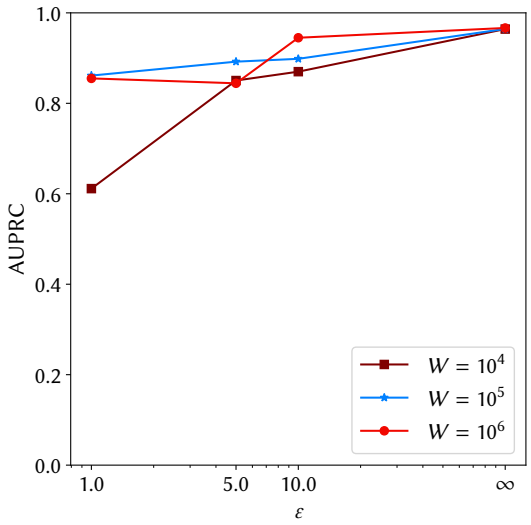

Fig. 4: Model performance (AUPRC) vs. the privacy parameter (ε) for an increasing number of participants (W).

with various feature sets. In this setting, all participants train on each round ($q = 1$).

First, we observe that FL significantly improves model accuracy, across the board, compared to local training, which is the setting with $W = 1$. Specifically, when all features are available to the classifier, AUPRC improves by 25.5% when 1,000 participants are training with FL compared to when each participant only trains on their local dataset.

Second, regardless of the feature set, already with 100 participants, the FL model achieves optimal performance (AUPRC 0.98 when all features are available); in fact, that only marginally improves with more than 100 participants, and thus we cut our plot at $W = 10^3$. This is an important observation from a deployment perspective, as the number of participants available at each round of training is typically much lower than the overall number of participants.

Third, models trained with *All*, *FP Inspector*, and *JShelter* feature sets all perform close to the centralized baseline, which is trained on all the scripts in the training dataset. This suggests that having access to the additional API call counts for the APIs present in Google Chrome (*All*) does not remarkably improve model performance compared to only using the APIs considered by FP-Inspector [39] (*FP Inspector*). However, this might be a limitation of the high-precision ground truth heuristic used in this work, which specifically does not include certain types of fingerprinting, e.g., Battery API (we discuss this further in Section VIII-E). These types of fingerprinting will, by definition, be missed by the *FP Inspector* feature set.

Finally, even in the Non-DP setting, detecting fingerprinting based on only the API call counts of APIs natively traced by Google Chrome (*High Entropy*) is not particularly effective. Even when the number of participants increases ($W = 10^3$), with this feature set, AUPRC does not go over 0.8.

Overall, this first set of experiments indicates that FL based on logistic regression shows great promise with respect to browser fingerprinting detection – even with a relatively small number of participants, detection performance matches that of centralized learning.

B. Impact of the Privacy Parameter (ε)

We now evaluate the performance of FP-Fed when adding differentially private noise. We experiment with three different levels of privacy ($\varepsilon \in \{1.0, 5.0, 10.0\}$), which we refer to, respectively, as high, moderate, and low privacy. We also include the Non-DP results ($\varepsilon = \infty$) for context. In each round, approximately 100 participants are chosen, using Poisson random sampling, from the large pool of participants ($q = 100/W$). We assume all features are available to the classifier. Figure 4 reports model performance with various levels of privacy and varying numbers of participants.

With a small number of participants ($W = 10^4$), FP-Fed only achieves AUPRC below 0.7, especially, at high levels of privacy ($\varepsilon = 1.0$). However, when the number of participants increases to $W = 10^5$, the model starts to perform better, with AUPRC above 0.8 even at high privacy levels ($\varepsilon = 1.0$). When the number of participants is high ($W = 10^6$), the model performance without DP can be recovered albeit only at a low level of privacy ($\varepsilon = 10.0$). Additionally, when the number of participants is large ($W \geq 10^5$), the model performs similarly (within 2 standard deviations) at high and moderate privacy levels ($\varepsilon = \{1.0, 5.0\}$). Note that this is not the number of participants per round but the total number of participants available. This confirms that, as it is common in DP-FL, having more than a handful of participants is imperative to improve model performance, even if not all of them participate in the training each round due to privacy amplification by sampling. In other words, for the same privacy level, the more participants exist, the less noise needs to be added.

Overall, we show that FP-Fed achieves acceptable performance (AUPRC ≥ 0.8) even with high privacy guarantees ($\varepsilon = 1.0$) when enough participants are present ($W \geq 10^5$).

C. Impact of the Feature Sets

Next, we evaluate the feasibility of training lighter-weight classifiers with DP. Once again, ≈ 100 participants are chosen for each round, using Poisson random sampling. However, the total number of participants is now fixed at $W = 10^6$. In

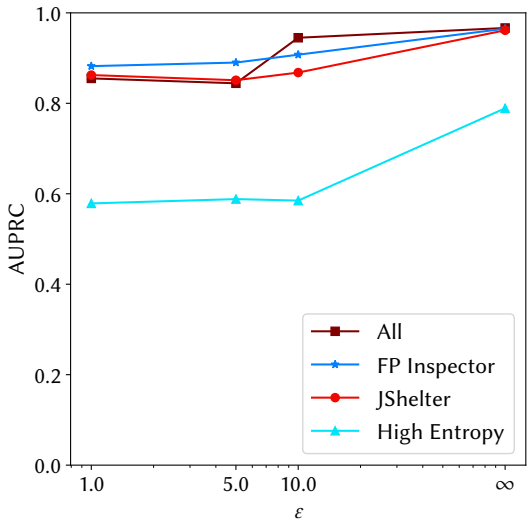

Fig. 5: Model performance (AUPRC) with different feature sets for increasing levels of privacy.

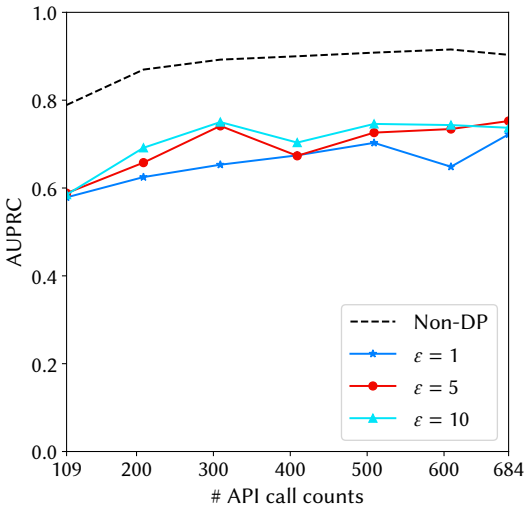

Fig. 6: Model performance (AUPRC) with an increasing number of API call counts only available to the model.

Figure 5, we plot the model performance at various levels of privacy for different feature sets.

We observe that when enough features are available (*All*, *FP Inspector*, and *JShelter*), the model performs well even at high privacy levels. Interestingly, at moderate to high privacy levels, the *JShelter* feature set has comparable performance (within 2 standard deviations) to the *FP Inspector* and *All* feature sets even though it only contains 40% of the features present in the *All* feature set. This is most likely due to the differentially private feature normalization step used in FP-Fed. Since the noise added to the mean and variance of each feature scales with the number of features used by the classifier, having fewer but more informative features can be enough to recover the model performance compared to having more, potentially uninformative features.

Also, the model continues to perform poorly when limited to the *High Entropy* feature set under differentially private training. In fact, even at a low level of privacy ($\varepsilon = 10.0$), the model performance is very poor (AUPRC < 0.6), which suggests that the model has little to no utility at any reasonable privacy level.

D. Extended High Entropy Feature Set

One of the main objectives of FP-Fed is to build a lightweight system that does not depend on complex hand-crafted features that are 1) hard to extract on the fly while executing scripts and 2) not robust to new types of fingerprinting that might emerge. However, our experiments thus far show that the API call counts of APIs natively instrumented by Chrome (*High Entropy* feature set) do not really yield good model performance. Therefore, it remains an open question to determine how many features FP-Fed requires to perform well. More precisely, we set to answer two main questions: 1) how many APIs and 2) how many custom features are needed for FP-Fed to reliably detect fingerprinting scripts?

First, we sort the features extracted by our crawl according to the *feature importance score*, which quantifies the impact each feature has on the final classification result. That is, features with higher scores are more informative and correspondingly are more important to include in a minimal feature set compared to features with lower scores. Since FP-Fed includes a feature normalization step, the weights of a trained classifier are unaffected by the scale of each feature. Therefore, we use the absolute weights of the trained classifier as our feature importance score. After sorting according to feature importance, we add additional features to the *High Entropy* feature set (more informative features are added first) and plot the model performance against the number of features added.

Adding API Call Counts. Figure 6 plots the model performance as the number of API call counts available to the model increases from 109 (*High Entropy* feature set) to 684 (total number of API call counts captured by the crawl) at various privacy levels. We set the total number of participants to $W = 10^6$ as before. Regardless of the number of API call counts available to the model, the performance only improves marginally and remains low (AUPRC < 0.8). In fact, even when there is no DP noise added, there is a significant drop in model performance of 6.8% from a model with access to all features, which answers the first question: no number of APIs (for which only call counts are collected) by themselves is enough for the model to reliably detect browser fingerprinting.

Adding API Call Counts & Custom Features. Figure 7 plots the model performance as the number of (potentially custom) features increases from 109 (*High Entropy* feature set, API call counts only), in 5 small increments of 20 features, to 209. (We stop after adding 100 additional features, as we find an optimal configuration where the model performance improves significantly after adding a small number of features.)

Once again, we set $W = 10^6$. When $\varepsilon \geq 5.0$, the performance of the model improves with the number of features available, plateauing after 20 additional features are added to the *High Entropy* feature set. For $\varepsilon = 1.0$ on the other hand, AUPRC improves until 40 additional features are added, after which performance momentarily drops. This is most likely due to the differentially private feature normalization step, where the number of features impacts the amount of noise in training, therefore causing a tradeoff between adding more (informative)

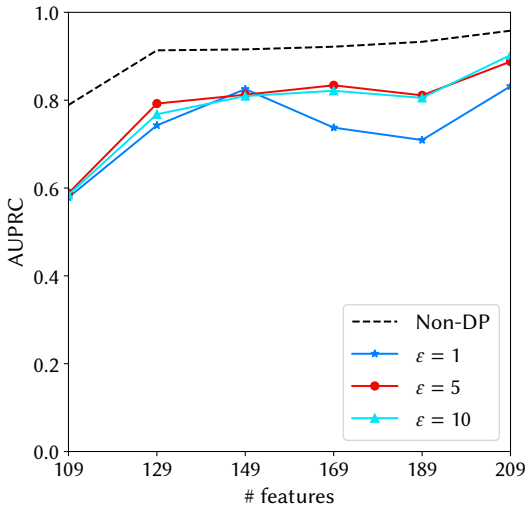

Fig. 7: Model performance (AUPRC) with an increasing number of (possibly custom) features available to the model.

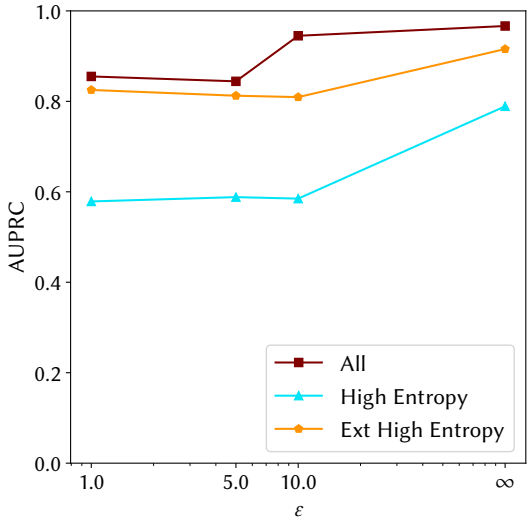

Fig. 8: Model performance (AUPRC) vs ε for various feature sets.

features and adding more noise in training. In general, across all privacy levels, we find that 40 additional features provide an optimal tradeoff between adding a minimal number of features and achieving good model performance. More precisely, these 40 features consist of 17 API call counts and 23 custom features. This shows that to achieve good model performance, custom features are indeed necessary—although a minimal set of 23 out of the original 830 features seems to be enough.

Among these 40 additional features, we find the `BatteryManager.level` API call count to be highly informative in detecting browser fingerprinting behavior. Note that Battery API was not used in defining the high-precision ground truth heuristic. Rather, we find that it is often used *together* with other APIs used by the heuristic.

The *Ext High Entropy* set. We define the union of the *High Entropy* feature set and the 40 additional features identified above as the Extended High Entropy (*Ext High Entropy*) feature set. In Figure 8, we compare the model performance with this set to *All* and *High Entropy* feature sets at various

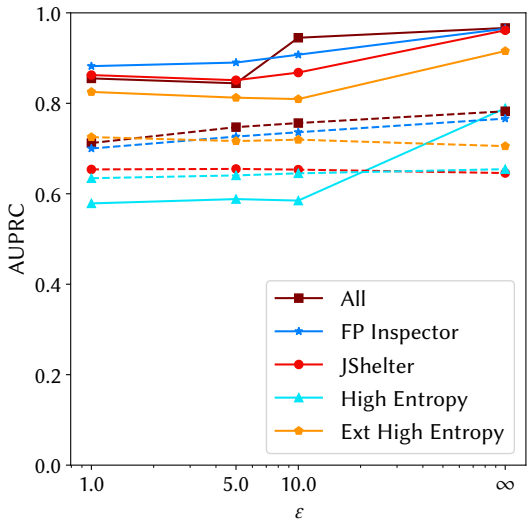

Fig. 9: Model performance (AUPRC) vs ε for various feature sets. The dotted lines represent results without feature normalization.

levels of privacy. As expected, *Ext High Entropy* yields much better performance at all levels of privacy than *High Entropy*. Furthermore, as observed with the *JShelter* feature set earlier, at a high privacy level ($\varepsilon = 1.0$), *Ext High Entropy* performs comparably to *All* (most likely due to feature selection) with only 9.8% of the features available to the former.

E. Impact of Feature Normalization

To the best of our knowledge, ours is the first work to use differentially private feature normalization in the FL setting. Therefore, we also investigate its effects on performance. To that end, Figure 9 plots the model performance at various levels of privacy for different feature sets. Specifically, we plot the model performances when FP-Fed is run with (in solid lines) and without (in dotted lines) the feature normalization step.

First, except for a few settings (*High Entropy* feature set, $\varepsilon \leq 10.0$), adding the differentially private feature normalization step always improves model performance. However, this is a specific setting where the model has access to very little, possibly uninformative features; thus, model training is easily impacted by the addition of even small amounts of noise.

When the model has access to richer custom features, it always performs appreciably better due to feature normalization. Specifically, at a high privacy level ($\varepsilon = 1.0$), model performance improves by up to 20.8% (*JShelter* feature set).

Consequently, we believe that feature pre-processing can indeed be an important step when training with DP. Allocating privacy budgets for feature pre-processing steps, such as normalization, can be advantageous, even though this may come at the cost of higher noise at each round of training.

F. Impact of Non-IID distributions

So far, participants have drawn scripts from domains according to the Tranco ranking, thus resulting in Independent and Identically Distributed (IID) training data. However, in practice, training data might be distributed in more complex ways, which might impact the performance of FP-Fed.

%Participants with Limited Knowldege	Non-IIDness Score
0	1.00
50	6.50
100	9.05

TABLE II: Non-IIDness scores vs. percentage of participants with limited knowledge, with 0% corresponding to an IID distribution and 100% to an extreme non-IID distribution.

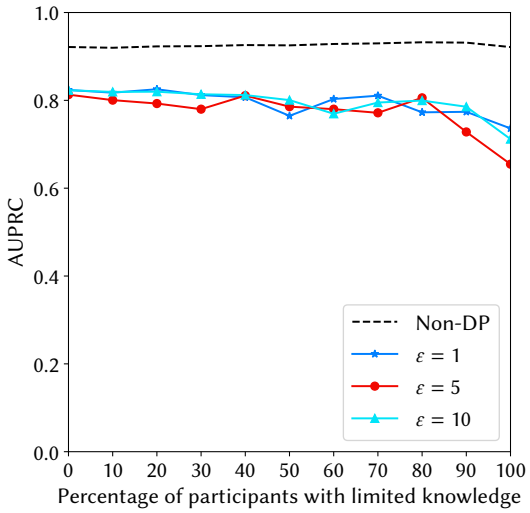

Fig. 10: Model performance (AUPRC) vs. various percentages of participants with limited knowledge.

Consequently, we stress-test FP-Fed with respect to increasingly Non-IID distributions by introducing *"Participants with Limited Knowledge"*, following recent work by Naseri et al. [63]. These only have access to at most one type of fingerprinting script (Canvas, Canvas Font, Audio, or WebRTC), thus yielding a different distribution of fingerprinting scripts encountered by each participant. By varying the percentage of these participants in FP-Fed, we vary the Non-IIDness of the distributions and quantify "how Non-IID" the resulting distributions are using the Non-IIDness score from [64]; please refer to Appendix B for details on how the score is computed. We report the Non-IIDness scores for three settings in Table II, which confirms that the Non-IIDness level does increase with increasing numbers of participants with limited knowledge.

We then fix the number of participants to $W = 10^6$, use the *Ext High Entropy* feature set, and plot the corresponding AUPRC scores at various privacy levels in Figure 10. As expected, the model performance drops with more participants with limited knowledge. However, the drop is pronounced only when more than 80% of participants have limited knowledge (with $\varepsilon = 5$), which is highly unlikely in practice. Note that, although $\varepsilon = 1$ appears to yield higher AUPRC than $\varepsilon = 5$ with 100% limited-knowledge participants, the scores are actually within $\pm$ standard deviation of each other. Therefore, we conclude that FP-Fed is reasonably resistant to Non-IID distributions which might be encountered in real-world settings.

G. Prototype Deployment

To evaluate the deployability of FP-Fed, we built a prototype Chrome extension that performs the data collection steps (Steps 2a to 2c). These steps have the greatest impact on browser performance as every call to the Javascript APIs from every script has to be instrumented and captured, while the machine learning steps might only happen once a day.

Type of FP	Manual	Basic Auto	Advanced Auto
Canvas	189	49	40
Canvas Font	8	8	5
Audio	32	27	21
WebRTC	49	10	8
Total	220	54	43

TABLE III: Number of fingerprinting scripts captured by manual vs. automated crawls. *NB:* The same script can perform multiple types of fingerprinting.

Effectiveness. We manually visit the top 300 domains from the Tranco ranking, excluding domains that do not expose a website (e.g., content delivery networks) and adult entertainment websites. In this manual crawl, we imitate real user interactions by logging in using a temporary Google account whenever possible, solving CAPTCHAs, and consenting to all cookie notices. We then compare the number of fingerprinting scripts encountered by the manual crawl with that of two automated crawlers (a basic and an advanced one). Both automated crawlers use the same Puppeteer architecture used in the rest of the paper. However, the advanced crawler uses the *puppeteer-extra-stealth-plugin* [15] commonly used in prior work [41, 71] that includes bot detection evasion techniques that are continuously updated to remain highly effective. By closely mimicking human-operated browsers [41], the plugin enables the browser to appear legitimate, and crucially bypass bot detectors that might otherwise prevent a crawler from visiting a website.

The difference in the number of fingerprinting scripts collected by each crawl technique is reported in Table III. This showcases the advantage of FP-Fed: when a real user logs in and goes through the authentication flow, the system captures 3.07× more fingerprinting scripts than the automated crawlers. Additionally, although the advanced crawler encounters slightly fewer fingerprinting scripts than the basic counterpart, we believe this may be because bot detection scripts may only fingerprint suspicious activity. Since the crawler deploys evasion techniques against bot detection, it may not appear suspicious enough to be fingerprinted as often as the basic crawler, thus resulting in this discrepancy.

Computational Overhead. Finally, we compare the computational overhead of deploying FP-Fed using the lighthouse performance metrics, which are widely used to evaluate the performance of webpages.[4] The metric produces an overall "score" between 0 and 1, which is the weighted average of several factors, such as the time taken for the first element to be painted and for the webpage to be fully interactive. We repeat the top-300 crawl with and without the prototype extension and evaluate the difference in the overall "score" assigned by the metric. We find that there was a mean performance drop of 0.0018 and a maximum drop of 0.078 when the extension is enabled, which shows that FP-Fed introduces a negligible computational overhead.

[4] https://github.com/GoogleChrome/lighthouse

VII. Related Work

In this section, we review prior work on browser fingerprinting and Federated Learning (FL).

A. Browser Fingerprinting

Early research on detecting browser fingerprinting mainly resulted in the creation and manual curation of blocklists, managed by vendors such as Disconnect [25], EasyList [27], EasyPrivacy [28], and PrivacyBadger [30]. However, maintaining these lists can be difficult due to constantly changing Web APIs and fingerprinting techniques; thus, more sophisticated methods have been proposed, ones that rely on hand-crafted heuristics and machine learning.

Heuristics-based detection. Acar et al. [3] and Roesner et al. [73] use a combination of heuristics and manual investigations to detect fingerprinting in the wild. In follow-up work, Acar et al. [2] introduce the first heuristic to detect canvas fingerprinting automatically, based on the arguments and return values of `fillText`, `strokeText`, and `toDataURL`. Englehardt and Narayanan [32] expand on these heuristics to detect three additional types of fingerprinting: Canvas Font, WebRTC, and AudioContext. This heuristic was also used by Iqbal et al. [39] and Das et al. [22]. The former also note that these heuristics often have to be defined narrowly, thus leading to fingerprinting scripts potentially being missed. Additionally, maintaining these heuristics in the face of a constantly evolving web platform can be difficult.

Machine Learning-based detection. Ikram et al. [37] use a one-class SVM to detect browser fingerprinting scripts using static features extracted from the Javascript code. Iqbal et al. [39] build a Decision Tree classifier that used features based on both dynamic (execution traces) and static (abstract syntax trees) analysis. However, as both studies trained classifiers on a centralized dataset, they might potentially miss webpages that only load fingerprinting scripts *after* waiting for user interaction or after a user has logged in [3].

Other mitigation strategies. Extensions like Canvas Defender [62] and privacy-focused browsers like Tor [48] and Brave [17] address browser fingerprinting by either normalizing Javascript APIs to return exactly the same value for all devices, or by randomizing the outputs of potential fingerprinting APIs. However, in almost all cases, this leads to some amount of website breakage and loss in functionality [39]. For example, Tor keeps the browser window at a standard size to prevent it from being used for fingerprinting, thus disabling the ability for users to maximize it to fit their screens.

B. DP in FL

DP is being increasingly integrated into FL applications, ranging from next-word prediction [57] to medical image analysis [5], and spam classification [68]. As discussed in Section II-C, the DP variants used in FL can be distinguished as Local DP (LDP) or Central DP (CDP). While LDP has been deployed in FL applications [64, 76, 77], it often reduces model performance even at moderately large privacy parameters. On the other hand, CDP has been more successfully applied to FL [5, 57, 63, 64] and has been deployed in large-scale production systems [43]. However, CDP requires the users to trust the aggregation server with their raw model updates and to aggregate and add differentially private noise correctly. Relaxing this trust assumption can make inference attacks possible [58, 65].

In our work, we chose the CDP approach for FL as this is the first work applying federated learning to the problem of browser fingerprinting detection. Providing formal DP guarantees in this setting is already challenging due to the significant class imbalance between fingerprinting and non-fingerprinting scripts. Therefore, we focus on experimenting with and adapting DP-FL techniques rather than introducing novel DP release algorithms or FL schemes.

C. FL for Security Tasks

Previous work has also used FL in security-oriented applications, e.g., to detect or predict security events [51, 52, 63]. This makes sense as it is often useful to have multiple data contributors, given that these are often relatively rare events. However, as data could be sensitive, data owners may not be willing to openly share them with a central repository. Therefore, FL has been used to fill this gap, enabling more accurate machine learning models to be built without compromising the privacy of the data owners.

For instance, Cerberus [63] uses FL to train a recurrent neural network (RNN) model to predict future security events based on past events contributed by participating organizations. DeepFed [51] is a framework that collaboratively trains a deep neural network to detect intrusions into cyber-physical systems (CPSs). Furthermore, Liu et al. [52] present a framework that collaboratively trains an attention-based deep neural network model to detect anomalies in Industrial Internet of Things (IoT) devices. Additionally, FL has also been used to detect malware and intrusions into IoT and mobile devices [34, 46, 59, 72].

VIII. Discussion & Conclusion

A. Summary

In this paper, we experimented with using Differentially Private Federated Learning (DP-FL) in the context of browser fingerprinting detection. We introduced and evaluated FP-Fed, a federated system to detect fingerprinting based on the collective browsing behavior of many users while providing formal differential privacy guarantees. We collected fingerprinting scripts in the wild using Puppeteer, Google Chrome, and the CrUX website ranking. This allowed us to capture more APIs than prior work and more accurately characterize the prevalence of browser fingerprinting on popular websites.

We conducted several experiments evaluating the impact of different feature sets and privacy levels, aiming to assess the impact of real-world constraints on model performance. We also developed a model based on a minimal feature set comprising only 149 API call counts and 23 custom features, which achieves AUPRC above 0.8 even at high privacy levels ($\varepsilon = 1.0$). This is a significantly smaller feature set compared to prior work that used 500 API call counts and 2,128 custom features and is thus far more practical for real-world deployment on end-user devices.

B. Privacy & Robustness

As mentioned, from a security point of view, there are two main challenges when using FL in real-world applications: privacy and robustness to poisoning attacks.

Membership/Property Inference. Previous work [58] has shown that membership and property inference attacks are possible in FL when only a handful of participants are involved (e.g., less than 100). However, FP-Fed is not vulnerable to them as it is meant to be deployed in settings with many users. Moreover, it provides (Central) DP guarantees, which provably mitigate membership inference attacks, as recently shown by Naseri et al. [64], with $\varepsilon = 5.8$ and as little as four participants. Furthermore, by providing participant-level instead of record-level DP guarantees, we also protect against property inference.

Malicious Servers. Model updates are sent to the FP-Fed server unperturbed/unencrypted; thus, a malicious server can potentially infer the presence of specific records in users' training data [65]. However, as discussed in Section II-C, in this context, we believe that trusting the server with access to model updates – rather than raw data – is a reasonable compromise. Our work is the first to use FL in the context of browser fingerprinting; we focus on assessing the *feasibility* of this approach and measuring the impact of different settings (data distribution, feature sets, privacy levels, number of participants, etc.), while leaving it to future work to relax this assumption by extending FP-Fed to support Local or Distributed DP.

Poisoning Attacks. Data poisoning and backdoor attacks are also significant concerns in FL due to the distributed nature of the computation with untrusted participants [12]. In generic data poisoning attacks, adversarial participants attempt to compromise the utility of the global model by contributing malicious model updates. In targeted (aka *backdoor*) attacks, the adversary wants the FL system to deliberately misclassify specific samples or records to an adversary-defined class. In the context of browser fingerprinting, malicious participants could try to mount backdoor attacks so that FP-Fed will classify fingerprinting scripts as non-fingerprinting. However, even though previous work [63, 64] shows that CDP can defend against backdoor attacks, we leave it to future work to include a full experimental evaluation of data poisoning attacks against FP-Fed.

C. Practical Deployments

An important aspect of the real-world deployability of FP-Fed is whether a large range of devices with different computing power and resource constraints can support it. Browser fingerprinting detection is inherently relevant to a large variety of devices, from powerful computers/laptops to mobile devices and embedded devices. This setting is typically referred to as cross-device FL [16]. Therefore, having a lightweight FL system is very important. The choice of using a simple logistic regression model rather than a deep neural network means that models are small and training them does not require specialized hardware like GPUs.

Furthermore, as discussed in Section V-B, the complexity of the features that can be extracted from the scripts is also affected by the capabilities of the devices. In previous work, fingerprinting detectors are deployed through browser extensions like JShelter [40], instrumenting well-known APIs used for fingerprinting. However, a relevant percentage of web browsing happens on mobile devices where browser extensions cannot always be installed.[5] Therefore, using browser extensions as a deployment method is a significant limitation with respect to learning fingerprinting behaviors.

By contrast, our work focuses on natively traced APIs by Google Chrome. By doing so, FP-Fed can be deployed directly in the browser, regardless of the kind of device, and increase the coverage of participants, thus improving utility as a whole. Furthermore, by restricting to a small subset of features, we ensure that FP-Fed does not add significant performance costs to normal browser operation across a wide range of devices.

D. Implications for Browsers

The web ecosystem is crucial for an accessible and free Internet. As such, it has to continue to bring powerful capabilities for developers and users to complement dedicated apps available on mobile and desktop platforms. However, powerful web capabilities and APIs need to be evaluated for their potential to track end users' devices. Any API with a sufficiently high entropy can simultaneously be used for legitimate purposes and abused as a fingerprinting surface. Therefore, although this is relevant not only to FP-Fed but to the fingerprinting ecosystem as a whole, we advocate for a common way to evaluate new web APIs for "fingerprinting potential" before they are deployed. This type of evaluation could guide their specifications and implementation to balance the creation of new capabilities with legitimate privacy and tracking concerns for users.

For example, while the recently announced WebGPU API [60] can significantly assist high-performance computations and complex graphics rendering from within the browser, it might also enable potential fingerprinting scripts to more precisely identify the GPU hardware present in a machine compared to the old WebGL standard. In fact, this has already been acknowledged in the working draft of the WebGPU specification as increasing the risk of browser fingerprinting from the old WebGL standard [80].

E. Limitations & Future Work

Simulating distributed settings. Although FP-Fed is designed to learn from distributed real-world user behavior, due to cost and privacy challenges with collecting real-world browsing data (e.g., execution traces) from a large number of users, we chose to gather fingerprinting scripts using a centralized crawl and instead *simulate* a distributed setting. Having demonstrated a proof of concept for FP-Fed and its effectiveness, our next step is to focus on these challenges and evaluate its performance in an actual distributed setting.

Accuracy. FP-Fed achieves reasonably high accuracy even at strong privacy levels (with 1M participants, 0.86 AUPRC with $\varepsilon = 1$) and significantly improves over the local-only setting, i.e., all clients only train on their local datasets (0.78 AUPRC).

[5] As of May 2023, 52% of Web traffic is from mobile, as reported from https://gs.statcounter.com/platform-market-share/desktop-mobile-tablet/worldwide#monthly-202102-202202

W	ε=1	ε=5	ε=10	No DP
10^4	0.61	0.85	0.87	0.96
10^5	0.86	0.89	0.90	0.96
10^6	0.86	0.85	0.94	0.97

TABLE IV: Summary of Area Under the Precision-Recall Curve (AUPRC) values with different numbers of participants (W) and levels of privacy (using the *All* feature seat). For comparison, a fully centralized setting yields 0.97 AUPRC, while local only results in an average of 0.78 AUPRC.

However, there is still a non-negligible drop in performance compared to fully centralized settings (0.97 AUPRC) – please see Table IV, which provides a concise summary of FP-Fed's performance for different numbers of participants and privacy budgets. However, this drop is due to the use of differentially private algorithms rather than the federated nature of FP-Fed, as when we experiment with lower privacy settings, we quickly approach the fully centralized model's performance (e.g., 0.94 AUPRC with $\varepsilon = 10$).

This suggests there should be ample room to improve FP-Fed's performance. First, the effectiveness of inference attacks drops with large numbers of participants [58]; thus, FP-Fed does not necessarily need small values of ε. Second, there are possible optimizations from the point of view of DP through the use of newer differentially private optimization algorithms like DP-Follow-The-Regularized-Leader [43], which performs better than DP-SGD at moderate to high levels of privacy. Finally, more advanced neural network architectures and performing feature selection on top of feature pre-processing, etc., can also be experimented with to achieve higher utility at the same level of privacy. Since our main objective is to experiment with and assess the feasibility of a first-of-its-kind federated architecture for fingerprinting detection, we believe these improvements can be addressed in future work.

Moreover, we find that the overwhelming majority of misclassifications reducing AUPRC are false negatives. These could be considered less problematic than false positives in the context of fingerprinting and, more importantly, could be reduced with access to larger datasets covering more fingerprinting scripts. Also, as discussed, centralized crawls likely miss a number of fingerprinting scripts due to bot detection and user login walls, so their actual *recall* is likely much lower.

FL Bootstrap. While FP-Fed provides formal privacy guarantees, any real-world deployments of DP-FL face a number of practical challenges. For instance, devices often join and drop out of FL systems in unpredictable ways, which makes it difficult to sample users in a truly random manner. However, recent work [55] introduces new algorithms and privacy analysis techniques that mitigate this concern while providing comparable utility, at least in theory. Since we use second-order methods (LBFGS) instead of the first-order methods (SGD) as in [55], adapting their algorithms in a straightforward manner might potentially degrade FP-Fed's model performance, and thus we leave this to future work.

Heuristics. Finally, our ground truth relies on heuristics developed by FP-Inspector in 2019 [39]. We did so since this is a well-established, high-precision heuristic. However, the introduction of new APIs, such as WebGPU, and the imminent removal of third-party cookies from Chrome can potentially result in new types of fingerprinting being introduced. Detecting these new scripts would require new heuristics to be defined. Therefore, future work should continue to explore new avenues for fingerprinting and to define new high-precision heuristics.

F. Acknowledgments

This work has been supported by the National Science Scholarship (PhD) from the Agency for Science Technology and Research, Singapore, and a Google Research Faculty Award. The authors also wish to thank Umar Iqbal for sharing their automated web crawl data and source code for FP-Inspector [39].

References

[1] M. Abadi, A. Chu, I. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar, and L. Zhang. Deep learning with differential privacy. In *ACM CCS*, 2016.

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

[3] G. Acar, M. Juarez, N. Nikiforakis, C. Diaz, S. Gürses, F. Piessens, and B. Preneel. FPDetective: Dusting the Web for Fingerprinters. In *ACM CCS*, 2013.

[4] L. A. Adamic and B. A. Huberman. Zipf's law and the Internet. *Glottometrics*, 3(1), 2002.

[5] M. Adnan, S. Kalra, J. C. Cresswell, G. W. Taylor, and H. R. Tizhoosh. Federated learning and differential privacy for medical image analysis. *Scientific reports*, 12(1), 2022.

[6] N. Agarwal, P. Kairouz, and Z. Liu. The Skellam Mechanism for Differentially Private Federated Learning. *NeurIPS*, 2021.

[7] S. A. Akhavani, J. Jueckstock, J. Su, A. Kapravelos, E. Kirda, and L. Lu. Browserprint: An Analysis of the Impact of Browser Features on Fingerprintability and Web Privacy. In *Information Security Conference*, 2021.

[8] F. Alaca and P. C. Van Oorschot. Device fingerprinting for augmenting web authentication: classification and analysis of methods. In *ACM CCS*, 2016.

[9] Amazon. We will be retiring Alexa.com on May 1, 2022. https://web.archive.org/web/20220102200605/https://support.alexa.com/hc/en-us/articles/4410503838999, 2022.

[10] A. H. Amjad, Z. Shafiq, and M. A. Gulzar. Blocking JavaScript without Breaking the Web: An Empirical Investigation. *arXiv:2302.01182*, 2023.

[11] P. Arntz. Brave browser goes the extra mile to block third party cookies. https://www.malwarebytes.com/blog/news/2022/03/brave-browser-goes-the-extra-mile-to-block-third-party-cookies, 2022.

[12] E. Bagdasaryan, A. Veit, Y. Hua, D. Estrin, and V. Shmatikov. How to backdoor federated learning. In *International Conference on Artificial Intelligence and Statistics*, 2020.

[13] P. N. Bahrami, U. Iqbal, and Z. Shafiq. FP-Radar: Longitudinal Measurement and Early Detection of Browser Fingerprinting. *PETS*, 2022, 2022.

[14] R. Batuwita and V. Palade. Class Imbalance Learning Methods for Support Vector Machines. *Imbalanced Learning: Foundations, Algorithms, and Applications*, 2013.

[15] Berstend. puppeteer-extra-plugin-stealth. https://github.com/berstend/puppeteer-extra, 2023.

[16] K. Bonawitz, H. Eichner, W. Grieskamp, D. Huba, A. Ingerman, V. Ivanov, C. Kiddon, J. Konečný, S. Mazzocchi, B. McMahan, T. V. Overveldt, D. Petrou, D. Ramage, and R. Jason. Towards Federated Learning at Scale: System Design. *Proceedings of Machine Learning and Systems*, 1, 2019.

[17] Brave. Fingerprint Randomization. https://brave.com/privacy-updates/3-fingerprint-randomization/, 2020.

[18] E. Bursztein, A. Malyshev, T. Pietraszek, and K. Thomas. Picasso: Lightweight device class fingerprinting for web clients. In *Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2016.

[19] D. Cameron. Apple Declares War on 'Browser Fingerprinting,' the Sneaky Tactic That Tracks You in Incognito Mode. https://gizmodo.com/apple-declares-war-on-browser-fingerprinting-the-sneak-1826549108, 2018.

[20] Chrome. Chrome User Experience Report. https://developer.chrome.com/docs/crux/.

[21] A. Clauset, C. R. Shalizi, and M. E. Newman. Power-Law Distributions in Empirical Data. *SIAM Review*, 51(4), 2009.

[22] A. Das, G. Acar, N. Borisov, and A. Pradeep. The Web's Sixth Sense: A Study of Scripts Accessing Smartphone Sensors. In *ACM CCS*, 2018.

[23] S. Das. NHS data breach: trusts shared patient details with Facebook without consent. https://www.theguardian.com/society/2023/may/27/nhs-data-breach-trusts-shared-patient-details-with-facebook-meta-without-consent?CMP=Share_iOSApp_Other, 2022.

[24] S. Das. NHS data breach: trusts shared patient details with Facebook without consent. https://www.theguardian.com/society/2023/may/27/nhs-data-breach-trusts-shared-patient-details-with-facebook-meta-without-consent, 2023.

[25] Disconnect. Disconnect defends the digital you. https://disconnect.me, 2018.

[26] C. Dwork and A. Roth. The Algorithmic Foundations of Differential Privacy. *Foundations and Trends in Theoretical Computer Science*, 2014.

[27] EasyList. https://easylist-downloads.adblockplus.org/.

[28] EasyList. EasyPrivacy. https://easylist.to/easylist/easyprivacy.txt.

[29] P. Eckersley. How Unique Is Your Web Browser? *PETS*, 2010.

[30] EFF. PrivacyBadger. https://github.com/EFForg/privacybadgerfirefox/blob/master/data/cookieblocklist.txt, 2023.

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

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

[33] Fingerprint. Bot Detection Guide. https://dev.fingerprint.com/docs/bot-detection-quick-start-guide.

[34] R. Gálvez, V. Moonsamy, and C. Diaz. Less is More: A privacy-respecting Android malware classifier using federated learning. *arXiv:2007.08319*, 2020.

[35] R. C. Geyer, T. Klein, and M. Nabi. Differentially private federated learning: A client level perspective. *arXiv:1712.07557*, 2017.

[36] F. Hartmann and P. Kairouz. Distributed differential privacy for federated learning. https://ai.googleblog.com/2023/03/distributed-differential-privacy-for.html, 2023.

[37] M. Ikram, H. J. Asghar, M. A. Kaafar, B. Krishnamurthy, and A. Mahanti. Towards Seamless Tracking-Free Web: Improved Detection of Trackers via One-class Learning. *PETS*, 2017.

[38] Iovation. Iovation Fraud Protection. https://web.archive.org/web/20191130164107/\https://www.iovation.com/fraudforce-fraud-detection-prevention, 2019.

[39] U. Iqbal, S. Englehardt, and Z. Shafiq. Fingerprinting the fingerprinters: Learning to detect browser fingerprinting behaviors. In *IEEE S&P*, 2021.

[40] JShelter. https://jshelter.org/.

[41] J. Jueckstock, S. Sarker, P. Snyder, A. Beggs, P. Papadopoulos, M. Varvello, B. Livshits, and A. Kapravelos. Towards Realistic and Reproducible Web Crawl Measurements. In *WWW*, 2021.

[42] P. Kairouz, Z. Liu, and T. Steinke. The Distributed Discrete Gaussian Mechanism for Federated Learning with Secure Aggregation. In *ICML*, 2021.

[43] P. Kairouz, B. Mcmahan, S. Song, O. Thakkar, A. Thakurta, and Z. Xu. Practical and Private (Deep) Learning Without Sampling or Shuffling. In *ICML*, 2021.

[44] P. Kairouz, H. B. McMahan, B. Avent, A. Bellet, M. Bennis, A. N. Bhagoji, K. Bonawitz, Z. Charles, G. Cormode, R. Cummings, et al. Advances and Open Problems in Federated Learning. *Foundations and Trends in Machine Learning*, 14(1–2), 2021.

[45] P. Kairouz, S. Oh, and P. Viswanath. The composition theorem for differential privacy. In *ICML*, 2015.

[46] J. Kang, Z. Xiong, D. Niyato, S. Xie, and J. Zhang. Incentive Mechanism for Reliable Federated Learning: A Joint Optimization Approach to Combining Reputation and Contract Theory. *IEEE Internet of Things Journal*, 6(6), 2019.

[47] B. Krishnamurthy and C. Wills. Privacy diffusion on the web: a longitudinal perspective. In *WWW*, 2009.

[48] P. Laperdrix. Browser Fingerprinting: An Introduction and the Challenges Ahead. https://blog.torproject.org/browser-fingerprinting-introduction-and-challenges-ahead/, 2019.

[49] P. Laperdrix, G. Avoine, B. Baudry, and N. Nikiforakis. Morellian analysis for browsers: Making web authentication stronger with canvas fingerprinting. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, 2019.

[50] R. Lawler. Google's turning off third-party cookies for 1 percent of Chrome users early next year. https://www.theverge.com/2023/5/18/23728263/google-chrome-ad-tracking-third-party-cookies-privacy-sandbox, 2023.

[51] B. Li, Y. Wu, J. Song, R. Lu, T. Li, and L. Zhao. DeepFed: Federated deep learning for intrusion detection in industrial cyber–physical systems. *IEEE Transactions on Industrial Informatics*, 17(8), 2020.

[52] Y. Liu, S. Garg, J. Nie, Y. Zhang, Z. Xiong, J. Kang, and M. S. Hossain. Deep anomaly detection for time-series data in industrial IoT: A communication-efficient on-device federated learning approach. *IEEE Internet of Things Journal*, 8(8), 2020.

[53] I. Lunden. Relx acquires ThreatMetrix for $817M to ramp up in risk-based authentication. https://techcrunch.com/2018/01/29/relx-threatmetrix-risk-authentication-lexisnexis/?guccounter=1, 2018.

[54] S. Maddock, G. Cormode, T. Wang, C. Maple, and S. Jha. Federated Boosted Decision Trees with Differential Privacy. In *ACM CCS*, 2022.

[55] B. McMahan and A. Thakurta. Federated learning with formal differential privacy guarantees. *Google AI Blog*, 2022.

[56] H. B. McMahan, E. Moore, D. Ramage, and B. A. y Arcas. Federated learning of deep networks using model averaging. *arXiv:1602.05629*, 2016.

[57] H. B. McMahan, D. Ramage, K. Talwar, and L. Zhang. Learning Differentially Private Recurrent Language Models. In *ICLR*, 2018.

[58] L. Melis, C. Song, E. De Cristofaro, and V. Shmatikov. Exploiting Unintended Feature Leakage in Collaborative Learning. In *IEEE S&P*, 2019.

[59] V. Mothukuri, P. Khare, R. M. Parizi, S. Pouriyeh, A. Dehghantanha, and G. Srivastava. Federated-Learning-Based Anomaly Detection for IoT Security Attacks. *IEEE Internet of Things Journal*, 9(4), 2021.

[60] Mozilla. WebGPU API. https://developer.mozilla.org/en-US/docs/Web/API/WebGPU_API, 2023.

[61] Mozilla. WebRTC API. https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API, 2023.

[62] MultiLogin. Canvas Defender. https://multilogin.com/canvas-defender/.

[63] M. Naseri, Y. Han, E. Mariconti, Y. Shen, G. Stringhini, and E. De Cristofaro. Cerberus: Exploring Federated Prediction of Security Events. In *ACM CCS*, 2022.

[64] M. Naseri, J. Hayes, and E. De Cristofaro. Local and Central Differential Privacy for Robustness and Privacy in Federated Learning. In *NDSS*, 2022.
[65] M. Nasr, R. Shokri, and A. Houmansadr. Comprehensive Privacy Analysis of Deep Learning: Passive and Active White-box Inference Attacks against Centralized and Federated Learning. In *IEEE S&P*, 2019.
[66] R. Ngan, S. Konkimalla, and Z. Shafiq. Nowhere to Hide: Detecting Obfuscated Fingerprinting Scripts. *arXiv:2206.13599*, 2022.
[67] V. Pihur, A. Korolova, F. Liu, S. Sankuratripati, M. Yung, D. Huang, and R. Zeng. Differentially-private "draw and discard" machine learning. In *CSCML*, 2022.
[68] V. Pihur, A. Korolova, F. Liu, S. Sankuratripati, M. Yung, D. Huang, and R. Zeng. Differentially-Private "Draw and Discard" Machine Learning: Training Distributed Model from Enormous Crowds. In *Cyber Security, Cryptology, and Machine Learning Symposium*, 2022.
[69] V. L. Pochat, T. van Goethem, S. Tajalizadehkhoob, M. Korczynski, and W. Joosen. Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation. In *NDSS*, 2019.
[70] G. Pugliese, C. Riess, F. Gassmann, and Z. Benenson. Long-Term Observation on Browser Fingerprinting: Users' Trackability and Perspective. *PETS*, 2020.
[71] J. Rack and C.-A. Staicu. Jack-in-the-box: An Empirical Study of JavaScript Bundling on the Web and its Security Implications. *CCS*, 2023.
[72] V. Rey, P. M. S. Sánchez, A. H. Celdrán, and G. Bovet. Federated learning for malware detection in iot devices. *Computer Networks*, 204, 2022.
[73] F. Roesner, T. Kohno, and D. Wetherall. Detecting and Defending Against Third-Party Tracking on the Web. In *USENIX*, 2012.
[74] K. Ruth, D. Kumar, B. Wang, L. Valenta, and Z. Durumeric. Toppling top lists: Evaluating the accuracy of popular website lists. In *ACM IMC*, 2022.
[75] J. Schuh. Building a more private web. https://www.blog.google/products/chrome/building-a-more-private-web/, 2019.
[76] L. Sun, J. Qian, and X. Chen. LDP-FL: Practical Private Aggregation in Federated Learning with Local Differential Privacy. In *IJCAI*, 2021.
[77] S. Truex, L. Liu, K.-H. Chow, M. E. Gursoy, and W. Wei. LDP-Fed: Federated learning with local differential privacy. In *ACM International Workshop on Edge Systems, Analytics and Networking*, 2020.
[78] W3C. Mitigating Browser Fingerprinting in Web Specifications. https://w3c.github.io/fingerprinting-guidance/, 2021.
[79] W3C. Improving the web without third-party cookies. https://www.w3.org/2001/tag/doc/web-without-3p-cookies/, 2023.
[80] W3C. WebGPU. https://www.w3.org/TR/webgpu/, 2023.
[81] A. White. Google's turning off third-party cookies for 1 percent of Chrome users early next year. https://developer.chrome.com/docs/privacy-sandbox/privacy-budget/, 2022.
[82] J. Wilander. Bounce Tracking Protection. https://github.com/privacycg/proposals/issues/6, 2020.
[83] J. Wilander. Full Third-Party Cookie Blocking and More. https://webkit.org/blog/10218/full-third-party-cookie-blocking-and-more/, 2020.
[84] M. Wood. Todays Firefox Blocks Third-Party Tracking Cookies and Cryptomining by Default. https://blog.mozilla.org/en/products/firefox/todays-firefox-blocks-third-party-tracking-cookies-and-cryptomining-by-default/, 2019.
[85] ZenRows. How to Bypass Cloudflare in 2023: The 8 Best Methods. https://www.zenrows.com/blog/bypass-cloudflare, 2023.

APPENDIX A
MISSED SCRIPTS

As mentioned, since centralized crawlers cannot replicate real human interactions, they might miss scripts on webpages protected by bot detectors, CAPTCHAs, and webpages that require user login. One specific example is available at https://docs.google.com/document/d/1luaU5I8oU8wVt-QFy5muGjM47GkW57I8rO8n8dcyX-8/.[6] This script, found in the wild during our top 300 crawl (see Section IV-D), enumerates various device properties, i.e., screen size, language, device timezone, plugins installed, and WebGL configuration, on top of performing canvas and audio fingerprinting.

Even though we use a bot detection evasion tool[7], our automated crawler was detected as a bot and prevented from visiting the website, thus missing the script. On the other hand, our manual crawl with real user interaction was able to visit the website and flag the script as fingerprinting.

APPENDIX B
NON-IIDNESS SCORE

To quantify the Non-IIDness of the distributions tested, we follow an approach inspired by previous work [63], which introduces and uses the so-called *Non-IIDness score.* This calculates the average pairwise distance, expressed in terms of Kullback–Leibler (KL) divergence, between the distribution of classes among participants. For instance, when the training data is IID distributed, the distribution of classes among participants is roughly similar, which leads to a small average distance and a small Non-IIDness score. However, when the training data is unevenly distributed, the distribution of classes is starkly different, resulting in a high average pairwise distance between distributions and thus high Non-IIDness score.

To use it in our setting, we need to introduce some modifications. First, while [63] calculates the average distance across all participants, the number of participants in our setting is much greater than in theirs (we have millions of users, whereas they only had a few thousand). Thus, we cannot enumerate (possibly) trillions of pairwise combinations. Rather, we calculate the average over a sample of 1000 participants (corresponding to roughly 1M pairwise combinations).

Next, since browser fingerprinting is a heavily class-imbalanced problem, the distribution of fingerprinting scripts in participants does not change the overall distribution of scripts significantly. Therefore, in this work, we calculate the distance between the distribution of only fingerprinting scripts by type (excluding non-fingerprinting scripts).

Finally, since scripts can contain multiple types of fingerprinting behavior, the distribution of fingerprinting scripts consists of not just each individual type of fingerprinting but all combinations of types of fingerprinting as well (e.g., Canvas, Canvas Font, Audio, WebRTC, Canvas and Canvas Font, Canvas and Audio, etc.).

[6] In the spirit of responsible disclosure, we omit the domain name where the script has been found along with any identification detail.
[7] https://www.npmjs.com/package/puppeteer-extra-plugin-stealth