DeepStalker: Improving Tracking Fingerprint Evolution

William Zhang, Eryk Pecyna, Barron Wei

Abstract

Browser fingerprinting is a technique used to track users without storing user states. By using the unique combinations of attributes freely handed over by browsers, websites can identify users. However, because browser fingerprints change over time, identifying the correct user often has some nonzero error rate. One technique recently used in [24] to identify browser fingerprints is a hybrid rule-based and machine learning-based approach. We build off of the work in [24] by incorporating deep learning techniques in their hybrid approach as well as improving their training scheme. We find that our approach results in a 3× average speedup in execution time and better precision than state-of-the-art methods for browser fingerprint tracking.

ACM Reference Format:
William Zhang, Eryk Pecyna, Barron Wei. 2020. DeepStalker: Improving Tracking Fingerprint Evolution. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

A website might want to track its users for various reasons, including targeted advertising, content personalization, and security [1]. The most common method of tracking users involves assigning unique identifiers in cookies. However, recent legislation and discussions of privacy concerns have raised awareness of cookies, making users more likely to regularly delete their cookies to prevent themselves from being tracked. A study by Microsoft [26] found that they were unable to keep track of 32% of their users through cookies only since the cookies were regularly deleted. Browser fingerprinting was proposed in [16] as a technique for tracking users without needing to store per-user states, like cookies. Since then, additional studies have been conducted on browser fingerprint uniqueness . These techniques include capturing OS and hardware-level features [5], fonts and font-metrics [7], and features from HTML5 header and canvas information [14, 17].

Fingerprint uniqueness is insufficient for tracking users because fingerprints change over time. Several recent defenses to exploit fingerprint uniqueness rely on adding randomness [11, 13, 23], so another recent work [24] focuses on linking similar fingerprints to track the same user's fingerprint over time. A web server with a large number of users that is also interested in keeping track of its users in real-time would be interested in faster methods of linking fingerprints.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Conference'17, July 2017, Washington, DC, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 Background & Motivations

Browser fingerprinting takes advantage of standard web APIs, HTTP request headers, and the local JavaScript environment to read information about the browser and device of the user, generating a unique identifier, or fingerprint, from this set of information hashed without any personal information on the user. When a website that uses fingerprinting loads, JavaScript is executed in the browser of the user to access the web APIs, request headers, and JavaScript environment and generate the fingerprint on the fly without storing any information locally. Commonly collected information include the user agent, content language, time zone, fonts, and plugins [12]. While these pieces of information on their own are not uniquely identifying, when combined, all of this information allows a website to identify a unique user with high accuracy [6].

2.1 Fingerprinting Usage

While browser cookies have been a common method to uniquely identify and track users, the usage of cookies to track users has been controversial in terms of privacy as well as insecure in terms of implementation. Restrictions on the usage of cookies to collect information such as those instituted by the General Data Protection Regulation [8] prevent cookies from being a reliable source of tracking as continual consent is required. Additionally, browser cookies are vulnerable to both memory-read attacks and cross-site scripting attacks [3] because browser cookies are stateful, storing the private data locally on disk. Moreover, because cookies are stateful, cookies can be easily removed by users themselves.

The disadvantages of using cookies to track users as described above make way for using browser fingerprinting instead. While fingerprinting in the same fashion as cookies could violate user privacy, explicit consent from the user is not necessary. Government limitations on fingerprinting

to the best of our knowledge do not exist. Moreover, fingerprinting is not vulnerable to memory-read attacks and cross-site scripting attacks on the user end as local state is not stored. Furthermore, users are unable to simply delete the fingerprint, since the fingerprint, if stored, is stored on the server of the website.

While clearly advantageous for targeted advertising and content personalization [1], user tracking, especially in the form of fingerprinting, is also critical from a security standpoint. Browser fingerprinting as well as other forms of user tracking could detect potentially unauthenticated access. By uniquely identifying a user, a website is able to prevent an attacker from authenticating as that user [12]. Unlike cookies, a user fingerprint cannot be stolen online as mentioned above, providing additional security. Moreover, fingerprinting can be used in conjunction with cookies by indicating whether deleted cookies should be replenished [12]. Furthermore, fingerprinting can also be used to detect an insecure configuration of the browser as well as other vulnerabilities of the local device [12]. Browser fingerprinting offers strong advantages in terms of security.

2.2 Fingerprinting Defenses

Despite the above advantages, browser fingerprinting could be viewed as a violation of privacy, leading to attempts to defend against fingerprinting. While browser fingerprinting cannot be mitigated in the same fashion as cookies, by simple removal, browsers can modify the attributes of the collected information [12] in order to convince the website that the user is a different one. Any minor change in one of the attributes would generate a completely new fingerprint. For an example, the user agent string contains the version of the browser, which changes on a browser update, resulting in a different fingerprint. This renders fingerprinting less effective because a website would no longer be able to easily connect past activity to any future activity of that user by simply matching fingerprints. While this example is an unintentional case of defending against fingerprinting, a user could intentionally alter, or spoof, browser attributes with either deterministic or random ones [12]. As described in [12], methods to change these attributes such as installing extensions and plugins could, however, lead to generating a more distinctive fingerprint, improving instead of mitigating fingerprinting. Moreover, modified attributes could also be detected through inconsistencies and reverted as described in [24]. As a result, spoofing attributes or introducing noise can be counteracted if inconsistent, which is often the case due to the large and ever-growing number of attributes and web APIs. In order to stay consistent, a browser extension would have to modify most if not all attributes. This would be computationally expensive and slow, detracting from the user experience, which is important in web browsing.

Although some browsers like TOR implement effective defenses against fingerprinting, we ignore users employing these strategies for this study. As brought up by Vastel, users that employ these techniques are a small subset of all users and often have to degrade their browsing experience to make this strategy effective [24]. The results from our study are still applicable and useful to the broader population that is not tech savvy enough to counteract fingerprinting but may know to clear their cookies.

One recent work in fingerprinting defense comes from Mozilla and proposes a more reasonable approach that does not alter the browsing experience of the user. They proposed a semi-supervised learning approach that uses patterns in API accesses exhibited by fingerprinting scripts to detect domains attempting to track users [4]. Upon detecting a fingerprinting script, the browser can block the script. This is a promising new approach although the paper does not address or discuss performance or the computational burden on the user.

2.3 Recent Works

The first known work to study the stability of browser fingerprints was [6]. The authors in [6] found that 37.4% of the cookie-accepting users in their study changed browser fingerprints and proposed a simple algorithm for tracking fingerprint evolution. They observed that the average browser carries at least 18 bits of entropy and demonstrated that 94% of the browser fingerprints they collected were unique. They further demonstrated that even a simple heuristic is able to to identify when a fingerprint is an upgraded version of a previous one. This work provided ample evidence that tracking a browser through time by using its fingerprint is viable.

Prior work has also been done for applying deep learning to the task of website fingerprint identification. [20] used convolutional neural networks to identify the websites visited by users actively trying to anonymize their traffic using TOR. The network was trained on network traffic and was able to identify websites with an accuracy of 0.99 which only dropped to 0.96 when faced with state of the art traffic analysis counter-measures. This papers goal was to identify websites, not users, but it nonetheless demonstrates the efficacy of using neural networks to identify some entity based on a variable feature set.

[2] used an LSTM recurrent neural network model for user fingerprinting through motion sensors, identifying users based on behavioural patterns while interacting with their smartphones. Their approach was to use feature embedding on the behavioral patterns of users as measured by the motion sensors in their phones to map to a lower-dimensional feature space. They then used a recurrent neural network to identify users with at least 64.01% accuracy. While their approach is interesting and yielded surprising results, the information underlying the fingerprint varies too greatly from

Algorithm 1: Hybrid matching algorithm

```
Function FingerprintMatch(F, f_u, λ, diff):
    rules = {rule1, rule2, rule3};
    F_ksub ← ∅;
    for f_k ∈ F do
        if VerifyRules(f_k, f_u, rules) then
            if ExactMatch(f_u, f_k) then
                return f_K.id
            else
                F_ksub ← F_ksub ∪ ⟨f_k⟩;
            end
        end
    end
    candidates ← ∅;
    for f_k ∈ F_ksub do
        ⟨x_1, x_2, ..., x_M⟩ =FeatureVector(f_u, f_k);
        p ←ML.Prediction(⟨x_1, x_2, ..., x_M⟩);
        if p ≥ λ then
            candidates ← candidates ∪ ⟨f_k, p⟩;
        end
    end
    if |candidates| > 0 then
        c_{h_1}, p_{h_1} ←GetByRank(candidates, 1);
        c_{h_2}, p_{h_2} ←GetByRank(candidates, 2);
        if p_{h_1} ≥ λ then
            if c_{h_1}.id == c_{h_2}.id or p_{h_1} > p_{h_2} + diff then
                return c_{h_1}.id
            end
        end
    end
    return GenerateNewID()
```

`GetByRank(list, i)` is a function that returns the entry *list* of rank *i*, based on the probabilities

browser features to be a useful comparison. It nonetheless demonstrates the power of deep neural networks for classifying users based on fingerprints whose changes through time are not clearly defined.

[10] proposed an algorithm for tracking using various heuristics and rules and managed to attain an accuracy of about 0.9 for recognizing if two fingerprints are from the same browser instance. Their approach, however, is very expensive computationally and does not yield an accuracy high enough for effective long term tracking.

Our work follows the work in [24], in which they propose a procedure to link fingerprints and build a chain of fingerprints that correspond to a single user.

Our work is the first use of deep learning to track fingerprint evolution. The approach of [24] can link fingerprints in an online setting: whenever a new unknown fingerprint f_u arrives, the server compares it against a list of the fingerprints F that it already has. First, they use a rule-based algorithm to filter out fingerprints that could not belong to the same user. If there is a match or if the fingerprints are sufficiently close (as measured by a machine learning-based approach), then the server will mark the unknown fingerprint as belonging to one of the known fingerprints f in its database. Specifically, for every pair of fingerprints f_u and f_k, they extract a feature vector $\langle x_1, x_2, \ldots, x_M \rangle$ by comparing each attribute of the fingerprint (e.g. Javascript attributes, time zone, Flash attributes, etc.) and extracting either an indicator variable (if the attributes are the same) or a similarity metric (e.g. Levenshtein distance between string values). If there is no match, then a new ID is generated for the unknown fingerprint. The `ML.Prediction` function is trained by observing feature vectors of positive links (which come from pairs of fingerprints from the same user) and feature vectors of negative links (pairs of fingerprints from different, random users). This procedure is described more precisely in Algorithm 1, where `FingerprintMatch` returns the user that it predicts the unknown fingerprint to be associated with. It takes in additional parameters λ and $diff$ that determine how confident the `ML.Prediction` function must be in order to label the unknown fingerprint as either a match or not. Both of these are tunable according to the training of the prediction model. Because of the extreme unlikelihood of users with identical fingerprints, it is reasonable to assume that any two matching fingerprints must correspond to the same user.

The hybrid linking algorithm developed in [24] uses three rules because they are constraints that are very unlikely to be violated between two fingerprints from the same browser. The rules are defined as follows:

1. The OS, platform, and browser family must be identical for any given browser instance. Even if this may not always be true (e.g. when a user updates from Windows 8 to 10), [24] considers it reasonable for their algorithm to lose track of a browser when such a large change occurs since it is not frequent.
2. The browser version remains constant or increases over time. This would not be true in the case of a downgrade, but this is also not a common event.
3. Local storage, Dnt, cookies, and canvas should be constant for any given browser instance. These attributes do not change often, if at all, for a given browser instance. In the case of canvas, even if it seldomly changes for most users, the changes are unpredictable making them hard to model. Since canvas are quite unique among browser instances, and don't change too frequently, it is still interesting to consider that it must remain identical between two fingerprints of the same browser instance.

In [24], they collected 172,285 fingerprints from 7,965 different browser instances. They filter out all fingerprints associated with browser instances with less than 7 fingerprints in

the raw data as well as fingerprints from which they detect countermeasures to artificially modify them, giving a dataset of 98,598 fingerprints from 1,905 browser instances. The participants in their study installed the AmIUnique extensions for Chrome and Firefox between July 2015 and early August 2017, the duration for this data collection procedure. We use the subset of the 15,000 finger that were publically released for the study—the authors were unable to release the full dataset for legal reasons.

3 Linking using a Neural Network

3.1 Methods

We improve the results from [24] by using better machine learning models and training methods. For training and evaluation, [24] uses replay sequences. They use a function `GenerateReplaySequence`, which takes as inputs F and $visitFrequency$, and returns a subset F' of F such any two consecutive fingerprint instances in F' belonging to the same user are at least $visitfrequency$ days apart. This mimics the more realistic scenario where a user only visits the website infrequently rather than the web server being able to record user browser fingerprints on a regular basis, as is done by the browser plugin data collection for the study. Including different sample frequencies between consecutive fingerprints during training allows the model to learn positive similarity features vectors across varying time spans, which is helpful because there is no guarantee that a user would visit a given site on a regular basis. Just as in [24], we use a 40/60 train/test split, where we split the data chronologically to ensure that no data leakage due to the temporal nature of the dataset occurs. The training procedure of [24] is described further in Algorithm 2.

We provide three modifications to their algorithms. First, we oversample from the negative training set by a factor of k, illustrated in Algorithm 3. In our implementation, we chose $k = 20$. This results in better performance at test time for two reasons: (1) during test time, the server should encounter much more negatives than positives, so training on data more similar to the testing data should give better results, and (2) increasing the amount of training data improves model generalization. Second, we replace the random forest model for [24] with a neural network model.

The architecture of our layers is as shown in Fig. 1. Every dense layer has batch normalization [9] applied to its output as well as a dropout [21] of 0.5. Batch normalization is a technique used to increase the stability of neural networks. During training, this method re-centers and re-scales the output of the layer for the current batch of training data using a moving mean μ and standard deviation σ. These learned parameters are then used for normalization during inference. Dropout on the other hand randomly chooses some predefined percentage of nodes in the layer to deactivate at each stage and uses the remaining activated nodes for that stage. This method is a very powerful technique to prevent over-fitting and help the model generalize better.

We tried various architectures and tuned them with different parameters until eventually settling on this architecture.

Algorithm 2: Training data generation

```
Function GenerateTrainData(F, maxFrequency):
    X ← [];
    y ← [];
    for i ∈ [1, ..., maxFrequency] do
        replay ← GenerateReplaySequence(F, i);
        for f ∈ replay do
            fPos ←GetNextFP(replay, f);
            featureP ←FeatureVector(f, fPos);
            X.append(featureP);
            y.append(1);
            fNeg ←GetRandomExcept(F, f);
            featureN ←FeatureVector(f, fNeg);
            X.append(featureN);
            y.append(0);
        end
    end
    return X, y
```

`GetNextFP(replay, f)` is a function that returns the next fingerprint in $replay$ after f that is attached to the same user. `GetRandomExcept(F, f)` is a function that picks a random fingerprint in F that is not associated with the same user corresponding to f.

Algorithm 3: Improved Training data generation

```
Function GenerateTrainDataBetter(F,
  maxFrequency, k):
    X ← [];
    y ← [];
    for i ∈ [1, ..., maxFrequency] do
        replay ← GenerateReplaySequence(F, i);
        for f ∈ replay do
            fPos ←GetNextFP(replay, f);
            featureP ←FeatureVector(f, fPos);
            X.append(featureP);
            y.append(1);
            repeat k times
                fNeg ←GetRandomExcept(F, f);
                featureN ←FeatureVector(f,
                  fNeg);
                X.append(featureN);
                y.append(0);
            end
        end
    end
    return X, y
```

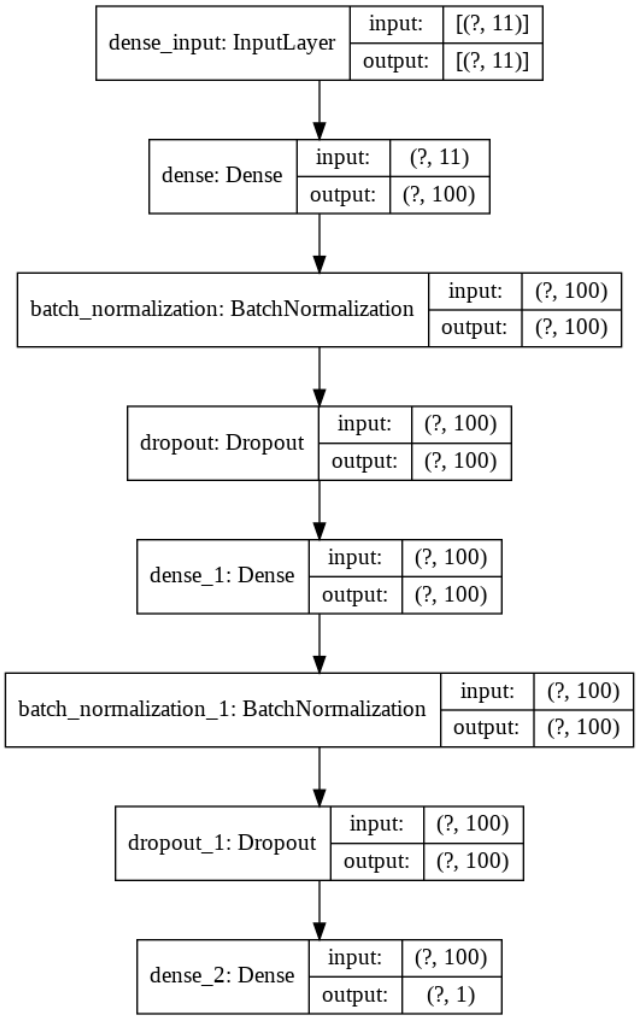

Figure 1. Neural Network Architecture
?s indicate arbitrary batch size

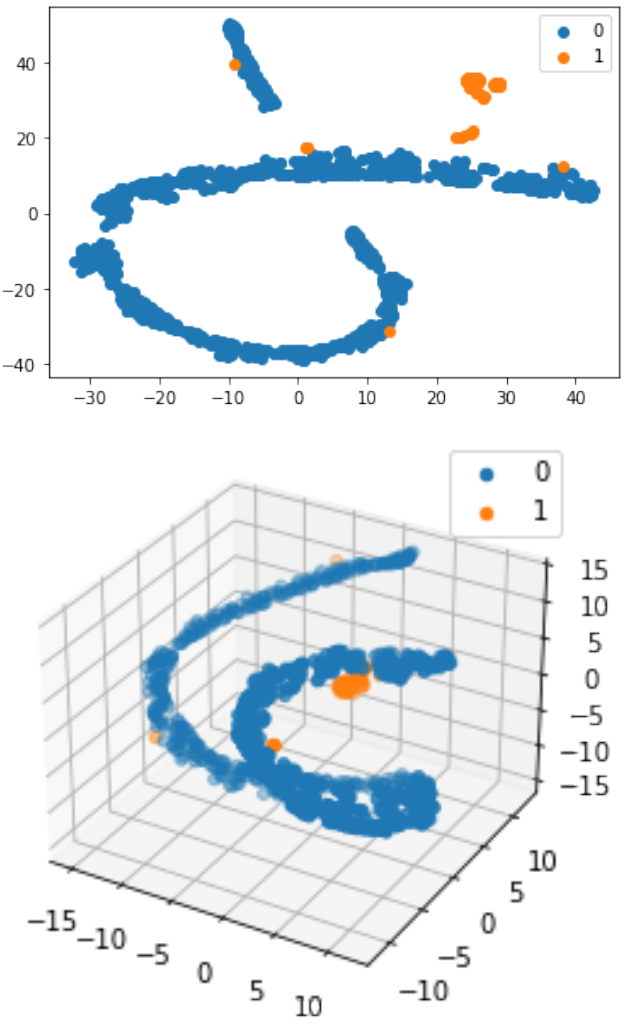

Figure 2. t-SNE plots of a random sample of 1000 training points
t-SNE is nonlinear dimensionality reduction tool for visualizing high-dimensional data.

Smaller architectures would trade blows with random forest but this one manages to outperform the random forest used in [24] on all metrics, including time. The random forest model proposed by [24] trains much faster than our neural network, but takes much more time on inference. This is a great trade-off as training only has to occur once but a server trying to employ fingerprinting will constantly be running inference so our speed advantage in inference is quite significant. Third, we propose a multi-round training scheme to obtain better classification boundaries. In this scheme, we train the neural network multiple rounds, but in each round we bias the network to be trained on the data in which it has the highest loss (as measured using cross-entropy). The main idea behind this training scheme is that we want our training data to include the points closest to the true decision boundary to ensure a more precise boundary. We use this strategy of *hard example learning* [19] because we expect that most randomly sampled feature vectors with the negative label are going to be far from the true decision boundary, i.e. easy examples, and including hard examples in training data is known to speed up convergence. We provide a low-dimensional visualization of the data to justify this, where we observe that most of the positive examples are concentrated in a small region, with a few outliers lying along the manifold of negative examples. This is expected because positive feature vectors are computed by taking near-identical fingerprints whereas in negative feature vectors, there are many different dimensions in which two fingerprints can differ and thus the data manifold of the negative examples would have a higher dimension. The outliers in the positive example correspond to pairs of browsing instances from the

same user in which many different attributes of the fingerprint changed between the pair. In these cases, it makes more sense for the algorithm to return false negatives and generate a new ID because these are difficult to link.

3.2 Evaluation

We improve the results in [24] on all benchmarks. Our baseline to compare against is their random forest model, using their hyperparameters: *number of trees* set to 10, *number of features* set to 3, *diff* set to 0.10, and λ set to 0.994. However, note that the results will not be exactly the same as in the graph of [24] since we only get access to a subset of their data, our subset being only 15,000 fingerprints rather than the 172,285 fingerprints that they were able to collect. After filtering out browsing instances with less than 7 fingerprints as well as fingerprints from which we detect fingerprinting countermeasures, we obtain a dataset of 4,804 fingerprints. Although this is substantially less than [24], it is enough to reach statistically valid conclusions, and we train the random forest with their hyperparameters on this data to ensure a fair comparison.

3.3 Comparison

Because we were able to adjust λ to trade off performance in the first three metrics in Fig. 3 with average ownership, our goal was to beat the random forest model in all metrics. For smaller models we received fair results and were able to beat the random forest for certain subsets of the tested collect frequencies, but our final model was able to achieve our goal.

We used the same metrics defined in [24] to get a fair comparison between our models. Tracking length is measured by the length of a chain generated for a particular browser instance. So average tracking duration is just the mean length of all the tracking chains a browser instance is present in. If we had an ideal model, each browser instance would only have one chain, but due to error there is some cross pollution of chains as well as splitting. The average maximum tracking duration is then defined as the mean length of the longest chain each browser is present in. Number of ids per user is simply how many different unique ids get assigned to the fingerprints generated by a specific browser instance. Finally an owner is the browser instance that has the most fingerprints in a chain. Therefore average ownership is the average percentage of owners fingerprints per chain.

On average our model is able to achieve an extra 10-20 more days of tracking, depending on the collect frequency, and does so while maintaining higher ownership across its tracking chains. In fact, for a subset of collect frequencies, our model was able to achieve perfect ownership. This means that each chain only contained fingerprints from one browser, although a browser may have been associated with more than one chain.

We also found that for small trade-offs in average ownership, we were able to significantly increase tracking duration. We also show in Fig. 3 how our model performed with a lower lambda, meaning we decreased average ownership to increase tracking duration. This kind of trade off would be perfect for an advertising company who might not care if a users information is a little polluted if it means they can track users for far longer.

When comparing execution speeds, we do inference with trained models on the same machine. Note that it is possible to speed up the neural network inference by several orders of magnitude by using GPU or other hardware that performs deep learning efficiently (e.g. TPU) [15, 25], but we do not include this in our evaluation because we lack the hardware. Something else to note is that if a server is receiving fingerprints from multiple users at once, it can batch those fingerprints together to increase inference speed even further. This does require the assumption that no two fingerprints in the batch come from the same browser, but this is a valid assumption to make for fingerprints collected from unique users or fingerprints collected in a small time window.

Because the random forest does not have as much potential for parallelization as a neural network does, we tested both models under the same setup we tested their tracking performance in. The fingerprints were getting linked one at a time so we timed how long it took to link the fingerprint using a specific model. Note that this includes the entire fingerprint linking function which does not necessarily always call the model, for example if two fingerprints are identical, or if they violate one of the three rules referenced in Algorithm 1, the model does not have to be called. This results in similar lower percentiles but the maximum and average times shown in Fig. 4 still demonstrate our neural network is about 3 times faster than the random forest. Because by default tensorflow will use all the cores available to it, we also set the n_jobs parameter on the random forest to -1 which allows it to use all the cores available to it. All tests were run on the same machine.

3.4 Upsampling Training Data

We did not find much success in biasing the training of our model towards fingerprint pairs close to the decision boundary. Although this method did not yield an improvement in the performance of our model, our upscaling of the negative training data did. Upsampling the amount of negative examples we trained did not yield much improvement in tracking duration but yielded massive improvements in ownership ratio. Before we tried this new approach, our models were beating the random forest in tracking duration but couldn't quite surpass it in ownership ratio, then after implementing this idea, we were able to finally achieve complete domination of the random forest performance.

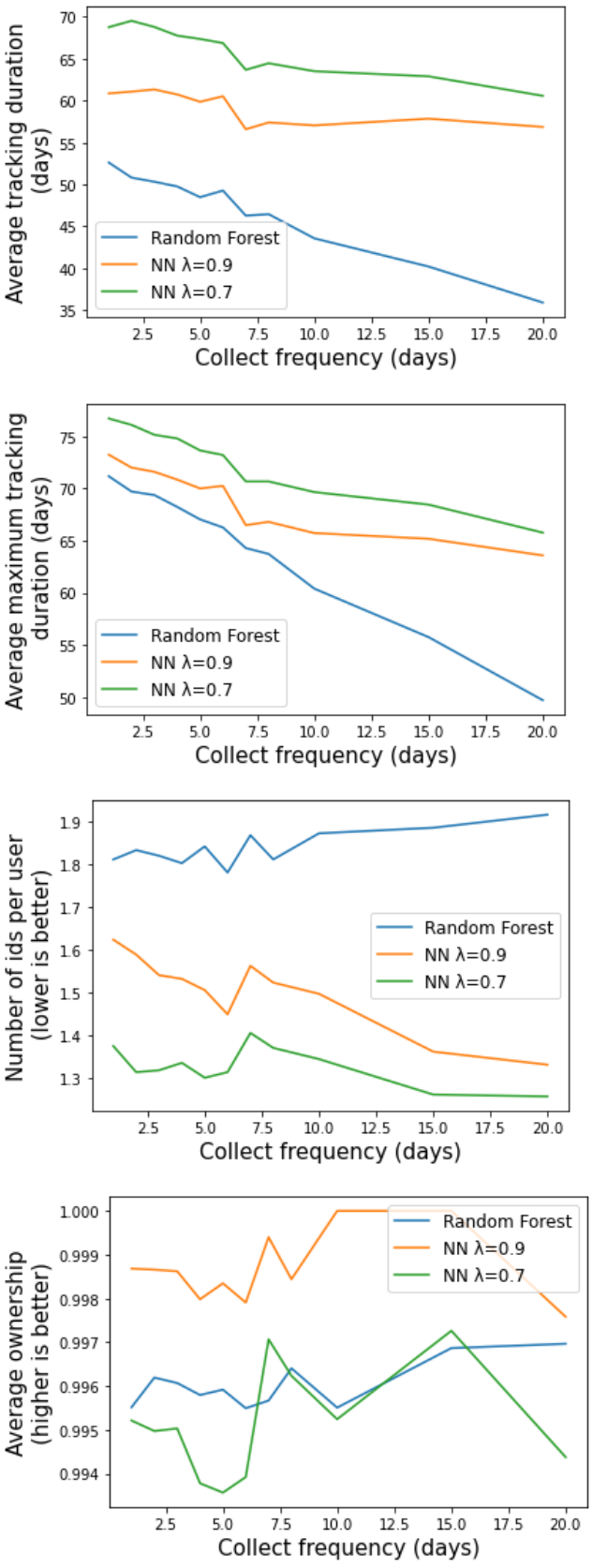

Figure 3. Comparisons

Link time (s)	Random Forest	Neural Network
Avg. time	0.026360	0.008014
Min. time	0.000000	0.000000
25th percentile	0.000000	0.000000
50th percentile	0.000991	0.000999
75th percentile	0.001001	0.001001
Max. time	0.448101	0.330074

Figure 4. Speed of linking using each model

4 Conclusion

We proposed applying deep learning to the task of tracking users in settings in which their fingerprint evolves over time. The advantages of this approach to previous state-of-the-art results are numerous. First, we improve the execution speed of inference, which allows a server interested in tracking fingerprints to handle a much larger load of users. A server could also use specialized hardware to accelerate the deep learning inference. Second, deep learning is extremely suitable for batch inference, unlike random forests. This would allow a server to track an even larger number of users in real-time since a user would not appear multiple times in the same batch (i.e. we do not expect a user to be visiting a site multiple times within a span of microseconds). Third, the performance of neural network models scale extremely well with larger datasets [18, 22]. Given that we already outperform the random forest model from [24] using only a small subset of the original data, it is expected that if we could get access to the full dataset that we would see even greater performance improvements.

References

[1] Gunes Acar, Marc Juarez, Nick Nikiforakis, Claudia Diaz, Seda Gürses, Frank Piessens, and Bart Preneel. 2013. FPDetective: dusting the web for fingerprinters. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 1129–1140.

[2] Sara Amini, Vahid Noroozi, Sara Bahaadini, S Yu Philip, and Chris Kanich. 2018. Deepfp: A deep learning framework for user fingerprinting via mobile motion sensors. In *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 84–91.

[3] AB Bhavani. 2013. Cross-site scripting attacks on android webview. *arXiv preprint arXiv:1304.7451* (2013).

[4] Sarah Bird, Vikas Mishra, Steven Englehardt, Rob Willoughby, David Zeber, Walter Rudametkin, and Martin Lopatka. 2020. Actions speak louder than words: Semi-supervised learning for browser fingerprinting detection. *arXiv preprint arXiv:2003.04463* (2020).

[5] Yinzhi Cao, Song Li, Erik Wijmans, et al. 2017. (Cross-) Browser Fingerprinting via OS and Hardware Level Features.. In *NDSS*.

[6] Peter Eckersley. 2010. How unique is your web browser?. In *International Symposium on Privacy Enhancing Technologies Symposium*. Springer, 1–18.

[7] David Fifield and Serge Egelman. 2015. Fingerprinting web users through font metrics. In *International Conference on Financial Cryptography and Data Security*. Springer, 107–124.

[8] Xuehui Hu and Nishanth Sastry. 2019. Characterising Third Party Cookie Usage in the EU after GDPR. In *Proceedings of the 10th ACM Conference on Web Science*. 137–141.

[9] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167* (2015).

[10] Wei Jiang, Xiaoxi Wang, Xinfang Song, Qixu Liu, and Xiaofeng Liu. 2020. Tracking your browser with high-performance browser fingerprint recognition model. *China Communications* 17, 3 (2020), 168–175.

[11] Pierre Laperdrix, Benoit Baudry, and Vikas Mishra. 2017. FPRandom: Randomizing core browser objects to break advanced device fingerprinting techniques. In *International Symposium on Engineering Secure Software and Systems*. Springer, 97–114.

[12] Pierre Laperdrix, Nataliia Bielova, Benoit Baudry, and Gildas Avoine. 2020. Browser fingerprinting: a survey. *ACM Transactions on the Web (TWEB)* 14, 2 (2020), 1–33.

[13] Pierre Laperdrix, Walter Rudametkin, and Benoit Baudry. 2015. Mitigating browser fingerprint tracking: multi-level reconfiguration and diversification. In *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE, 98–108.

[14] Pierre Laperdrix, Walter Rudametkin, and Benoit Baudry. 2016. Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 878–894.

[15] Eric Lind and Ävelin Pantigoso Velasquez. 2019. A performance comparison between CPU and GPU in TensorFlow.

[16] Jonathan R Mayer and John C Mitchell. 2012. Third-party web tracking: Policy and technology. In *2012 IEEE Symposium on Security and Privacy*. IEEE, 413–427.

[17] Keaton Mowery and Hovav Shacham. 2012. Pixel perfect: Fingerprinting canvas in HTML5. *Proceedings of W2SP* (2012), 1–12.

[18] Preetum Nakkiran, Gal Kaplun, Yamini Bansal, Tristan Yang, Boaz Barak, and Ilya Sutskever. 2019. Deep double descent: Where bigger models and more data hurt. *arXiv preprint arXiv:1912.02292* (2019).

[19] Abhinav Shrivastava, Abhinav Gupta, and Ross Girshick. 2016. Training region-based object detectors with online hard example mining. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 761–769.

[20] Payap Sirinam, Mohsen Imani, Marc Juarez, and Matthew Wright. 2018. Deep fingerprinting: Undermining website fingerprinting defenses with deep learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1928–1943.

[21] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research* 15, 1 (2014), 1929–1958.

[22] Noé Sturm, Andreas Mayr, Thanh Le Van, Vladimir Chupakhin, Hugo Ceulemans, Joerg Wegner, Jose-Felipe Golib-Dzib, Nina Jeliazkova, Yves Vandriessche, Stanislav Böhm, et al. 2020. Industry-scale application and evaluation of deep learning for drug target prediction. *Journal of Cheminformatics* 12 (2020), 1–13.

[23] Marthony Taguinod, Adam Doupé, Ziming Zhao, and Gail-Joon Ahn. 2015. Toward a moving target defense for web applications. In *2015 IEEE International Conference on Information Reuse and Integration*. IEEE, 510–517.

[24] Antoine Vastel, Pierre Laperdrix, Walter Rudametkin, and Romain Rouvoy. 2018. FP-STALKER: Tracking browser fingerprint evolutions. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 728–741.

[25] Yu Emma Wang, Gu-Yeon Wei, and David Brooks. 2019. Benchmarking TPU, GPU, and CPU platforms for deep learning. *arXiv preprint arXiv:1907.10701* (2019).

[26] Ting-Fang Yen, Yinglian Xie, Fang Yu, Roger (Peng) Yu, and Martin Abadi. 2012. Host Fingerprinting and Tracking on the Web:Privacy and Security Implications. In *The 19th Annual Network and Distributed System Security Symposium (NDSS) 2012* (the 19th annual network and distributed system security symposium (ndss) 2012 ed.). Internet Society. https://www.microsoft.com/en-us/research/publication/host-fingerprinting-and-tracking-on-the-webprivacy-and-security-implications/