sciendo

Proceedings on Privacy Enhancing Technologies ..; .. (..):1–21

Pouneh Nikkhah Bahrami*, Umar Iqbal, and Zubair Shafiq

FP-Radar: Longitudinal Measurement and Early Detection of Browser Fingerprinting

Abstract: Browser fingerprinting is a stateless tracking technique that attempts to combine information exposed by multiple different web APIs to create a unique identifier for tracking users across the web. Over the last decade, trackers have abused several existing and newly proposed web APIs to further enhance the browser fingerprint. Existing approaches are limited to detecting a specific fingerprinting technique(s) at a particular point in time. Thus, they are unable to systematically detect novel fingerprinting techniques that abuse different web APIs. In this paper we propose FP-Radar, a machine learning approach that leverages longitudinal measurements of web API usage on top-100K websites over the last decade, for early detection of new and evolving browser fingerprinting techniques. The results show that FP-Radar is able to early detect the abuse of newly introduced properties of already known (e.g., `WebGL`, `Sensor`) and as well as previously unknown (e.g., `Gamepad`, `Clipboard`) APIs for browser fingerprinting. To the best of our knowledge, FP-Radar is also first to detect the abuse of the `Visibility` API for ephemeral fingerprinting in the wild.

DOI Editor to enter DOI
Received ..; revised ..; accepted ...

1 Introduction

The online tracking ecosystem employs increasingly sophisticated tracking techniques to track users across the web [25, 56, 70, 112]. In addition to well-known *stateful tracking* using third-party cookies, trackers have now started to use more intrusive *stateless tracking* techniques such as browser fingerprinting to gather device-specific identifying information captured through various HTTP header fields and APIs [16, 25, 27, 54, 71, 72, 78]. Stateless tracking is more intrusive than stateful tracking because the former does not lend itself to transparency and control. While cookies are directly observed and removed at the client-side, browser fingerprint is not directly visible at the client-side and it cannot be trivially removed or even modified. As web browsers have started to implement aggressive countermeasures against stateful tracking [90, 108, 110], it has encouraged trackers to migrate to more opaque and invasive stateless tracking [41, 90].

Browser fingerprinting techniques have evolved over time. As web browsers support new functionality by adding new APIs or update existing APIs [95], the browser's fingerprinting surface has continued to expand. Early work by Mayer [57] and Eckersley [24] demonstrated simple fingerprinting techniques that abuse information exposed in HTTP headers and a few APIs. A steady stream of more sophisticated fingerprinting techniques have since been developed that abuse existing and new APIs. For example, researchers have shown that `Canvas` [72], `WebGL` [16], fonts [27], extensions [96], the `Audio` API [25], the `Battery Status` API [78, 79], the `Performance` API [88], and even sensor APIs [7, 17] can expose information that can be abused to build a more reliable fingerprint. Thus, as new APIs are introduced in web browsers, it is reasonable to expect that they might be abused to implement novel browser fingerprinting techniques. In summary, browser fingerprinting is not a static phenomenon, but it is rather evolving; as novel fingerprinting techniques are designed over time.

Browser fingerprinting and its privacy implications have received much attention from the research community. Researchers have conducted large-scale measurements to study the prevalence of browser fingerprinting [2, 3, 17, 25, 26, 28, 41, 76, 79]. However, prior research on browser fingerprinting is lacking in two major ways. First, prior work is mostly limited to analyzing a specific fingerprinting technique(s) at a particular point in time. Since fingerprinting techniques evolve over time, it is important to study browser fingerprinting longitudinally. Second, prior work is limited to detecting deployment of already known fingerprinting techniques. It is important to detect new fingerprinting techniques in a timely fashion because early detection can aid proactive mitigation efforts by the standards bodies [23] and also prompt deployment of targeted countermeasures by browser vendors [55, 81].

***Corresponding Author: Pouneh Nikkhah Bahrami**, University of California, Davis, Email: pnikkhah@ucdavis.edu
Umar Iqbal, University of Washington, Email: umar@cs.washington.edu
Zubair Shafiq, University of California, Davis, Email: zubair@ucdavis.edu

We propose FP-Radar, a machine learning approach for early detection of web API abuse for fingerprinting. FP-Radar detects abuse of new methods of existing APIs or new APIs altogether by using the guilt-by-association principle. More specifically, it first uses the Wayback Machine to crawl the historical snapshots of scripts on top-100K websites over the last decade. FP-Radar conducts static analysis to construct a series of temporal API co-occurrence graphs for each year. FP-Radar then uses hand-crafted and embedding features to predict the evolution of co-occurrence relationships between different API keywords over the years. FP-Radar then builds and labels temporal clusters, including the fingerprinting cluster, using the temporal graphs. Finally, FP-Radar tracks the membership of the fingerprinting cluster over time for early detection of API abuse.

The results show that FP-Radar is able to detect the abuse of already known as well as unknown APIs for fingerprinting. First, FP-Radar detects the abuse of a number of previously unknown APIs including `Page Visibility`, `Gamepad`, `Clipboard`, and `Network Information` for browser fingerprinting. We find novel types of user environment/hardware fingerprinting such as peripheral configuration via `Gamepad` and system capabilities via `Network Information` APIs. We also find that even though an API (e.g., `Page Visibility`) does not directly expose highly identifying information it can be abused for ephemeral fingerprinting. To the best of our knowledge, FP-Radar is also the first to detect the abuse of web APIs for ephemeral fingerprinting in the wild. Second, FP-Radar detects the abuse of newly introduced features of APIs that are already known to be abused for fingerprinting. We find that several of the newly introduced features of `Navigator` (e.g., related to hardware capabilities such as memory), `Performance` (e.g., time for DNS lookup and page rendering), and `WebGL` (e.g., WebGL2 capabilities) are now being abused for fingerprinting. Finally, FP-Radar is able to detect the fingerprinting abuse of APIs before/at their disclosure or at the time of their release by browser vendors or their first occurrence in our data. We find that FP-Radar's time-to-detection is often several years before public disclosure (e.g., as much as 6 years for `Gamepad` and 7 years for `Page Visibility` APIs).

We summarize our key contributions as follows:

1. A retrospective **longitudinal measurement** study of web API usage over the last decade.
2. A graph-based **supervised ML approach** that builds a series of API co-occurrence graphs to predict the evolution of API usage in the future.
3. A graph-based **unsupervised ML approach** that clusters temporal API co-occurrence graphs for early detection of their abuse for fingerprinting.

2 Background & Related Work

2.1 Background

Web browsers support standardized web APIs to facilitate feature-rich websites that can be seamlessly loaded on different browsers (e.g., Chrome, Firefox), operating systems (e.g., Mac/Windows), and devices (e.g., mobile/desktop). Unfortunately, the rich set of information exposed by the web APIs can also be exploited by trackers to fingerprint users' devices. Trackers can simply combine several pieces of readily available information, such as the operating system name, browser name, browser version in the user-agent field, to build a fingerprint that can distinguish between different devices. Trackers can also use more sophisticated fingerprinting techniques that exploit subtle differences in the underlying hardware/software configurations and capabilities to gather distinctive information. For example, canvas images are rendered differently on different browsers due to the differences in their hardware/software image processing pipeline. Combining several of these fingerprinting techniques, trackers can create a fingerprint that is often sufficient to uniquely and persistently identify the web browser [24].

Browser fingerprinting is called stateless tracking since there is no need to store state at the client-side, as done in traditional cookie-based stateful tracking. Stateless tracking is considered more intrusive than stateful tracking because the former does not lend itself to transparency and control. While cookies and other types of client-side storage mechanisms (e.g., localStorage, IndexedDB) can be readily observed and blocked at the client-side, browser fingerprint is not directly visible at the client-side and it cannot be trivially removed or even modified. As web browsers have started to implement aggressive countermeasures against stateful tracking [90, 108, 110], it has encouraged trackers to migrate to more opaque and invasive stateless tracking [41, 90]. Browser fingerprinting is already being used for cross-site tracking [4, 41, 52] and is universally regarded as an abusive practice by standards bodies [23, 77] and web browsers [15, 55, 73].

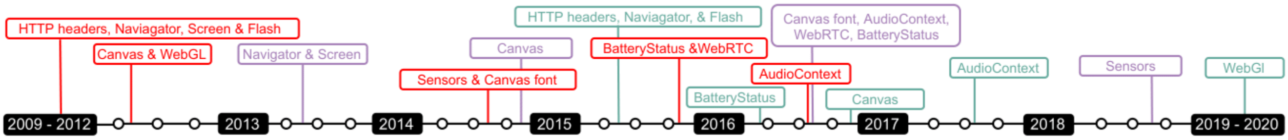

Fig. 1. The timeline summarizes the chronological disclosure and adoption of fingerprinting APIs and countermeasures. Disclosures are represented with red, adoptions are represented with purple, and countermeasures are represented with green.

2.2 Chronology of Browser Fingerprinting

The fingerprinting surface has continued to expand with the introduction of new APIs and the disclosure of fingerprinting potential in existing APIs. Soon after an API is disclosed to have fingerprinting potential, they are adopted by trackers. Countermeasures, also follow suit, and attempt to mitigate the fingerprintability of the API. As shown in Figure 1, this pattern has been repeated over the years. We next provide a chronology of the disclosure, adoption, countermeasure of web APIs for fingerprinting.

Disclosure. Mayer [57] first investigated browser fingerprinting in 2009 and showed that the fingerprints created through `navigator` and `screen` can uniquely identify 96.23% of the browsers. Soon after that in 2010, Eckersly [24] conducted a large scale user study to demonstrate that the information exposed through HTTP headers, e.g., `User-Agent` and APIs, e.g., `navigator`, and Flash, e.g. `fonts`, can be used to uniquely identify 94.2% of the browsers. In 2012, Mowery and Shacham [72] first introduced "execution-based" canvas and WebGL fingerprinting and showed that the certain images rendered through `canvas` and `WebGL` APIs on different devices produce different outputs due to the variance in hardware (e.g., graphics card) and software (e.g., browser version, configurations). Since then researchers have demonstrated the fingerprinting potential of mobile sensors and canvas font in 2014 [7, 100], `Battery Status` and `WebRTC` [10, 78] in 2015, and `AudioContext` in 2016.

Adoption. Roughly after 2 years of disclosure, i.e., in 2013, browser fingerprinting, based on HTTP header information, JavaScript APIs, and Flash, was discovered on 40 of the top-10K websites [76]. Within the next year, fingerprinting adoption exploded and canvas fingerprinting was discovered on 5,542 of top-100K websites, which is only 2 years after its initial disclosure [2]. The wide adoption of canvas fingerprinting was attributed to the release of fingerprintjs2 [1], an open-source fingerprinting library. Later, in 2016, Englehardt et al., conducted a large scale study of top-1 million websites and further found the deployment of canvas font, `WebRTC`, `Audiocontext`, and `Battery Status` API fingerprinting on 14,371, 3,250, 715, 518, and 22 websites, respectively [25, 79], i.e., only after 1-2 years of their disclosure. In 2018, Das et al. [17] found the usage of sensors, such as motion and orientation, for browser fingerprinting on 3,695 of the top-100K websites, which is 4 years after their initial disclosure.

Countermeasures. Countermeasures against browser fingerprinting have a difficult time keeping up with the adoption of APIs for browser fingerprinting. It nearly took 2 years, after the adoption of HTTP header (e.g., `User-Agent`) and APIs (e.g., `Navigator`) exploitation for fingerprinting to propose robust countermeasures against them [75, 101]. Similarly, the countermeasures against `Battery Status`, `canvas`, `AudioContext`, and `WebGL`, fingerprinting were first proposed in 2016 [68], 2016 [5], 2017 [53], and 2019 [111], respectively, which is nearly 1–7 years after their adoption. Some recent heuristics and machine learning approaches [18, 25, 41, 85, 86] have attempted to detect known fingerprinting techniques and block the scripts that implement them. Englehardt and Narayanan [25] proposed heuristics to detect fingerprinting scripts that implement `canvas`, `canvas Font`, and `webRTC` fingerprinting techniques. They incidentally discovered the use of `AudioContext` fingerprinting in their manual analysis of the detected fingerprinting scripts. Iqbal et al. [41] proposed a supervised machine learning approach to detect fingerprinting scripts that implement various fingerprinting techniques, such as `canvas`, `canvas Font`, `webRTC`, `WebGL`, and `AudioContext`. They also incidentally discovered the potential use of peripheral probing (e.g., `getLayoutMap`) and `Permissions` API based fingerprinting in the post-hoc analysis of the detected fingerprinting scripts. DuckDuckGo proposed to detect browser fingerprinting scripts based on the sum of "API weights" – which are the ratio of API's appearance in "suspicious scripts" to "non-suspicious scripts" [18]. Based on the API weights, DuckDuckGo incidentally discovered the potential use of `deviceMemory` and `Presentation` APIs for browser fingerprinting [19].

2.3 Takeaway

In conclusion, prior work is limited to reactive detection of scripts that implement known fingerprinting techniques. Unsurprisingly, as discussed above, existing approaches have a difficult time keeping up because they are not designed to detect new fingerprinting techniques [18, 25, 41]. Thus, as we discuss next, it is important to design approaches to detect new fingerprinting techniques in a timely fashion.

3 FP-Radar

We present the design and implementation of FP-Radar, a temporal graph based machine learning approach for early detection of web API abuse for browser fingerprinting. As shown in Figure 2, FP-Radar can be divided into four components. First, it models the temporal co-occurrence of web APIs in scripts using a graph representation. Second, it leverages the temporal graph representation to predict future co-occurrence of web APIs. Third, it leverages the predicted co-occurrence to cluster web APIs based on their functionality. Finally, the temporal clusters are analyzed to detect abuse of specific APIs (and their respective keywords) for browser fingerprinting.

3.1 Modeling Temporal API Co-occurrence

FP-Radar relies on the principle of *guilt by association* to detect the abuse of web APIs for browser fingerprinting. It means that if an API is being used alongside known fingerprinting APIs then we can presume that the API in question is also being abused for browser fingerprinting. We rely on the insight that trackers often use several fingerprinting techniques, and thus several fingerprinting APIs, together, to conduct browser fingerprinting [17, 25, 41, 54]. FP-Radar operationalizes this insight in a longitudinal fashion to capture temporal trends in web API usage and early detection of web API abuse for browser fingerprinting.

3.1.1 Longitudinal Data Crawling

To longitudinally analyze web APIs, FP-Radar needs to measure their usage on the web over time. We conduct a retrospective measurement study to analyze how web API usage has evolved on popular websites. To gather historical snapshots of popular websites, we rely on the Internet Archive's Wayback Machine [106]. The Wayback Machine has periodically archived popular websites and their resources (e.g., scripts, images) since 1996 and has already archived more than 600 billion web pages thus far. The Wayback Machine has been used in prior literature to conduct longitudinal measurements of online tracking [42, 56].

Crawling scripts using the Wayback Machine. FP-Radar relies on the Wayback Machine [106] to crawl historical snapshots of a large set of scripts present on Alexa top-100K websites over the last decade (2010–2019). Since crawling the Wayback Machine incurs significant additional overheads as compared to live web crawls, we limit our Wayback Machine crawls to scripts observed in our initial live crawl of Alexa top-10K websites and 10K websites randomly sampled from Alexa 10K-100K websites. To improve coverage of fingerprinting scripts, we further use the Wayback Machine to crawl historical snapshots of known fingerprinting scripts reported in recent prior work on Alexa top-100K websites [41]. It is noteworthy that FP-Radar is able to establish a comprehensive longitudinal view of web APIs usage because it conducts large-scale crawls of Alexa top-100K websites using Wayback machine instead of narrowly analyzing historical snapshots of a few number of JavaScript libraries for fingerprinting such as `fingerprintingjs2` [1].

Completeness issues in the Wayback Machine. The Wayback Machine has completeness issues due to the inherent challenges of archiving the web [9, 34, 42, 47, 56]. First, the Wayback Machine used to not crawl websites based on their `robots.txt` policy.[1] Second, the Wayback Machine's crawls might miss dynamic resources. The Wayback Machine does not fully execute JavaScript during its archival process and thus misses some client-side dynamically generated URLs [56]. Moreover, a resource might also not be captured by the Wayback Machine if the resource URL (file name or path) changes; thus the same resource is present with a different URL in the Wayback Machine's archival crawl as compared to our initial live crawl. Third, the Wayback Machine crawls less popular websites less frequently and thus might not crawl resources on low-ranked websites at least once every year.

1 Note that the Wayback Machine has resumed crawling websites since 2017 irrespective of their `robots.txt` policy [30].

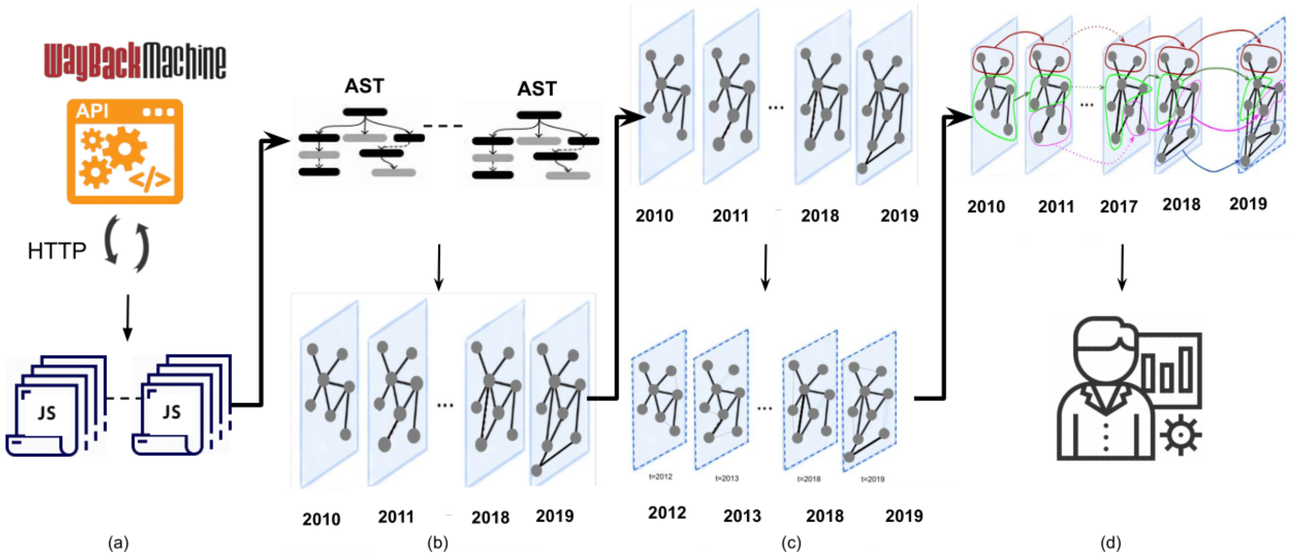

Fig. 2. FP-Radar: (a) We use the Wayback Machine to crawl historical snapshots of scripts present on Alexa top-100K websites from 2010 to 2019, (b) We first create AST representation of crawled scripts to extract web API keywords, we then model the temporal co-occurrence of web APIs in scripts in a graph representation, (c) We leverages the temporal graph representation to predict future co-occurrence of web API keywords, (d) We first leverage the predicted co-occurrence to cluster web APIs based on their functionality, we then analyze the temporal clusters to detect abuse of specific API keywords for browser fingerprinting.

Wayback Machine crawl statistics. Despite the aforementioned completeness issues in the Wayback Machine's crawls, we are able to longitudinally crawl yearly snapshots of almost 100K scripts from the Wayback Machine over the last decade (2010–2019). Based on classification of [103], this includes 1,658 fingerprinting and 92,193 non-fingerprinting scripts from our initial live crawl. Note that we use a two step process to crawl the Wayback Machine: we first fetch the URLs that point to the historical snapshots of scripts [105] and then send requests for those URLs to gather their script content. The first step returns URLs with the timestamp and the hash digest of the script content. The timestamps enables us to crawl scripts that are one year apart from each other and the hash digest helps us avoid crawling duplicate scripts in the second step.

We acknowledge that FP-Radar's longitudinal data collection misses a substantial number of scripts due to the completeness issues in the Wayback Machine. Specifically, with reference to our initial live crawl, we note that FP-Radar is unable to crawl snapshots of 43.09% of the scripts from the Wayback Machine. While not ideal, we do not observe any bias in the missing scripts. Specifically, both fingerprinting and non-fingerprinting scripts are missed with roughly the same proportion, i.e., 43.60% and 46.74%, respectively. Moreover, despite the missing data, FP-Radar's longitudinal data collection is able to capture the overall trend of increasing adoption of browser fingerprinting over the years. Specifically, we observe fingerprinting scripts on 1.16% and 3.70% of the top-100K websites in 2016 and 2018, respectively. This corroborates with the findings of prior studies of browser fingerprinting, which reported that 1.43% of the top-million [25] and 3.69% of the top-100K [17] websites conduct browser fingerprinting in 2016 and 2018, respectively. Thus, we conclude that FP-Radar's longitudinal data collection using the Wayback Machine is sufficient for us to retrospectively study the evolution of browser fingerprinting and draw meaningful conclusions. We discuss alternates to the Wayback Machine and ideas to improve completeness of longitudinal crawls in Section 5.

3.1.2 Graph Representation

To model guilt by association, we represent web API co-occurrence in a graph. Specifically, we model API keywords as nodes and include an edge between the nodes if the API keywords co-occur in the same script. We further weigh the edges based on the normalized co-occurrence frequency.

API keyword extraction. To extract API keywords from scripts, we model script text in abstract syntax trees (ASTs) that normalize scripts for developer coding styles.[2] ASTs also remove the non-essential script content (e.g., comments), generalize the APIs into generic primitives (e.g., `VariableDeclaration` and `ForStatement`), and capture the syntactical relationship between the APIs in form of a tree (e.g., an API call in a loop). Most importantly, ASTs provide a traverse-able tree representation of scripts, which allows us to extract the API keywords. We then traverse the ASTs from their roots to extract API keywords that match the standardized web APIs [69].

Temporal graph representation. We capture the longitudinal co-occurrence of API keywords by annotating the edges with the timestamp, i.e., year, of API keyword co-occurrence. Furthermore, we capture the frequency of co-occurrence between the APIs over years by summing the edge weights. Figure 3 demonstrates the creation of temporal graph. The figure shows a sample non-temporal graph representation for year 2010, 2011, and 2012 and their aggregated temporal graph representation. It can be seen in the aggregated graph representation that the edges are annotated with all the years in which the APIs co-occur and that the weight over the years is combined together. For example, the weight of edge between `random` and `fullPath` in the aggregated graph (0.0041) is the sum of the weight of the edges between 2010 and 2012.

3.2 Predicting API Co-occurrence

To assist with FP-Radar's goal of early detection of web API abuse for browser fingerprinting, we attempt to predict API co-occurrence in the future. To this end, we leverage the longitudinal connectivity of APIs with each other to predict their future connectivity. We capture the longitudinal connectivity of APIs using *hand-crafted* and *graph-embedding* features. Our rationale for relying on these features is that the existing connectivity of APIs is indicative of their future connectivity.

Hand-crafted features. We first capture API co-occurrence patterns, targeting neighborhood connectivity, through hand-crafted features. These features model the connectivity between APIs, centrality of APIs, and the commonalities in API neighborhood. We also incorporate node weight and temporal information by giving more value to the recently formed edges. Specifically, the weight between two nodes is multiplied by a time factor, which decreases by one per year, for prior years. Incorporating weighted temporal information allows us to give more importance to the recent API co-occurrence patterns in the graph, which might be a better representative of the future connectivity between APIs.

We list hand-crafted features below:

1. *Common Neighbors:* The number of common neighbors between a node pair. The value is higher if the nodes have high number of common neighbors.
2. *Adamic-Adar Index:* The sum of the inverse logarithmic degree of the neighbors shared by a node pair. The nodes with fewer common neighbors have higher values.
3. *Hub Promoted Index:* The number of common neighbors divided by the number of neighbors of the node with least degree in a node pair. The node pairs adjacent to hubs (high-degree nodes) have high values.
4. *Hub Depressed Index:* The number of common neighbors divided by the number of neighbors of the node with highest degree in a node pair. The node pairs adjacent to hubs (high-degree nodes) have low values.
5. *Jaccard Index:* The proportion of common neighbors by the total number of neighbors of a node pair. The value is higher if a node pair has more common neighbors in their neighborhood.
6. *Leicht-Holme-Newman Index:* The number of common neighbors divided by the product of the degree of the node pair. The value is higher if the nodes have low degree.
7. *Resource Allocation Index:* The summation of the inverse of the degree of common neighbors between a node pair. The value is higher if the neighbors have low degree.
8. *Salton Index (Cosine similarity):* It measures the cosine of the angle between the neighbors of a node pair. The more common the neighboring nodes, the higher the value.
9. *Sorensen Similarity:* The proportion of the common neighbors by the sum of the degree of a node pair. The value is higher if the node pair has low degree.

Prior research [13] has shown that these features are highly predictive of the future connectivity in temporal graphs. However, these features were only evaluated on temporal social network graphs and they may not be effective on temporal web API co-occurrence graphs. To this end, we compute the information gain [46] (feature importance) of these features to evaluate their potential

2 We unpack `eval`'ed scripts with an instrumented browser [43]. Unpacking allows us to treat scripts as code, which otherwise will be treated as a text string.

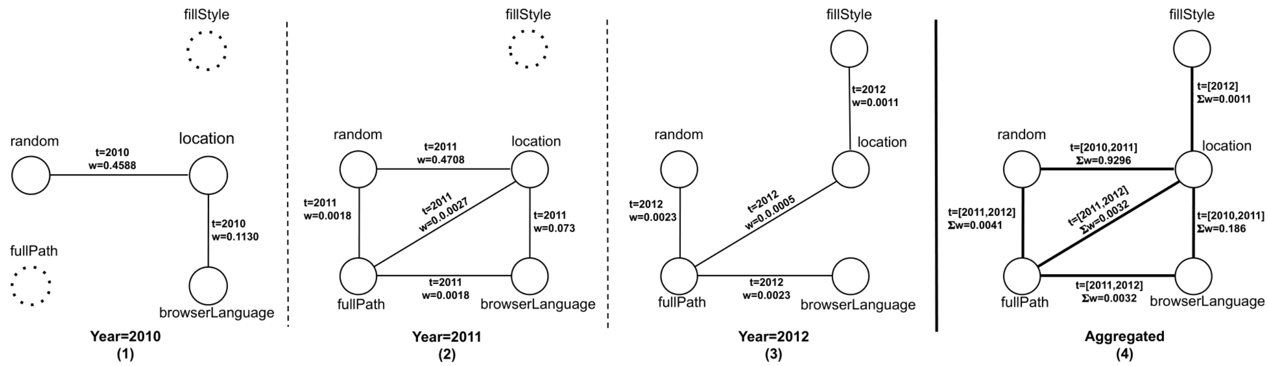

Fig. 3. Creation of a sample temporal graph representation over years. (1), (2) and (3) represent the sub graphs for years 2010, 2011, and 2012, respectively. (4) represents the aggregated temporal graph.

in predicting the future connectivity between web APIs in temporal API co-occurrence graphs. Table 1 lists the information gain of hand-crafted features. It can be seen from the table that almost all feature provide an information gain of at least 5% and the top three features provide the information gain of more than 12%. Overall, information gain indicates that the hand-crafted features are generic enough to be used for predicting future connectivity between web APIs in temporal API co-occurrence graphs.

Features	Information gain (%)
Leicht Holme Newman Index	16.52 ± 1.63
Temporal Edge Weight	13.42 ± 3.48
Edge Weight	12.06 ± 3.16
Salton	8.61 ± 1.77
Resource Allocation	8.5 ± 1.94
Average Degree	7.8 ± 2.19
Sorensen Similarity	6.43 ± 1.58
Jaccard	6 ± 1.6
Common Neighbiors	5.74 ± 1.51
Hub Depressed	5.5 ± 1.31
Adamic-Adar	4.84 ± 1.19
Hub Promoted	4.57 ± 2.07

Table 1. Hand-crafted features used by FP-Radar for graph prediction and information gain values (averaged over 10 years)

Graph embedding-based features. We capture more nuanced API co-occurrence patterns, potentially not modeled by our hand-crafted features, through graph embeddings. Graph embeddings encapsulate a node's neighborhood in a vector representation, such that the similar nodes in the graph have similar vector representation [33, 80]. We determine a node's neighborhood through a series of biased random walks. Specifically, the random walks respect time order, i.e., edges are traversed in ascending order of time, and recently formed edges are selected with higher probability. Once a node's neighborhood is determined, it is mapped to an embedding space, such that the embeddings of two nodes that are similar to each other in the graph also have similar embeddings. After creating the node embeddings, we combine the emdeddings of a node pair using a weighted L2 regularization [33].

Edge Prediction. We use a random forest [8] machine learning ensemble to predict the JavaScript APIs future co-occurrence. Random forest combines the decisions from several decisions trees, each trained on a subset of features selected at random, and outputs the majority decision. We configure a random forest ensemble with 100 decision trees.

Each node in the decision tree is split using the best feature, based on information gain, among the subset of features. We note that our classes are imbalanced, i.e., API pairs are far less likely to not co-occur than they are to co-occur. Thus, we bias our model by down sampling no-occurrence instances to the half of co-occurrence instances. Penalizing the model allows us to predict the APIs co-occurrence more favorably.

We predict the API co-occurrence over the year, i.e., from 2010–2020, by iteratively building the temporal graph. Specifically, as we move forward in time, our temporal graph contains APIs co-occurrences from all the snapshots thus far. For example, for year 2010, our temporal graph only contains API co-occurrence that existed in year 2010, however, for year 2014, the temporal graph contains the API co-occurrence that existed between years 2010 and 2014. For each year Y, we treat all possible API pairs, from the temporal graph of the last year G_{Y-1}, as probable candidates that may co-occur in the current year. The actual co-occurrence between the APIs in the current year Y, is considered as ground truth. We then use this information to train FP-Radar's random forest ensemble. Once we train the model, we use it to predict the future graph in the following year, i.e., year $Y+1$. Specifically, we treat all possible API pairs, from the temporal graph of current year G_Y, as probable candidates that may co-occur next year. Since we are retrospectively predicting the APIs co-occurrence, we are in a unique position to also validate the predicted APIs co-occurrence that would happen in the future, i.e., by using the future API co-occurrence in year $Y+1$. It is noteworthy that we combine both hand-crafted and graph embedding based features to train a combined random forest ensemble.

Results. Table 2 presents FP-Radar's accuracy in predicting API co-occurrence over the years. We provide separate as well as the combined accuracy of hand-crafted and graph embedding-based features. Table 2 shows that FP-Radar's accuracy is significantly improved when hand-crafted and graph embedding-based features are combined together. Specifically, the average accuracy, over the years, for hand-crafted features is 87.5% and graph-embedding based features is 76.24%. When combined together, the mean accuracy increases to 88.13%.

3.3 Clustering API Temporal Graphs

FP-Radar's temporal graphs allow us to longitudinally investigate the evolution of web API co-occurrence. To this end, we partition temporal API graphs into clusters to systematically analyze APIs that are used for similar functionality together. FP-Radar clusters the graphs based on the Louvain method [6], which partitions the graph such that the modularity is maximized between clusters. If a cluster contains more than one-third of the API keywords, FP-Radar partitions it again into sub-clusters.

FP-Radar clusters temporal API co-occurrence graphs and links clusters across consecutive years together to form temporal clusters. Specifically, FP-Radar links clusters together if their Jaccard similarity in more than 20%. If more than one cluster meets the similarity threshold in the prior year, they are merged together in the following year. If a cluster from prior year matches more than one clusters in the following year, it is attached to all of the clusters in the following year. If none of the clusters from prior years meet the similarity threshold, a new temporal cluster is created in the following year. Cluster from prior years that do not get attached to the clusters in the following year, may get attached to clusters in the coming years. Short-lived clusters, with a lifespan of at most 2 years, are filtered because they do not capture meaningful longitudinal trends. Finally, FP-Radar extracts 14 temporal clusters.

Jaccard similarity threshold. Figure 4 plots the trade-off between the number of short-lived clusters and the merging and splitting of clusters with varying Jaccard similarity threshold. It can be seen from the figure that as the similarity threshold increases, the number of short-lived clusters increases and the number of merging and splitting of clusters reduces. We pick 20% as a threshold to link clusters, across consecutive years, because it provides the best trade-off. If we pick a higher similarity threshold, we risk losing a significant number of APIs, present in the short-lived clusters, and also risk merging clusters with varying functionality, together.

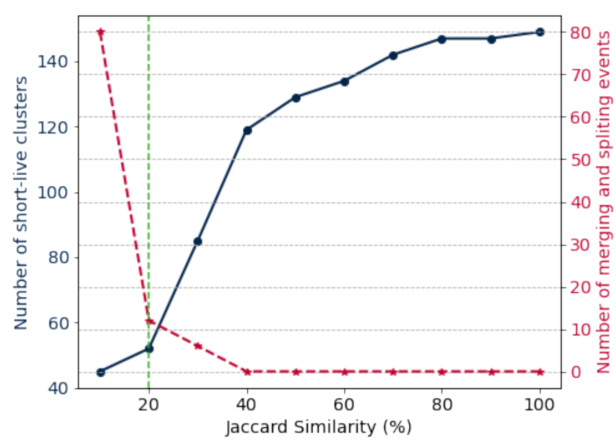

Fig. 4. Number of short-lived temporal clusters along with number of merging and splitting events with different similarity thresholds.

3.4 Labeling Temporal Clusters

We next semi-automatically analyze the temporal clusters to label them. To this end, we expect functionally related APIs to appear together in a temporal cluster. To map keywords to their respective interfaces and APIs, we use MDN's [69] hierarchical taxonomy of 88 APIs and 1024 interfaces. We then identify the dominant APIs of each temporal cluster using this taxonomy. Specifically, we measure the dominance of an API in a cluster as the fraction of its keywords that exist in the cluster.

Year	# of Nodes	# of Edges	Hand-crafted			Graph embeddings			Combined		
			Accuracy	Precision	Recall	Accuracy	Precision	Recall	Accuracy	Precision	Recall
2012	1,170	226,013	87.12 %	78.90%	71.10%	79.40%	82.93%	86.99%	88.40%	89.50%	98.40%
2013	1,354	310,954	91.12%	84.25%	77.35%	77.63%	79.16%	90.13%	86.50%	91.80%	93.10%
2014	1.896	564,448	86.20%	89.40%	63.80%	76.18%	71.93%	79.20%	90.10%	96.0%	93.40%
2015	2,096	746,379	86.50%	81.30%	72.20%	71.93%	73.66%	90.76%	87.30%	89.70%	96.50%
2016	2,599	1,286,524	85.90%	82.10%	73.30%	73.28%	75.16%	89.50%	87.70%	97.60%	89.20%
2017	2,978	1,669,328	87.30%	81.14%	83.30%	74.75%	75.91%	91.01%	91.10%	94.80%	95.70%
2018	3,409	2,241,026	87.96%	80.90%	83.4%	74.42%	75.87%	90.35%	90.30%	95.40%	94.20%
2019	3,603	2,684,157	88.41%	80.30%	81.20%	78.14%	83.08%	84.41%	91.70%	92.70%	98.80%
Mean	2,385	1,216,103	87.58%	82.29%	75.71%	75.71%	78.13%	90.08%	89.13%	93.44%	94.91%

Table 2. FP-Radar's accuracy in predicting APIs co-occurrence with hand-crafted and graph embedding-based features.

Labeling the fingerprinting cluster. Since different fingerprinting techniques are often used together [17, 25, 41, 54], we expect that the web APIs abused for fingerprinting will be partitioned in a separate temporal cluster. To label the fingerprinting cluster, we analyze the following fingerprinting metrics for each of the 14 temporal clusters:

1. Percentage of API keywords that appear in fingerprinting scripts reported by [41].
2. Percentage of API keywords that are used in the open-source fingerprintjs2 fingerprinting library containing 152 API keywords [1].
3. Percentage of API keywords that *only* appear in known fingerprinting scripts reported by [41] (i.e., not in any non-fingerprinting scripts).
4. Ratio of the fraction of API keywords that appear in fingerprinting scripts to that in non-fingerprinting scripts as reported by [41].

Note that FP-Radar partially relies on FP-Inspector [41] to label the fingerprinting cluster. However, we argue that it is the best available ground truth for browser fingerprinting, as compared to using other alternatives such as filter lists. Disconnect [22] only provides the domain names of fingerprinting vendors, rather than the full URLs of fingerprinting scripts, and thus cannot distinguish between fingerprinting and non-fingerprinting resources served from the same domain.

Results. Table 3 shows the temporal clusters and their key characteristics. Each row represents a cluster and the rows are sorted based on the cluster size. We note that the top-ranked cluster clearly has significantly more pronounced fingerprinting metrics than other clusters, we label it as fingerprinting and the remaining as other. First, 63% of the keywords in the fingerprinting cluster are used in fingerprinting scripts, which is at least ≈3X more than any other temporal cluster. Second, 36% of the keywords in the fingerprinting cluster are used in fingerprintjs2, which is at least ≈6X more than any other temporal cluster. Third, 13% of the keywords exclusively appear in fingerprinting scripts, which is at least ≈4X more than any other temporal cluster. Finally, the fraction of the keywords appearance in fingerprinting to non-fingerprinting scripts is 26.85, which is ≈4X more than any other temporal cluster.

4 Analysis of APIs in the Fingerprinting Cluster

In this section, we conduct an in-depth analysis of the fingerprinting cluster detected by FP-Radar. Table 4 lists a subset of the keywords of top dominant APIs in the fingerprinting cluster.[3] We investigate how the functionality of dominant APIs is being abused for fingerprinting. We also assess the time-to-detection of FP-Radar as compared to their browser release and disclosure dates. For each API keyword, we define release, disclosure, and detection dates as follows:

1. *Release* refers to the earliest date of support by one of the major browsers (i.e., Chrome, Firefox, Safari).
2. *Appearance* refers to the earliest date when the API keyword appeared in our dataset.
3. *Disclosure* refers to the earliest date that a proof-of-concept fingerprinting design or implementation involving the API keyword was presented in a research publication, W3C documentation, or public forums.

3 We select a representative subset out of 313 total keywords to capture diverse use cases and cover almost all of the time-to-detection categories for each API. We will include the complete table along with the code/data as part of the artifact release.

Cluster size	Life-span in years	% keywords in FP scripts	% keywords in fpjs2 [1]	% keywords in only FP scripts	FP/Non-FP ratio	Dominant APIs
313	9	63%	36%	13%	26.85	Battery, Navigator, Network Information
313	5	23%	2%	0%	6.06	Long Tasks, Resource Timing, Background Tasks
256	4	15%	1%	0	1.43	XMLSerializer, Mouse, ShadowRoot
222	6	17%	6%	2%	3.13	Mouse, TouchEvents, Canvas
161	4	11%	3%	2%	0.89	CSS Painting, XMLHTTPRequest
143	4	11%	3%	0	0.79	VideoTrack, Geolocation, Long Tasks
142	10	20%	5%	0	0.61	HTMLIFrameElement, Navigator, URL
141	5	15%	4%	0	1.19	Visual Viewport, Crypto, Channel Messaging
125	4	10%	2%	0	0.25	Fetch API, Notification, NodeFilter
121	9	18%	4%	0%	1.24	Resource Timing, Page Visibility, History
92	5	20%	2%	0%	0.98	FullScreen, VideoTrack, HTMLMediaElement
78	4	3%	<1%	3%	1.94	FileReader, Web Animations, XMLHttpRequest
67	4	3%	2%	0	0.27	Sensors, Gamepad, Fullscreen, Web Bluetooth
28	4	2%	1%	0	0.00	History, HTMLElement, HTMLTableElement

Table 3. Temporal clusters detected by FP-RADAR and their key characteristics. Based on their fingerprinting potential, clusters are marked with different gradients of red. The fingerprinting cluster (represented by dark red) clearly stands out as compared to the remaining clusters in terms of its similarity with known fingerprinting scripts.

4. *Detection* refers to the earliest date when the API keyword was detected as a member of the fingerprinting cluster by FP-RADAR.

Based on this information, we classify each API keyword in the fingerprinting cluster into the following 4 categories:

1. FP-RADAR detects abuse of API-keywords that are yet **undisclosed** to the best of our knowledge. Denoted with green color in Table 4, FP-RADAR detects a number of yet-undislosed API keywords such as `deviceMemeory` (Navigator), `WebGL2RenderingContext` (WebGL), `illuminance` (Sensor), and `paint` (Performance).
2. FP-RADAR detects abuse of APIs **before disclosure**. Denoted with yellow color in Table 4, FP-RADAR detects a number of API keywords before their disclosure such as `getGamepads` (GamePad), `visibilityState` (Page Visibility), and `clipboardData` (Clipboard).
3. FP-RADAR detects abuse of APIs **after disclosure**. Denoted with red color in Table 4, FP-RADAR detects some API keywords after their disclosure such as `longitude` (Geolocation), `DeviceMotionEvent` (Sensor), and `plugins` (Navigator).
4. FP-RADAR detects abuse of APIs **at disclosure**. Denoted with orange color in Table 4, FP-RADAR detects a number of API keywords at their disclosure such as `chargingTime` (Battery Status), `now` (Performance), and `force` (Touch). Note that most of the late detections are in fact detected as early as possible by FP-RADAR because the API keywords did not appear in our data before the detection date. In other words, FP-RADAR detects these API keywords at the first possible opportunity. We also denote these with orange color in Table 4 and include API keywords such as `altitudeAccuracy` (Geolocation), `bufferData` (WebGL), and `chargingchange` (Battery Status).

Next, we do a manual deep dive into each of the APIs listed in Table 4 in the descending order of their dominance. Note that we do not discuss some of the known web APIs, such as `canvas` and `canvas font`, `webRTC`, `AudioContext`, that are already shown to be widely abused for browser fingerprinting [2, 25].

`Battery Status`, standardized in 2011 [49] and supported in major browsers as early as 2014, is a non-permissioned API that provides information about a device's battery status to help web applications adjust resource usage when battery power is low. In 2015, Olejnik et al. disclosed that battery capacity and charging level can be abused for fingerprinting [78].[4] More specifically, the information about current battery level (`level`) and predicted time to charge (`chargingTime`) or discharge (`dischargingTime`) can be used to estimate a device's battery capacity, which is lower than its design capacity and often distinctive. FP-RADAR detects these keywords in 2015, right at the

4 Due to these fingerprinting concerns [74], Firefox stopped supporting the API in 2017 [11].

API Name	Keywords	Release Date	Appearance Date	Disclosure Date	Detection Date
Battery Status	chargingTime	2014	2015	2015 [78]	2015
	chargingchange	2014	2017	2015 [78]	2017
	dischargingTime	2014	2016	2015 [78]	2016
Navigator	deviceMemory	2017	2017	-	2019
	hardwareConcurrency	2014	2014	2017 [87]	2014
	oscpu	2004	2015	2009 [57]	2015
	plugins	2003	2011	2009 [57]	2016
	vendorSub	2004	2013	-	2013
	webdriver	2015	2014	-	2017
Network Information	downlink	2017	2018	2020 [32]	2018
	downlinkMax	2017	2018	2020 [32]	2019
	rtt	2017	2017	2020 [32]	2017
Geolocation	altitudeAccuracy	2009	2016	2008 [82]	2016
	geolocation	2009	2012	2008 [82]	2011
	longitude	2009	2012	2008 [82]	2016
	watchPosition	2009	2014	2008 [82]	2019
WebGL	bufferData	2011	2014	2012 [72]	2014
	viewport	2011	2012	2012 [72]	2018
	webgl	2011	2011	2012 [72]	2011
	WEBGL_debug_renderer_info	2014	2014	2016 [25]	2014
	WEBGL_depth_texture	2013	2017	-	2017
	WebGL2RenderingContext	2017	2017	-	2017
Performance	domainLookupEnd	2015	2015	-	2016
	domainLookupStart	2015	2015	-	2016
	now	2012	2012	2016 [40]	2016
	paint	2017	2018	-	2019
Page Visibility	visibilityState	2013	2013	2020 [35]	2017
	focused	2013	2013	2020 [35]	2013
	prerender	2013	2013	-	2016
Web Worker	applicationCache	2010	2011	2017 [109]	2011
	Worklet	2018	2018	-	2018
GamePad	Gamepad	2014	2014	2020 [14]	2019
	getGamepads	2014	2014	2020 [14]	2014
	mapping	2014	2015	-	2017
Mouse	movementX	2014	2016	2013 [92]	2016
	onmousemove	2003	2012	2004 [84]	2018
Touch	force	2012	2013	2013 [92]	2013
	ontouchstart	2011	2011	2013 [92]	2011
	rotationAngle	2015	2017	-	2017
	touchenter	2012	2015	2013 [92]	2016
Sensor	AbsoluteOrientationSensor	2018	2018	-	2018
	AmbientLightSensor	2017	2017	-	2017
	acceleration	2011	2013	2014 [7]	2017
	DeviceMotionEvent	2014	2013	2014 [7]	2018
	illuminance	2017	2017	-	2018
	Magnetometer	2017	2017	-	2018
	rotationRate	2011	2017	-	2018
Clipboard	copy	2007	2018	2020 [37]	2019
	clipboardData	2013	2018	2020 [37]	2018
	paste	2007	2018	2020 [37]	2019

Table 4. List of dominant API detected by FP-Radar and their time-to-detection: not-yet-disclosed early detection on-time detection late detection.

time of disclosure. Furthermore, FP-Radar detects a change in the abuse of `Battery Status` API staring 2017. More specifically, fingerprinters started gathering the change frequency of the battery status using keywords such as `chargingchange`, `chargingtimechange`, `dischargingtimechange`, and `levelchange` that reflect different workloads to create short-lived fingerprint. Script 2 shows a fingerprinting snippet that uses the aforementioned keywords.

`Navigator`, standardized in 1997 and supported by all major browsers since then [62], is a non-permissioned interface that provides information about the browser. In 2009, Mayer [57] disclosed that the navigator object provides information about a browser's settings that can be abused for fingerprinting. More specifically, the user agent string (`userAgent`), the languages supported by the browser (`languages`), the list of installed plugins (`plugins`), and supported file formats (`mimeType`) can reveal distinctive information about a browser. Since these features individually might not be sufficient to uniquely identify a browser, fingerprinters tend to gather a number of device-specific information exposed by navigator to increase the entropy of the fingerprint [54]. Script 3 shows a fingerprinting snippet that gathers 18 different navigator properties including the aforementioned keywords. FP-Radar detects navigator-related keywords as early as 2013, which is roughly around the time when researchers first documented fingerprinting on the web through large-scale measurements [76]. Note that the `Navigator` interface has been updated several times over the years to support new features. FP-Radar is able to detect the abuse of most of the newly introduced navigator properties in a timely fashion. For example, FP-Radar detects `hardwareConcurrency`, which returns the available number of logical processor cores, in 2014 right after its standardization even though its abuse was disclosed later in 2017 [87].

`Network Information` API, standardized in 2014 [51] and supported by major mobile browsers (except Safari) since 2017 [63], is a non-permissioned API that provides network connection information of the browser. More specifically, connection type (`type`, such as WiFi, WiMAX, Ethernet), delay (`rtt`), bandwidth (`downlink` and `downlinkMax`), and change in connection type (`onchange`) information are accessible via this API. It is noteworthy that potential privacy issues of the Network Status API were originally dismissed by W3C ("minimal impact on privacy or fingerprinting") [50] and none of the prior fingerprinting measurement studies report its abuse [10, 24, 57, 72, 78, 100]. However, as later acknowledged by W3C in 2020 [32], this information could be abused to fingerprint a user based on the time and order of transitions between networks as well as user location. Note that Firefox and Safari explicitly declined to support this API due to fingerprinting concerns [12, 94]. FP-Radar is able to detect these keywords as soon as 2017, right at their release date but before their disclosure. Script 4 shows an example fingerprinting snippet that collects all of the aforementioned network properties.

`Geolocation` API, standardized in 2008 [82] and supported in all major browsers around 2009, is a permissioned API that provides information about geographical location of device including (`latitude`, `longitude`, `altitude`, `speed`), as well as the accuracy of the acquired location data (`altitudeAccuracy`), and whenever the position of the device changes (`watchPosition`). The information exposed by the Geolocation API can be abused for fingerprinting due to its high precision (a double representing the position in decimal degrees). Note that the Geolocation API was permissioned from the very beginning because of the obvious privacy concerns acknowledged by W3C [82]. FP-Radar detects these keywords as early as 2011, at the earliest formation of the fingerprinting cluster. Note that the permission status (i.e., whether or not the user has granted permission) itself reveals one bit of information that can be combined with other fingerprinting features. FP-Radar detects the abuse of `PERMISSION_DENIED` and `POSITION_UNAVAILABLE` in 2016. Script 5 shows a fingerprinting snippet that gathers the aforementioned geolocation information, in addition to other fingerprinting information.

`WebGL` API, standardized in 2010 [48] and supported in all major browsers soon afterwards, is a non-permissioned API that can render interactive3D objects in the browser and manipulate them through JavaScript. WebGL API can be abused for fingerprinting in two main ways. First, WebGL can be used to list all WebGL capabilities to build a fingerprint. For example, scripts can check for WebGL support using `window.WebGLRenderingContext` and `getContext('webgl')` and list capabilities such as `SHADING_LANGUAGE_VERSION` or `WEBGL_debug_renderer_info`. Second, WebGL can be used to render a canvas image (using `WebGLRenderingContext.canvas`) that is then encoded and hashed (using `toDataURL`) to build a fingerprint. The rendering varies across devices due to differences in the rendering pipeline that involves the operating system, web browser, rendering engine, graphics driver, and the underlying hardware. Note that WebGL 1 [21] was extended to WebGL 2 [20] in 2017 to include new capabilities such as pixel buffer objects (`GetBufferSubData`), Primitive restart (`draw_primitive_restart`), and rasterizer discard (`RASTERIZER_DISCARD`). FP-Radar detects the keywords associated with WebGL 1 as early as 2011 and WebGL 2 as early as 2017.

`Performance` API, standardized in 2011 [45] and supported in all major browsers around 2012, is a non-permissioned API that covers `Performance Timeline`, `Resource Timing`, `Performance Timeline`, `Navigation Timing`, `Resource Timing`, and `Paint Timing`. It allows scripts to accurately measure various performance-related metrics during the page load such as DNS using `domainLookupStart` and `domainLookupEnd` or HTTP using `fetchStart`, `requestStart`, `responseStart`, and `responseEnd`. However, access to high resolution timing information (up to sub-millisecond granularity) can be abused for fingerprinting by precisely timing certain operations that depend on the underlying software/hardware pipeline [107]. For example, in [88] they measured clock difference on a device using combination of `Performance` and `Crypto` API. Specifically, they used `performance.now` to time the execution of the pseudo-random generator (`getRandomValues`) to create a browser fingerprint. FP-Radar detects most of the associated keywords as early as 2016. `Paint Timing` API is a recent addition to *Performance* API and has been supported by Chrome since 2017 and in other major browsers since 2020 [64]. This API measures the time it takes between the moment a user navigate to a URL and the moment a pixel renders on a screen (e.g., `first-paint` or `first-contentful-paint` representing time between navigation start `performanceEntry.startTime` and when the browser renders any/content pixel, respectively). This timing information can be distinctive across different browsers based on differences in their underlying compute/communication performance. FP-Radar captures the abuse of Paint Timing API in 2019, the first time it appears in our data. Script 7 shows a fingerprinting snippet that measures the First Time to Paint and First Contentful Paint in addition to other fingerprinting information.

`Page Visibility` API, standardized in 2011 [44] and supported in all major browsers by 2013, is a non-permissioned API. This API provides access to determine the visibility state (i.e. `visible`, `hidden`, and `prerender`) or be notified when the visibility state of a document changes. While the visibility state (or the change in visibility state) is not directly useful for fingerprinting, but it can be abused for `ephemeral fingerprinting` [35] when the changes in page visibility state can be correlated across different sites. Specifically, when a user switches between a pair of tabs/windows then a change in the visibility state will be simultaneously triggered for both tabs/windows. This information can be correlated by a script on both tabs/windows to link whether the tabs/windows are on the same browser/device. For example, Script 1 measures timestamps of the changes in page visibility state. It uses `Date.now` to log the exact time the page visibility state changes (`onvisibilitychange`). The sequence of timestamps when the page visibility state changes is expected be the same and distinctive across all of the co-visible sites in a user's browser/device. Thus, it can be used to build a cross-site ephemeral fingerprint. Disclosed in 2020 [35], FP-Radar first detects the abuse of this API in 2017.

```
...
// Register an event that will be
// triggered on visibility state change.
document.addEventListener('visibilitychange'
    , VisibilityStateHandler);

// return visibility state of the page
function getVisibilityState() {
    return document.visibilityState;
}

// return current time
function getCurrentTime() {
    return Date.now();
}

// Capture current time & visibility state.
function VisibilityStateHandler() {
    ...
    VisibilityStateFP = {
      VisibilityState: getVisibilityState(),
      CurrentTime: getCurrentTime()
    };
    ...
}
...
```

Script 1. Simplified version of a script that uses the `Visibility` API to conduct ephemeral fingerprinting. Each time the visibility state changes, it is recorded with the current timestamp.

`Web Worker` API, standardized in 2009 [36] and supported in all major browsers by 2010, is a non-permissioned API. This API allows sites to run heavy processing scripts in the background without affecting the performance of the main page. Although DOM and Window objects are not accessible to `Web Worker` API, however, they do have access to a number of other APIs including `WebGL`. `Workers` can be used to run a fingerprinting technique (e.g., Canvas fingerprinting using `OffscreenCanvas` [29]) in a background thread separate from the main execution thread of a web application without making the main thread slow or blocked. We have not detected such a scenario in our dataset of scripts. However, FP-Radar detects the presence of this API as early as 2011 where scripts simply probe the support status of this API (e.g., using `window.Worker`) alongside other fingerprinting information. FP-Radar also detects `SharedWorkerGlobalScope.applicationCache` in a number of scripts as the cache of the `worker` that allow scripts to set and get client-side state as an alternative to cookies.

`Gamepad` API, standardized in 2014 [91] and supported in all major browsers since then is a non-permissioned API that allows browsers to connect to gamepads. `getGamepads` method returns the list of `Gamepad` objects as well as their configuration such as `axes`, `buttons`, `displayId` or `hand`. Probing whether a browser has a connected `Gamepad` and, if there is one connected, collecting the aforementioned configuration information can reveal distinctive information about a browser. Due to its potential privacy threats, starting 2020, Mozilla requires thirds-party iframes to ask for permission before calling the `getGamepads` method [14]. FP-Radar detects these keywords as early as 2014, right after it was supported in major browsers, even though it was disclosed 6 years later. Script 8 shows a fingerprinting snippet from 2014 that probes the presence of `Gamepad` API and calling the `getGamepads` method in addition to collecting other fingerprinting information.

`Mouse`-related interfaces, including `MouseEvent`, `WheelEvent`, `MouseScrollEvent`, `MouseWheelEvent`, and `Pointer Lock`, was first introduced in 2004 and has since been updated to support new features. It can capture coordinates of a pointing device's (such as a mouse) including `clientX/Y`, `pageX/Y`, `offsetX/Y`, `movementX/Y` in addition to its events such as `click`, `dblclick`, and `mousemove` without granting any permission. Beginning as early as 2004 [84], there has been a steady stream of studies demonstrating how mouse movements can be used to identify users [99]. FP-Radar first detects the abuse of mouse-related keywords for user behavior fingerprinting in 2016 and since then has detected other properties such as `movementX/Y`, `deltaX/Y/Z`, and `wheelDelta`. Script 9 shows an example fingerprinting snippet that collects mouse movement information in addition to other fingerprinting information.

`TouchEvent` interface, standardized in 2011 [67] and supported by all major browsers including mobile version of browsers since 2013.[5] `TouchEvent` is a non-permissioned interface that is similar to mouse interfaces except that it supports simultaneous touches and at different locations on the touch surface. Beginning as early as 2013 [92], there has been a steady stream of studies demonstrating how touch events can be use to identify users [99]. Specifically, frequency of tapping (captured by events such as `ontouchstart`, `touchenter`, `touchleave`, and `touchmove`), and strength of tapping (captured by `force`) can be used for user behavior fingerprinting. FP-Radar first detects the abuse of touch-related keywords for user behavior fingerprinting in 2011 and since then has detected other properties such as `rotationAngle`.

`Sensor` APIs, standardized in 2012 [102] and is only supported in Chrome since 2017. Privacy-oriented browsers like Firefox and Safari have declined to implement this API due to privacy concerns [65, 94]. It is a permissioned API that provides sensor information such as light intensity (using `AmbientLightSensor`) and the force caused by vibration or a change in motion (using `Accelerometer`. Older interfaces such as `DeviceMotionEvent` and `DeviceOrientationEvent`, which are not part of `Sensor` AIP but implemented by all major browsers (except Safari) since 2011 [60], provide non-permissioned access to a subset of sensors related to a device's position and orientation. The information exposed by these APIs and interfaces has been shown to be used for user behavior fingerprinting [7, 98, 104]. FP-Radar detects the sensor keywords associated with `DeviceMotionEvent` starting from 2017. Although in the previous studies, the sensor data was collected using `DeviceMotionEvent`, we detect the abuse of `Sensor` API that are not supported by `DeviceMotionEvent` and `DeviceOrientationEvent`. For example, FP-Radar detects the abuse of `AmbientLightSensor` and `illuminance` that are not yet disclosed. Script 10 shows an example fingerprinting snippet that collects sensor information in addition to other fingerprinting information.

`Clipboard` API, standardized in 2015 [97] but not supported as early as 2018 by major browsers [59], implements clipboard operations such as `copy`, `cut`, and `paste`. Moreover, if a user grants permission, it provides asynchronous access to read and modify the contents of the system clipboard using `read` (or `readText` or `clipboardData.getData('Text')`) and `write` (or

5 Desktop version of Firefox started supporting this interface lately in 2017 [66].

writeText) methods. However, since this API can access the clipboard data, there are serious privacy concerns due to the possibility that the clipboard might contain personally identifiable information (PII) [37]. FP-Radar detects the abuse of Clipboard API as early as 2018. Script 6 shows an example fingerprinting snippet that collects clipboard information in addition to other fingerprinting information.

Validation. We sift through public disclosures to validate the fingerprinting potential and abuse of APIs listed in Table 4. Figure 5 shows the breakdown of disclosed and undisclosed APIs along with FP-Radar's detection time. We find that the 44% of API keywords detected by FP-Radar are still publicly undisclosed. We try to validate the remaining undisclosed detections by comparing with DuckduckGo's recently released list of fingerprinting APIs [19]. We note that FP-Radar detects more than 80% of fingerprinting API keywords detected by DuckDuckGo. However, 90% of keywords detected by FP-Radar, including several well-known fingerprinting APIs, such as `Battery.changingTime`, `Geolocation.geolocation`, and `WebGL.WEBGL_debug_renderer_info` are still undetected by DuckDuckGo. DuckDuckGo primarily misses the remaining keywords because it detects a limited number of APIs, i.e., 96, using a very simple heuristic that uses the ratio of an APIs appearance in "suspicious" script to an "non-suspicious" script to label them as fingerprinting (more details in Section 2).

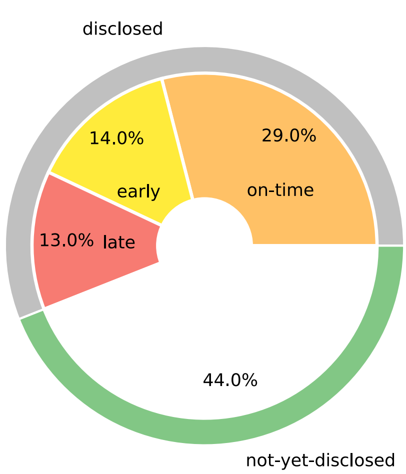

Fig. 5. Breakdown of disclosed and undisclosed APIs along with FP-Radar's detection time.

5 Limitations

In this section, we discuss some of the limitations of FP-Radar's pipeline including completeness of retrospective measurements, robustness of the analysis technique, and ground truth assessment of fingerprinting scripts and fingerprinting techniques.

Measurements. FP-Radar relies on the Wayback Machine for retrospective longitudinal measurements of browser fingerprinting. As we discuss in Section 3.1.1, the Wayback Machine's archiving process has limitations that lead to potentially incomplete coverage. Unfortunately, to the best of our knowledge, there is no other publicly available service that archives complete historical version of webpages. HTTP Archive [38] is a related project that archives millions of URLs each month. However, it does not store the response bodies of all of the resources [31] and the downloadable data is only available for the last 6 year, i.e., 2016 to 2021 [39]. Given the democratization of large-scale web crawling tools and capabilities, going forward, future work can consider conducting live crawls to complement missing resources in archiving services such as the Wayback Machine or HTTP Archive.

Robustness. FP-Radar relies on static analysis of JavaScript code snippets, i.e., AST-based representation of scripts, to extract web API keywords. Relying on static analysis makes it challenging for FP-Radar to process obfuscated scripts and attribute some generic keywords to APIs. Specifically, some fingerprinting scripts use `eval`-based code obfuscation techniques [93] and some keywords are implemented by multiple APIs, e.g., `font` is implemented by `CanvasRenderingContext2D` [58] and `HTMLElement.style` [61]. We attempt to unpack obfuscated scripts by loading them in an instrumented browser and extracting scripts as they are parsed by the JavaScript engine. This approach is able to unpack scripts containing `eval` or `Function`, but does not fully address other more sophisticated obfuscation techniques [89]. While we do not fully address the keywords attribution issue because it is non-trivial to attribute a generic keyword to the calling API without executing the scripts, we mitigate this issue in our analysis by filtering generic keywords that belong to multiple APIs. To fully address these concerns, FP-Radar can be extended to include dynamic analysis as well; however, it suffers from code coverage issues that are non-trivial to address. Note that FP-Radar is not susceptible to obfuscation by a small number of scripts since it leverages tens of thousands of scripts to build its API keyword co-occurrence graph representation. Similarly, filtering a small number of generic keywords does not affect the correctness of the analysis.

Ground truth. Since FP-Radar uses unsupervised clustering, it relies on the classification of fingerprinting scripts [103], provided by FP-Inspector [41], to label the fingerprinting cluster. Since the classifications of FP-Inspector are not validated for scripts observed in prior years, we cannot solely rely on that as our ground truth while investigating fingerprinting techniques in Section 4. To mitigate this concern, we conduct manual analysis to validate the fingerprinting abuse of the APIs detected by FP-Radar. We also rely on a wide range of additional external sources including W3C documents, published research papers, and bug reports to assist with our manual analysis.

6 Conclusion

We presented FP-Radar, a machine learning approach for early detection of web API abuse for browser fingerprinting. FP-Radar advances the state-of-the-art in browser fingerprinting in two major ways. First, unlike prior work that is limited to analyzing a specific fingerprinting technique(s) at a particular point in time, FP-Radar conducts a retrospective longitudinal measurement study of browser fingerprinting over the last decade using the Wayback Machine. Second, unlike prior work that is limited to detecting deployment of already known fingerprinting techniques, FP-Radar is able to detect abuse of new methods of existing web APIs or new web APIs altogether by leveraging the aforementioned longitudinal measurements to model and cluster the evolution of API usage as a temporal graph. Most notably, FP-Radar detects novel types of user environment/hardware fingerprinting such as peripheral configuration via `Gamepad` and system capabilities via `Network Information` APIs as well as ephemeral fingerprinting of `Page Visibility` API even though it does not directly expose highly identifying information.

FP-Radar is able to detect the abuse of web API features before/at their disclosure, thus demonstrating its utility as an early detection system that can inform standards bodies and browser vendors interested in designing and deploying mitigations in a timely fashion. FP-Radar can complement prior approaches (e.g., [25, 41, 86]) to detect fingerprinting scripts by helping them adapt to new and evolving fingerprinting techniques. In addition to disclosing our findings to relevant entities, we plan to release FP-Radar's code and longitudinal measurement dataset artifacts to foster follow-up research. We also plan to collaborate with existing web tracking projects (e.g., [18, 83]) to develop a public-facing implementation that can leverage their live web crawls in the future.

Acknowledgment

This work is supported in part by the National Science Foundation under grant numbers 2102347, 2051592, 2103439, and 2127309 (Computing Research Association for the CIFellows 2021 Project).

References

[1] Modern & flexible browser fingerprinting library. https://github.com/Valve/fingerprintjs2.

[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 *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] F. Alaca and P. van Oorschot. Device Fingerprinting for Augmenting Web Authentication: Classification and Analysis of Methods. In *ACSAC*, 2016.

[5] P. Baumann, S. Katzenbeisser, M. Stopczynski, and E. Tews. Disguised Chromium Browser: Robust Browser, Flash and Canvas Fingerprinting Protection. In *ACM on Workshop on Privacy in the Electronic Society*, 2016.

[6] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, 2008(10):P10008, 2008.

[7] H. Bojinov, Y. Michalevsky, G. Nakibly, and D. Boneh. Mobile Device Identification via Sensor Fingerprinting. *arXiv preprint arXiv:1408.1416*, 2014.

[8] L. Breiman. Random Forests. In *Machine learning*, 2001.

[9] J. F. Brunelle, M. Kelly, H. SalahEldeen, M. C. Weigle, and M. L. Nelson. Not all mementos are created equal: Measuring the impact of missing resources. *International Journal on Digital Libraries*, 16(3):283–301, 2015.

[10] WebRTC Internal IP Address Leakage. https://bugzilla.mozilla.org/show_bug.cgi?id=959893.

[11] Remove web content access to Battery API. https://bugzilla.mozilla.org/show_bug.cgi?id=1313580, 2016.

[12] Bug 1372072 - Neutralize the threat of fingerprinting of network information API When 'privacy.resistFingerprinting' is true. https://bugzilla.mozilla.org/show_bug.cgi?id=1372072, 2017.

[13] E. Bütün, M. Kaya, and R. Alhajj. Extension of neighbor-based link prediction methods for directed, weighted and temporal social networks. *Information Sciences*, 463:152–165, 2018.

[14] M. Caceres. Securing Gamepad API. https://hacks.mozilla.org/2020/07/securing-gamepad-api/, 2020.

[15] 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.

[16] Y. Cao, S. Li, and E. Wijmans. (Cross-) browser fingerprinting via OS and hardware level features. In *NDSS*, 2017.

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

[18] DuckDuckGo's Tracker Radar. https://github.com/duckduckgo/tracker-radar/blob/3c82647d3a5ea16ab6408cad2a52ba4b72f66637/docs/FAQ.md.

[19] DuckDuckGo's Tracker Radar Detected Fingerprinting APIs. https://github.com/duckduckgo/tracker-radar/blob/main/build-data/generated/api_fingerprint_weights.json.

[20] J. G. Dean Jackson. WebGL 2 Specification. https://www.khronos.org/registry/webgl/specs/2.0/.

[21] J. G. Dean Jackson. WebGL specification. https://www.khronos.org/registry/webgl/specs/latest/1.0.

[22] Disconnect tracking protection lists. https://disconnect.me/trackerprotection.

[23] N. Doty. W3C Fingerprinting Guidance. https://w3c.github.io/fingerprinting-guidance.

[24] P. Eckersley. How unique is your web browser? In *International Symposium on Privacy Enhancing Technologies Symposium*, 2010.

[25] S. Englehardt and A. Narayanan. Online Tracking: A 1-million-site Measurement and Analysis. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[26] A. FaizKhademi, M. Zulkernine, and K. Weldemariam. Fpguard: Detection and prevention of browser fingerprinting. In *IFIP Annual Conference on Data and Applications Security and Privacy*, 2015.

[27] D. Fifield and S. Egelman. Fingerprinting web users through font metrics. In *International Conference on Financial Cryptography and Data Security*, pages 107–124. Springer, 2015.

[28] G. A. Fowler. Think you're anonymous online? A third of popular websites are 'fingerprinting' you. https://www.washingtonpost.com/technology/2019/10/31/think-youre-anonymous-online-third-popular-websites-are-fingerprinting-you/, 2019.

[29] E. Gasperowicz. OffscreenCanvas — Speed up Your Canvas Operations with a Web Worker. https://developers.google.com/web/updates/2018/08/offscreen-canvas, 2020.

[30] M. Graham. robots.txt meant for search engines don't work well for web archives. https://blog.archive.org/2017/04/17/robots-txt-meant-for-search-engines-dont-work-well-for-web-archives/, 2017.

[31] I. Grigorik. Quickstart guide to exploring the HTTP Archive. https://discuss.httparchive.org/t/quickstart-guide-to-exploring-the-http-archive/682.

[32] I. Grigorik. Network Information API. https://wicg.github.io/netinfo/, 2020.

[33] A. Grover and J. Leskovec. node2vec: Scalable feature learning for networks. In *KDD*, 2016.

[34] N. H. Hashim, J. Murphy, and P. O'Connor. Take me back: Validating the wayback machine as a measure of website evolution. In *Information and Communication Technologies in Tourism 2007*, 2007.

[35] A. Herath. Ephemeral Fingerprinting On The Web. https://github.com/asankah/ephemeral-fingerprinting, 2020.

[36] I. Hickson. Web Workers. https://www.w3.org/TR/2009/WD-workers-20090423, 2009.

[37] W. Hsieh. Async Clipboard API. https://webkit.org/blog/10855/async-clipboard-api, 2020.

[38] HTTP Archive. https://httparchive.org/.

[39] HTTP Archive Data. https://github.com/HTTPArchive/httparchive.org/blob/main/docs/gettingstarted_bigquery.md#understanding-how-the-tables-are-structured.

[40] J. M. Ilya Grigorik, James Simonsen. High Resolution Time Level 3. https://www.w3.org/TR/2016/WD-hr-time-3-20161031/#privacy-security, 2016.

[41] U. Iqbal, S. Englehardt, and Z. Shafiq. Fingerprinting the Fingerprinters: Learning to Detect Browser Fingerprinting Behaviors. In *Proceedings of the IEEE Symposium on Security & Privacy*, 2021.

[42] U. Iqbal, Z. Shafiq, and Z. Qian. The Ad Wars: Retrospective Measurement and Analysis of Anti-Adblock Filter Lists. In *IMC*, 2017.

[43] U. Iqbal, P. Snyder, S. Zhu, B. Livshits, Z. Qian, and Z. Shafiq. AdGraph: A Graph-Based Approach to Ad and Tracker Blocking. In *Proceedings of the IEEE Symposium on Security & Privacy*, 2020.

[44] A. J. Jatinder Mann. Page Visibility. https://www.w3.org/TR/2011/WD-page-visibility-20110602/, 2011.

[45] Z. W. Jatinder Mann. Performance Timeline. https://www.w3.org/TR/2011/WD-performance-timeline-20110811/, 2011.

[46] John Ross Quinlan. *Induction of decision trees*. Kluwer Academic Publisher, 1986.

[47] M. Kelly, J. F. Brunelle, M. C. Weigle, and M. L. Nelson. On the change in archivability of websites over time. In *International Conference on Theory and Practice of Digital Libraries*, pages 35–47. Springer, 2013.

[48] Khronos releases Final WebGL 1.0 specification. https://www.khronos.org/news/press/khronos-releases-final-webgl-1.0-specification, 2011.

[49] A. Kostiainen. Battery status event specification. https://www.w3.org/TR/2011/WD-battery-status-20110426/, 2011.

[50] M. Lamouri. The Network Information API. https://www.w3.org/TR/2012/WD-netinfo-api-20121129/#security-and-privacy-considerations, 2012.

[51] M. Lamouri. The Network Information API. https://dvcs.w3.org/hg/dap/raw-file/tip/network-api/Overview.html, 2014.

[52] P. Laperdrix, G. Avoine, B. Baudry, and N. Nikiforakis. Morellian Analysis for Browsers: Making Web Authentication Stronger with Canvas Fingerprinting. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2019.

[53] P. Laperdrix, B. Baudry, and V. Mishra. Fprandom: Randomizing core browser objects to break advanced device fingerprinting techniques. In *International Symposium on Engineering Secure Software and Systems*, pages 97–114. Springer, 2017.

[54] P. Laperdrix, N. Bielova, B. Baudry, and G. Avoine. Browser fingerprinting: A survey. *ACM Transactions on the Web*, 2020.

[55] A. B. Lassey. Combating Fingerprinting with a Privacy Budget Explainer. https://github.com/bslassey/privacy-budget.

[56] A. Lerner, A. K. Simpson, T. Kohno, and F. Roesner. Internet Jones and the Raiders of the Lost Trackers: An Archaeological Study of Web Tracking from 1996 to 2016. In *USENIX Security Symposium*, 2016.

[57] J. R. Mayer. Any person... a pamphleteer": Internet anonymity in the age of web 2.0. *Undergraduate Senior Thesis, Princeton University*, page 85, 2009.

[58] CanvasRenderingContext2D.font. https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D/font.

[59] Clipboard API. https://developer.mozilla.org/en-US/docs/Web/API/Clipboard_API.

[60] DeviceMotionEvent. https://developer.mozilla.org/en-US/docs/Web/API/DeviceMotionEvent.

[61] HTMLElement.style. https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement/style.

[62] Navigator - Web APIs: MDN. https://developer.mozilla.org/en-US/docs/Web/API/Navigator.

[63] Network Information API - Web APIs: MDN. https://developer.mozilla.org/en-US/docs/Web/API/Network_Information_API.

[64] PerformancePaintTiming. https://developer.mozilla.org/en-US/docs/Web/API/PerformancePaintTiming.

[65] Sensor APIs. https://developer.mozilla.org/en-US/docs/Web/API/Sensor_APIs.

[66] TouchEvent. https://developer.mozilla.org/en-US/docs/Web/API/TouchEvent.

[67] Touch Events Specification. https://www.w3.org/TR/2011/WD-touch-events-20110505, 2011.

[68] Battery Status API removed from Firefox. https://developer.mozilla.org/en-US/docs/Mozilla/Firefox/Releases/52#other_apis, 2016.

[69] MDN Web APIs. , https://developer.mozilla.org/en-US/docs/Web/API.

[70] G. Merzdovnik, M. Huber, D. Buhov, N. Nikiforakis, S. Neuner, M. Schmiedecker, and E. Weippl. Block Me If You Can: A Large-Scale Study of Tracker-Blocking Tools. In *IEEE European Symposium on Security and Privacy*, 2017.

[71] K. Mowery, D. Bogenreif, S. Yilek, and H. Shacham. Fingerprinting information in javascript implementations. In *Web 2.0 Workshop on Security and Privacy (W2SP)*, 2011.

[72] K. Mowery and H. Shacham. Pixel perfect: Fingerprinting canvas in html5. *Proceedings of W2SP*, 2012.

[73] How to block fingerprinting with Firefox. https://blog.mozilla.org/firefox/how-to-block-fingerprinting-with-firefox/.

[74] Removing the Battery Status API? https://groups.google.com/g/mozilla.dev.platform/c/5U8NHoUY-1k/m/9ybyzQIYCAAJ?pli=1, 2016.

[75] Nick Nikiforakis and Wouter Joosen and Benjamin Livshits. PriVaricator: Deceiving Fingerprinters with Little White Lies. In *WWW*, 2015.

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

[77] M. Nottingham. Unsanctioned Web Tracking. https://www.w3.org/2001/tag/doc/unsanctioned-tracking/, 2015.

[78] L. Olejnik, G. Acar, C. Castelluccia, and C. Diaz. The leaking battery: A privacy analysis of the HTML5 Battery Status API. In *Proceedings of the 10th International Workshop Data Privacy Management, and Security Assurance*, 2015.

[79] L. Olejnik, S. Englehardt, and A. Narayanan. Battery Status Not Included: Assessing Privacy in Web Standards. In *International Workshop on Privacy Engineering*, 2017.

[80] B. Perozzi, R. Al-Rfou, and S. Skiena. Deepwalk: Online learning of social representations. In *KDD*, 2014.

[81] M. Perry, E. Clark, S. Murdoch, and G. Koppen. Fingerprinting defenses in the tor browser. https://www.torproject.org/projects/torbrowser/design/#fingerprinting-defenses.

[82] A. Popescu. geolocation api specification. https://www.w3.org/TR/2008/WD-geolocation-API-20081222/, 2008.

[83] Princeton Web Transparency & Accountability Project. https://webtap.princeton.edu/.

[84] M. Pusara and C. E. Brodley. User re-authentication via mouse movements. In *2004 ACM workshop on Visualization and data mining for computer security*, 2004.

[85] N. Reitinger and M. L. Mazurek. Ml-cb: Machine learning canvas block. *Proceedings on Privacy Enhancing Technologies*, 2021.

[86] V. Rizzo, S. Traverso, and M. Mellia. Unveiling web fingerprinting in the wild via code mining and machine learning. *PETS*, 2021.

[87] T. Saito, K. Yasuda, K. Tanabe, and K. Takahashi. Web browser tampering: inspecting cpu features from side-channel information. In *International Conference on Broadband and Wireless Computing, Communication and Applications*, 2017.

[88] I. Sanchez-Rola, I. Santos, and D. Balzarotti. Clock around the clock: Time-based device fingerprinting. In *ACM CCS*, 2018.

[89] S. Sarker, J. Jueckstock, and A. Kapravelos. Hiding in Plain Site: Detecting JavaScript Obfuscation through Concealed Browser API Usage. In *ACM Internet Measurement Conference (IMC)*, 2020.

[90] J. Schuh. Building a more private web: A path towards making third party cookies obsolete. https://blog.chromium.org/2020/01/building-more-private-web-path-towards.html, 2020.

[91] T. M. Scott Graham. Gamepad. https://www.w3.org/TR/2014/WD-gamepad-20140225.

[92] M. Shahzad, A. X. Liu, and A. Samuel. Secure unlocking of mobile touch screen devices by simple gestures: You can see it but you can not do it. In *Proceedings of the 19th annual international conference on Mobile computing & networking*, 2013.

[93] P. Skolka, C.-A. Staicu, and M. Pradel. Anything to Hide? Studying Minified and Obfuscated Code in the Web. In *World Wide Web (WWW) Conference*, 2019.
[94] Apple Declined To Implement 16 Web APIs in Safari Due To Privacy Concerns . https://apple.slashdot.org/story/20/06/29/1456247/apple-declined-to-implement-16-web-apis-in-safari-due-to-privacy-concerns, 2020.
[95] P. Snyder, L. Ansari, C. Taylor, and C. Kanich. Browser feature usage on the modern web. In *Proceedings of the 2016 Internet Measurement Conference*, 2016.
[96] O. Starov and N. Nikiforakis. Xhound: Quantifying the fingerprintability of browser extensions. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 941–956. IEEE, 2017.
[97] H. R. M. Steen. Clipboard API and events. https://www.w3.org/TR/2015/WD-clipboard-apis-20151215/, 2015.
[98] H. M. Thang, V. Q. Viet, N. D. Thuc, and D. Choi. Gait identification using accelerometer on mobile phone. In *International Conference on Control, Automation and Information Sciences (ICCAIS)*, pages 344–348. IEEE, 2012.
[99] P. A. Thomas and K. P. Mathew. A broad review on non-intrusive active user authentication in biometrics. *Journal of Ambient Intelligence and Humanized Computing*, 2021.
[100] Tor browser canvas font fingerprinting. https://gitlab.torproject.org/legacy/trac/-/issues/13400.
[101] C. F. Torres, H. Jonker, and S. Mauw. FP-Block: Usable web privacy by controlling browser fingerprinting. In *ESORICS*, 2015.
[102] D. D. Tran. Sensor API Specification. https://dvcs.w3.org/hg/dap/raw-file/default/sensor-api/Overview.html, 2012.
[103] umar iqbal. FP-Inspector Code and Data. https://uiowa-irl.github.io/FP-Inspector/.
[104] T. Van Goethem, W. Scheepers, D. Preuveneers, and W. Joosen. Accelerometer-based device fingerprinting for multi-factor mobile authentication. In *International Symposium on Engineering Secure Software and Systems*. Springer, 2016.
[105] Wayback Machine API. https://archive.org/help/wayback_api.php.
[106] Wayback Machine. https://archive.org/web/.
[107] Y. Weiss. High Resolution Time, Privacy and Security. https://www.w3.org/TR/hr-time-3/#sec-privacy-security.
[108] J. Wilander. Full Third-Party Cookie Blocking and More. https://webkit.org/blog/10218/full-third-party-cookie-blocking-and-more/.
[109] J. Wilander. Full third-party cookie blocking and more. https://webkit.org/blog/10218/full-third-party-cookie-blocking-and-more, 2020.
[110] M. Wood. Today's Firefox Blocks Third-Party Tracking Cookies and Cryptomining by Default. https://blog.mozilla.org/blog/2019/09/03/todays-firefox-blocks-third-party-tracking-cookies-and-cryptomining-by-default/, 2019.
[111] S. Wu, S. Li, Y. Cao, and N. Wang. Rendered private: Making GLSL execution uniform to prevent WebGL-based browser fingerprinting. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*, 2019.
[112] Z. Yu, S. Macbeth, K. Modi, and J. M. Pujol. Tracking the Trackers. In *World Wide Web (WWW) Conference*, 2016.

```
// Battery Status API support probing.
if ('getBattery' in navigator) {
  BatteryManagerObj=navigator.getBattery()
  || navigator.battery()
  BatteryManagerObj.then(monitorBattery);
}
else {
  ChromeSamples.setStatus('not supported');
}

// Get battery level, charging,
// and discharging time.
function getStatus(battery) {
  return Math.floor(100 * battery.level)
}

// Trigger the function whenever
// the battery status changes.
function monitorBattery(battery) {
  // get the battery level
  getStatus(battery);

  // Monitor for further updates.
  ["chargingchange","chargingtimechange",
  "dischargingtimechange", "levelchange"].
  forEach(function(battery) {
    a.addEventListener(battery, null)
  })
}
```

Script 2. Simplified version of a script that uses the `Battery status` API for fingerprinting.

7 Appendix

We provide examples of actual fingerprinting snippets to support the discussion in the main text. We make minor revisions to the code to improve its readability. Note that all of the code snippets provided here use multiple fingerprinting techniques. However, we only show the relevant part of the code that is pertinent to our discussion in the main text.

```
function getUseragentData(t) {
  nvgtr_dict = {},
  nvgtr_dict.PX59 = navigator.userAgent,
  nvgtr_dict.PX61 = navigator.language,
  nvgtr_dict.PX313 = navigator.languages,
  nvgtr_dict.PX63 = navigator.platform,
  nvgtr_dict.PX86 = !!(navigator.doNotTrack
      || null === navigator.doNotTrack ||
      navigator.msDoNotTrack ||
      window.doNotTrack),
  nvgtr_dict.PX88 = getMimeType(),
  nvgtr_dict.PX169 = navigator.mimeTypes &&
      navigator.mimeTypes.length || -1,
  nvgtr_dict.PX62 = navigator.product,
  nvgtr_dict.PX69 = navigator.productSub,
  nvgtr_dict.PX64 = navigator.appVersion;
  nvgtr_dict.PX65 = navigator.appName
  nvgtr_dict.PX66 = navigator.appCodeName
  nvgtr_dict.PX67 = navigator.buildID
  nvgtr_dict.PX51 = navigator.plugins,
  nvgtr_dict.PX60 = "onLine" in navigator &&
       !0 === navigator.onLine,
  nvgtr_dict.PX68 = "cookieEnabled" in
      navigator && !0 ===
      navigator.cookieEnabled   }

function getMimeType() {
  try {
    var t = navigator.mimeTypes &&
        navigator.mimeTypes.toString();
    return "[object MimeTypeArray]" === t ||
         /MSMimeTypesCollection/i.test(t) }
  catch (t) { return !1}    }
```

Script 3. Simplified version of a script that reads several of the `Navigator` API properties to conduct fingerprinting.

```
function NetworkConnection(i) {
 function connectionObject(t, i, r) {
  // Returns the NetworkInformation object
  // that contains information about the
  // network connection of a device.
  return navigator.connection ||
      navigator.mozConnection ||
      navigator.webkitConnection
 }

 // Return network properties.
 return t(a, i.Events), r(a, [{
  value: function () {
  this._dataQueue.addToQueue(
  ...
  timestamp: this.getEventTimestamp(),
  connectionType: this._connection.type ?
  this._connection.type : "",
  efectivType:this._connection.effectiveType
  ? this._connection.effectiveType : "",
  downlinkMax: this._connection.downlinkMax
  ? this._connection.downlinkMax.toString()
  : "",downlink: this._connection.downlink ?
  this._connection.downlink.toString() : "",
  rtt: this._connection.rtt ?
  this._connection.rtt.toString() : "",
  ...
  )}
  ...
}
```

Script 4. Simplified version of a script that uses several properties of the `Network Information` API to conduct fingerprinting.

```
// Other tracking functionality.
...
this.monitorEmailHashes = function() {...},
this.doCookieMatching = function() {...},


// Collection of latitude and longitude.
this.requestGeo = function() {
var e = this;
navigator.geolocation.getCurrentPosition(
  function(t) {
  e.bountyAppend("lat",t.coords.latitude),
  e.bountyAppend("lng",t.coords.longitude),
  e.bountyAppend("acc",t.coords.accuracy)
}, function(t) {
  e.error("Could not lookup Geo Location")
}, {
    enableHighAccuracy: !0,
    timeout: 1500,
    maximumAge: Infinity
})
},

// Collection of other fingerprinting
    information.
this.collectBrowserInfo = function() {...}
...
```

Script 5. Simplified version of a script that collects email hashes, does cookie matching and uses `Geolocation` API to conduct fingerprinting.

```
...
// Capturing clipboard text & current time.
u._sendToQueue = function(e, t) {
  var n = u.getEventTimestamp(e),
  o = e.clipboardData ?
  e.clipboardData.getData("text") :
  window.clipboardData ?
  window.clipboardData.getData("text"):"";

  var s = u.getExports().
  EnumDef.Events.clipboardEventType[e.type];

  u._dataQueue.addToQueue("clipboard_event",
  {timestamp: n, copiedText: o,
  clipboardEventType: s})
  ...
 }
 ...
```

Script 6. Simplified version of script that uses the `Clipboard` API to conduct ephemeral fingerprinting.

```
...
// Reading the timing of webpage paint
    events.
{
key: "onWindowLoad",
value: function() {
  y.a.preloadAll();
  e = performance.getEntriesByType("paint");

  e.forEach(function(e) {
    console.log("".concat(e.name, ": ").
        concat(e.startTime))
  })
}
...
```

Script 7. Simplified version of script that uses the `Performance` API to conduct fingerprinting.

```
...
onLoad = function() {
frame = document.createElement('iframe');
flags = [];
...
if (isPresent(navigator, 'getGamepads')) {
    flags.push('gamepads');
}
...
flags = flags.join(',');
frame.src = ("http://" + host + "/statframe.html#") + flags;
frame.style.cssText = 'display: none;';
return document.body.appendChild(frame);
...
};
```

Script 8. Simplified version of a script that probes the `GamePad` API to conduct fingerprinting.

```
...
function mn(t) {
g("PX847");
var n = p();
if (va) {
  var e = pa[si];
  ua = si, la = n;
  var r = t.deltaY || t.wheelDelta
  || t.detail;

  if (r = +r.toFixed(2), null === e) {
    fa++;
    var o = wn(t, !1);
    o.PX172 = [r], o.PX173 = gt(n)
    , pa[si] = o
  }
else ma.mousewheel <= pa[si].PX172.length
? (Xn(), va = !1) : pa[si].PX172.push(r)}

X("PX847")
}

function gn() {
 if (g("PX847"), pa.mousemove) {
  t = pa.mousemove.coordination_start.length
  , n = pa.mousemove.
  coordination_start[t-1].PX70,
  e = Sn(Tn(_t(pa.mousemove.
  coordination_start))),
  r = Tn(_t(pa.mousemove.coordination_end));
  r.length > 0 && (r[0].PX70 -= n);
  var o = Sn(r);

   pa.mousemove.PX172 = "" !== o
   ? e + "|" + o : e,
   delete pa.mousemove.coordination_start,
   delete pa.mousemove.coordination_end,
   yn(pa.mousemove, "mousemove"),
   pa.mousemove = null
 }
 X("PX847")
}
...
```

Script 9. Simplified version of a script that uses the `Mouse` API to conduct fingerprinting.

```
...
vn.DataMappingDefs = {
// This script define 23 variable with the name of methods/properties related to each API. Then starts collecting information for each API including Sensor API.
...
AMBIENT_LIGHT_EVENT_MAP: ["eventSequence", "timestamp", "illuminance"],
ACCELEROMETER_EVENT_MAP: ["eventSequence", "timestamp", "x", "y", "z"],
GYRO_EVENT_MAP: ["eventSequence", "timestamp", "absolute", "alpha", "beta", "gamma"],
ORIENTATION_EVENT_MAP: ["eventSequence", "timestamp", "absolute", "alpha", "beta", "gamma"],
...
```

Script 10. Simplified version of a script that uses `Sensor` APIs to conduct fingerprinting.