CS-523 SecretStroll Report

Pierre GABIOUD, Justinas SUKAITIS

***Abstract*—The second project of the CS-523 course: SecretStroll is a location-based application, concentrated on preserving the privacy of the users and their personal data. The project is separated into three parts: an anonymous authentication mechanism, a privacy evaluation of users' location data and Cell Fingerprinting.**

I. Introduction

Many current mobile applications such as Yelp, Facebook and Foursquare have a location-based service. These features often provide a great amount of additional usability for the user. Unfortunately, in this day and age, most of these applications do not provide any privacy assurance against location tracking. SecretStroll aims at implementing a location-based app that keeps the users' personal data and location private. The application sends the user nearby Points of Interest (POIs), based on his/her monthly interest-based subscriptions.
The first part of the project implements an Attribute-Based Credentials (ABCs) system, that authenticates new users and provides message signing for the users' queries to prove ownership of his/her active subscriptions while keeping different signatures unlinkable.

II. Attribute-based credential

We implement **ABCs** by following the **PS** scheme [1] described in Sections 6.1 "Multi-Message Protocol", 6.2 on "Proving Knowledge of a Signature" and mapping the scheme to six main functions defined in `your_code.py` as well as several auxiliary functions defined in `credentials.py` as such:

- **Setup** and **Keygen** are called by the server in function `generate_ca`, where he takes the part of the Issuer and with the help of the petrelic library is able to easily compute the secret parameters sk: $(x, y_1, y_2, ...y_n, X)$ and the public parameters pk: $(Y_1, Y_2, ...Y_n, \widetilde{X}, \widetilde{Y_1}, \widetilde{Y_2}, ...\widetilde{Y_n})$ in addition to the cyclic groups, their generators and the list of attributes that are defined at function call. Furthermore, as a common practice, we let the Issuer include the pk into sk for future ease of code.
- When a client wishes to obtain a signature, he initiates the main **Protocol** of the **PS** scheme by calling `prepare_registration`.
  The public parameters (the different subscriptions) are decided on function call while the private attributes $(a_2...a_h) \in H$ are set inside the registration function separately with a secret key a_1. While keeping these values hidden from the server side, the client generates a random value t and computes $C = g^t \prod_{i \in H} Y_i^{a_i}$.

For the server to verify if the credential was computed correctly, we use the Fiat-Shamir heuristic to apply a non-interactive **ZKP** on C.

The client generates a random vector of values $(v_0, v_1, ...v_h) \in \mathcal{Z}_p$ and computes a vector $\gamma = (g^{v_0}, Y_1^{v_1}, ...Y_h^{v_h})$, where $h = |H - 1|$. Furthermore, using the `hashlib` library we generate a cryptographic hash c of the public parameters, C and γ. Finally, the client computes a final vector of values $r = ((v_0 + ct), (v_1 + ca_1), ...(v_h + ca_h))$ and sends (C, γ, r) to the server.
Having received the client's commitment, the server can compute the same hash c from the public parameters and the received γ and C and verify if $\prod_j \gamma_j = C^{-c} g_0^r \prod_{i \in H} Y_i^{r_i}$. If the verification succeeds the Protocol finalizes the signature by following the PS scheme.

Whenever the user does a query to the server, he calls the function `sign_request.py`, which internally handles the message signing and the second non-interactive **ZKP** over his secret attributes:
Using the previously defined σ The client generates two random values $k, t \in \mathcal{Z}_p$ and calculates $\sigma' = (\sigma_1^k, (\sigma_2 \sigma_1^t)^k)$ and computes a new commitment $C' = e(\sigma_1', \widetilde{g}^t) \prod_{i \in H} e(\sigma_1', \widetilde{Y_i})^{a_i}$.
As in the previous section, for the Fiat-Shamir heuristic, the client also generates a new random vector $(v_0, v_1, ...v_h) \in \mathcal{Z}_p$ and computes $\gamma = (e(\sigma_1', \widetilde{g})^{v_0}, e(\sigma_1', \widetilde{Y_1})^{v_1}, ...e(\sigma_1', \widetilde{Y_h})^{v_h})$. Furthermore, using the query message, C', the public attributes and γ, the client computes a new hash c and computes $r = ((v_0 + ct), (v_1 + ca_1), ...(v_h + ca_h))$. The client sends (message, σ', γ, r), where the signature of the message with σ' is held in r, since if either σ' or the message is changed, the value of r would not hold anymore.

The server, having received the clients' (message, σ', γ, r), computes on his own side the hash c and verifies if:

- $\sigma 1 \neq 1_{G1}$
- $\prod_j \gamma_j = left \cdot e(\sigma_1', \widetilde{g})^{r_0} \prod_{i \in H} e(\sigma_1', \widetilde{Y_i})^{r_i}$
  where *left* is:
  $(e(\sigma_2', \widetilde{g}) e(\sigma_1', \widetilde{X})^{-1} \prod_{i \in D} e(\sigma_1', \widetilde{Y_i})^{-a_i})^{-c}$
  In case of success, the server sends the user the nearby **POIs** that are based on his subscriptions.

By using unique private attributes for the credentials, the system requires the user to only reveal the subscriptions he has, thus minimizing the information leakage.

A. Test

We used the library *pytest* to test our system and started by defining some *pytest.fixture* to create common parameters for our tests. It is therefore easy to vary the public attributes or

overall valid attributes and their number to test the robustness of our code with different settings.

Our strategy to perform complete and effective tests is to first check the specific functions in `credentials.py` and then broaden the tests to the functions in `you_code.py` using the previously tested and verified functions. Hence the first test checks the correctness of the commitment made by the client in the `create_issuer_request` method and then the credentials issued by the Issuer with the `receive_issue_response` method.

Once the methods of `credentials.py` tested, we can test the complete registration paths of a client to the server with the methods in `you_code.py` considering the functions in `credentials.py` as correct black boxes. There are two possible paths in this case: one with a correct client's commit for the **ZKP** and one failure path with a wrong client's commitment.

If these tests pass, we can consider the PS credential issuance as a black box and apply the same strategy for the signature verification. Once the commit of the `sign_request` method and the output of the `verify` method checked, we can test the complete signature verification process and thus the complete system by checking the signature functions of `you_code.py`. Again, two distinct paths must be tested: one with a correct commit from a client and one failure path where the commit is incorrect and the verifier detects it.

Note that the results of the tests depend neither on the hardware they are running on nor the testing environment since it is the same as the system environment.

B. Evaluation

We first measured the communication delays in our system on the docker, before evaluating the computation cost locally. We divided the performance evaluation into three cases based on the number of valid attributes where we vary the number of private attributes only since the public attributes never impact the size of the commit from the client (note that we always used "oss117" as username and "Advanced topics on privacy enhancing technologies" as a message):

- - The default 4 private attributes and 2 public attributes "1,1", so 6 valid attributes in total.
- - The default 4 private attributes append to 14 other private attributes equal to to 1 and 0 subsequently, with 2 public attributes "1,1", meaning that there are 20 valid attributes in total.
- - The default 4 private attributes append to 44 other private attributes equal to to 1 and 0 subsequently, with 2 public attributes "1,1", meaning that there are 50 valid attributes in total.

There are two measurements of communication delay: the first is when the client sends its commit to the issuer to get credentials, and the second is when the client sends its signature to the verifier (see Figure 1).

From the results, we see that Part2 has bigger delays. This is because, in this part, we send a signature object containing one more additional G1Element than the part 1 and a string message (of size 44 for the tests), which explains the difference.

Communication time for:	Part 1	Part 2
4 private, 2 public	0.0052 s	0.0098 s
18 private, 2 public	0.0126 s	0.0186 s
48 private, 2 public	0.0168 s	0.0388 s

Fig. 1: Average communication time for the client to send its commit to the issuer (Part 1) and for the client to send its signature to the verifier (Part 2) depending on the number of private and public attributes.

Computation time for:		6 va	20 va	50 va
generate_ca	MIN	0.0077	0.0134	0.0301
	MEAN	0.0090	0.0190	0.0467
	STD	0.0010	0.0038	0.0071
prepare_registration	MIN	0.0053	0.0089	0.0162
	MEAN	0.0061	0.0096	0.0172
	STD	0.0003	0.0003	0.0005
register	MIN	0.0055	0.0093	0.0182
	MEAN	0.0063	0.0103	0.0195
	STD	0.0004	0.0004	0.0007
proceed_registration	MIN	0.0007	0.0007	0.0007
	MEAN	0.0009	0.0008	0.0008
	STD	0.0001	0.0001	0.0001
sign_request	MIN	0.0274	0.0201	0.0241
	MEAN	0.0283	0.0307	0.0331
	STD	0.0006	0.0025	0.0066
check_request	MIN	0.0246	0.0332	0.0697
	MEAN	0.0259	0.0511	0.0879
	STD	0.0008	0.0054	0.0203
Complete process	MIN	0.0725	0.0909	0.1455
	MEAN	0.0752	0.1276	0.1782
	STD	0.0016	0.0055	0.0404

Fig. 2: Computation time evaluation (in second) with "va" being the number of valid attributes, MIN the minimum measurement, MEAN the mean of the measurements and STD the standard deviation in the measurements

We evaluated the computation performance of our system using the same username and message as before for each measurement. Our evaluation strategy is to test the performance of each function in `you_code.py` separately and locally, with a varied number of input public parameters (2, 16 and 46 to have respectively 6, 20 and 50 valid attributes in total), and one performance evaluation for the complete process. We did not vary the number of private attributes because the global performance for computation only depends on the number of valid attributes and not on the private or public attributes specifically.

For the evaluation, we computed the mean and standard deviation over 500 samples (Figure 2). We additionally measure the minimum computation time over the samples, following the advice in the *timeit* official documentation, telling that many processes may interfere with the accuracy of our measurements and the minimum value is often the most representative one.

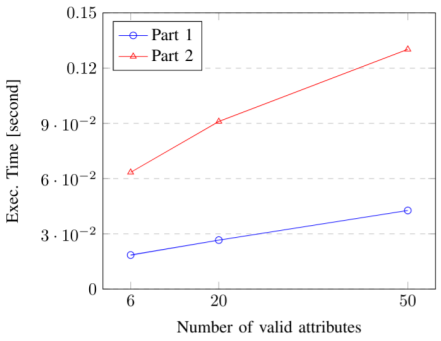

Fig. 3: Plot of mean execution time depending on the number of attributes for the registration (Part1) and the verification (Part2)

In Figure 3, we see the mean time required (communication included) for registration, and the time required to create and verify a signature. We see that the creation of signature and the verification take way more time than the registration. This is probably because we are working on bilinear values and doing pairing operation that are more costly. Moreover, we see that the number of attributes has a greater impact for signature verification, certainly because this affect the size and computation of bilinear value as well.

III. (De)Anonymization of User Trajectories

The application service provider receives locations from users at different times of the day, who wish to know nearby points of interest of a certain type. For this, the service provider holds a database, which contains 12 different types of **POIs** with their corresponding locations. When a user does a query, the server computes a subsection of the city where the user is located (a grid ID) and returns all **POIs** of the specified type in the same grid ID to the user. By combining the query data with the database he holds, he can identify the type of **POI** the user is located in.

A. Privacy Evaluation

1) Adversary and Assumptions: We evaluate the user location privacy against the service provider. Assuming the service provider is an honest but curious party, with no prior knowledge about the users, capable of human reasoning and that can deduce some generalities such as people are at home in the morning and the evening, they only work from Monday till Friday and don't go to random places with zero motive, etc. Amongst the 12 types of **POIs**, we assume users live in types *villa* or *appartment block* and work in types *office*, *laboratory* or *company*. The rest are only recreational types such as *bar, gym* etc.

2) Attack: In order to evaluate the privacy leakage, we do a standard two-step attack to determine where all users live, where they work, and then, by choosing one specific user, we try to learn as much as possible from his locations.

3) Wide-spread Attack: Before doing the attack, we merge the queries with the pois dataset to see what type of building the user was in during his query. We then filter this dataset by keeping only queries that were done from home locations and group by user ips. In less than a second, for all of the users we were able to find their unique home locations: Figure 4.

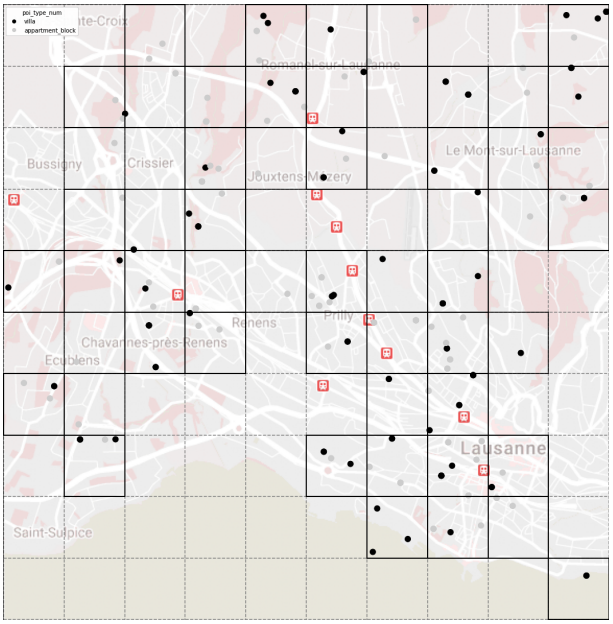

Fig. 4: All the user home locations, color coded to type of home they live in.

We apply the same technique to user work locations and find that 69 work in an office, 55 work in a laboratory and 76 in a company. The attack took 0.61s, but could be optimized to take even less.

4) Selective targeting: For our precision attack, we selected user with $ip\_address = 60.109.165.215$. Using the previous method for this single user, we found that he lives in Renens, near the train station and works in a company in Remanel-sur-Lausanne. By looking at the queries he did, we see that the user always goes out to eat during lunch break and a pattern of searching for clubs while at a restaurant or a bar, which is an often activity called "Bar hopping". Finally, the user often searched for gyms after eating at a restaurant or from home. A query from a gym in Sainte-Croix indicates he finally went to one, but a few weeks later he did a query from another one. We could assume the change was due to the first gym being too far from where he lived.

5) Attack conclusion: We can see that the service provider requires no effort at all to infer multiple types of information, on a scale of 1 to 5 we evaluate the severity of the privacy leakage:

- What the user searched for: severity 1. The queries don't contain searches for sensitive locations such as hospitals, religious buildings, etc. and are recreational. We also note that searching for a location does not imply the user having gone there.
- The server learned the users' home location: severity 5. As mentioned, the home address often has a one-on-one relationship with the user full name and thus his full identity is leaked.

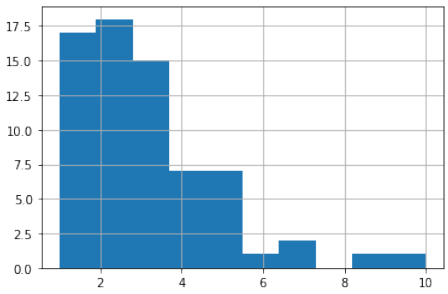

Fig. 5: Number of **POIs** of type *villa* or *appartment_block* per grid.

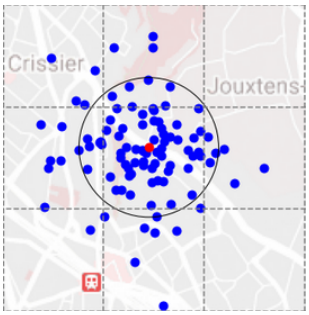

Fig. 6: Example use of the perturbation mechanism for scale = 3.5 with 100 samples. The red dot depicts the real location, while the blue ones depict the provided fake locations.

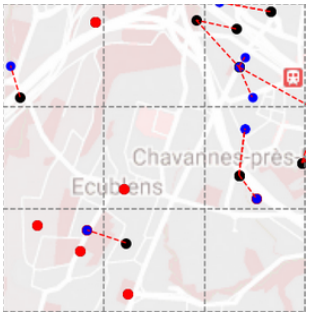

Fig. 7: Locations that the adversary guessed correctly (red), guessed wrongly (black), coupled using a dashed red line with the corresponding true location (blue).

- The server learned the users' work location: severity 4. While not as serious as learning the user's home address, depending on the company, the users' social group can be greatly narrowed and in some cases, his full identity can be leaked.
- The server learned the user was at a random point: severity 2. As stated previously, the other types of locations are not linked to sensitive information, but learning where the user was could reveal his daily patterns and with enough data could correctly predict the users' future locations and activities using Markov chains.

B. Defences

As we have stated, the severity changes depending on the attack. Unfortunately, the cost of these attacks remains constant and small, since all the adversary has to do is merge the databases and filter.

For our defense we concentrate on providing user location privacy by adding noise to the user location in the query, sampled from a Laplace distribution. We decided a Perturbation technique instead of a Hiding one for the following reasons:
1) the severity of home and work location is high, which leads us to wish high privacy.
2) As seen in (Figure 5) several homes are unique in their corresponding grid and the average tends to be between 2 and 3 homes per grid. Thus a hiding technique that would generalize the user location to his grid ID would only work for very few cases and have little to no privacy benefit.

Before the user submits his query to the server, he samples two random variables from a Laplace distribution using scale ϕ and adds the two values to his longitude and latitude coordinates. An example of this is shown in Figure 6, where the user whose real location is depicted with a red dot, does 100 samples of noise perturbation with $\phi = 3.5$. We also depict a circle centered around the user real location of radius equal to the inverse CDF (percentile) of 95% for the Laplace distribution.

1) Privacy measurement: For an adversary $\mathcal{A}$, we define our privacy as :
Privacy = $1 - pr(\mathcal{A} \rightarrow (x, y)|(x', y'), \theta)$
where θ is the perturbation factor and $(x', y') = (x+X, y+Y)$ is the provided user location, perturbed from the users' real location with $X, Y \in LaplaceDistribution(\theta)$.

We evaluate this notion of privacy for our defense on user home locations and work locations since they are the most sensitive. i.e. we evaluate the number of times the adversary can correctly guess the users' real home address and work address.
From Kerckhoffs's principle, the defense should not be based on obscurity, thus the attacker always knows that the location has noise added and knows the noise parameters. We also assume that using the timestamp of the query, the adversary knows *a priori* that the query was made from a home location or work. We also assume, the adversary being a strategic one, has access to all of the users' queries in the database and can try to guess using multiple queries. By using these assumptions we evaluate our defense with a simulated attack on the queries using perturbed locations.

For every users' fake location, the adversary computes all the **POIs** of type home in the circle of radius equal to 98% percentile of the Laplace distribution centered on the provided location. Having seen all the users' queries, he then counts the number of occurrences of each of these plausible locations and guesses the home address that has occurred the most.
We follow the same principle for the users' queries done from their work locations. We count the number of correct guesses the adversary made over the total number of users in our database. In figure 7 we show the locations that have been correctly guessed in red, while the real locations, depicted in blue, are coupled with a red dashed line to their corresponding wrong server guesses, shown in black.

2) Utility measurement: We use two different functions to measure the utility of the service when the defense is applied. Having the service responding to the users' query by giving all the requested **POIs** in his corresponding grid, the utility corresponds to the number of responses that give the correct grid id. We call this utility, strict utility. But as seen in figure 5, having the server know correctly the users' grid id is

scale	utility	strict utility	home privacy	attack cost	work privacy	attack cost
1	100%	83.72%	5.5%	13.53s	0.5%	10.71s
2	99.92%	68.62%	18.50%	14.62s	11.5%	12.2s
2.5	99.66%	62.08%	28.00%	16.11s	16.5%	12.03s
4	96.97%	45.70%	51.00%	19.97s	37.00%	14.48s
6	89.05%	31.32%	71.00%	27.78s	47.00%	20.29s
8	78.85%	22.54%	81.00%	37.08s	58.00%	22.94s
10	68.95%	17.06%	90.00%	47.25s	68.00%	23.87s

TABLE I: Home and work privacy (with the attack cost in seconds) with general utility trade-off using different scale factors of the Laplace distribution for location perturbation.

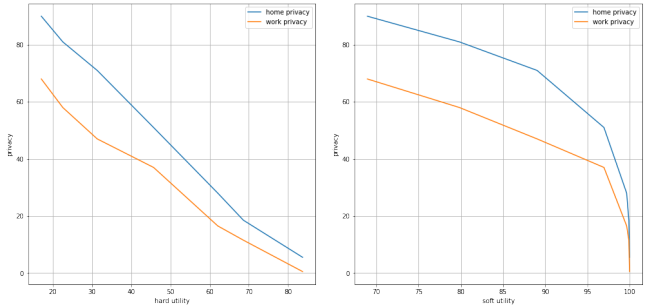

Fig. 8: Strict utility with privacy trade-off (Left). Utility with privacy trade-off(Right).

detrimental to most sensitive location privacy. We thus define a second measurement, called utility which counts the number of times the server either responds with the correct grid or one of the eight neighboring grids. In order to evaluate the utility, we apply our perturbation on all of the user queries and compute the utility functions based on the resulting fake location grid id and the real grid id.

3) Privacy-Utility trade-off: In Table I we demonstrate for different scale factors of the perturbation noise the privacy-utility trade-off. We also show the time it takes to apply the described attacks, which in hand demonstrates the change in the cost factor of the attack.

As predicted, the strict utility is negatively linearly proportional (Figure 8, Left) to our privacy definition. The second utility has a much more logical relationship with privacy, as seen in Figure 8 (Right) the relationship between privacy and utility is negatively sloped and concave, which implies that as privacy increases, the utility decreases in an increasing fashion and *vice-versa*.

4) Discussion: In order to achieve a modest amount of privacy, only 21% of utility is lost using our defense with a scale factor of 8. Note the defense could be improved by reducing the number of queries the user makes from the same location, reducing the attack space. This could be achieved by locally storing previous query responses and use them instead of contacting the server.
The scale could also be changed by the user, depending on the time of the day or if he is not in a sensitive location, the user could reduce the parameter to increase his temporary utility or simply by his choice, invoking privacy as control paradigm.

IV. Cell Fingerprinting via Network Traffic Analysis

A. Implementation details

Our Cell Fingerprinting implementation is divided into 3 distinct steps: the collection of the cell queries fingerprints, the feature extraction of the collected data, the pre-processing of those features for the classifier, and the classifier training.

1) Data collection: The data collection is done with a script running on the virtual machine at the client-side. It will first create a repository named after the current time and then, for each cell id, it will capture and analyze the packets for the query of the corresponding cell using *tcpdump*. The capture is only done on the packets for which the source IP address and the destination IP address are the ones from the server and the client. For each new cell query, a new *.pcap* file is generated, named after the cell id, and put in the repository created beforehand. The entire process is repeated n times, where n is the number of the desired samples for each cell. We have chosen n to be equal to 100, hence we obtained 100 (number of cases) * 100 (number of cycles) = 10'000 samples of data.

2) Feature extraction: Once the data is collected, we extract and format the features that will be fed to the classifier. We start by processing each repository (containing 1 cycle of 100 cases) into a *.json* file named after the repository. For each *.pcap* file, we extract the payload length of each of the packets. We distinguish the source of packets between server and client with a *sign()* function, where packets coming from the server side are set with a negative sign. We, therefore, obtain a list of values, representing the size, the order, and origin of the observed packets.
In addition, we measure for each packet the elapsed time since the first captured packet of the query. Thus for each cycle, we have a *.json* containing for each case label a list of packets length and time delay.
We also noticed that the acknowledgments packets significantly reduced the accuracy of our classifier, we therefore ignored them for the feature extraction.
We finally convert the obtained *.json* files in a global *pandas* dataframe, itself stored into a *pickle* file, that is easy to use for our pre-processing of data (see Figure 9).

3) Feature pre-processing: The longest packet exchange contained 803 total packets, thus our data was a $10'000 \times 802 \times 2$ matrix, where the last dimension contained the packet length, direction and time. For the messages with less than 802 packets, their rows had to be padded with zeros to obtain a uniform matrix.
Passing these raw features into our classifier, required high computation time, thus we computed several statistics that capture the features in smaller dimensions:

	case	fname	lengths	times
0	1.0	080446.json	[543.0, -1.0, -543.0, 1.0, 4077.0, -1.0, -1.0,...	[0.0, 0.0, 0.412, 0.412, 0.413, 0.414, 0.414, ...
1	10.0	080446.json	[543.0, -1.0, -543.0, 1.0, 4077.0, -1.0, -1.0,...	[0.0, 0.0, 0.464, 0.464, 0.465, 0.465, 0.465, ...
2	100.0	080446.json	[1057.0, -1.0, -543.0, 1.0, -543.0, 1.0, 1057....	[0.0, 0.0, 0.171, 0.213, 0.282, 0.283, 0.284, ...
3	11.0	080446.json	[543.0, -1.0, -543.0, 4077.0, -1.0, -1.0, 1121...	[0.0, 0.0, 0.343, 0.344, 0.344, 0.344, 0.344, ...
4	12.0	080446.json	[543.0, -1.0, -543.0, 1.0, 4077.0, -1.0, -1.0,...	[0.0, 0.0, 0.337, 0.337, 0.338, 0.338, 0.338, ...
...	...	...	...	...
9995	95.0	201006.json	[543.0, -1.0, -543.0, 1.0, 4077.0, -1.0, -1.0,...	[0.0, 0.001, 0.338, 0.338, 0.339, 0.339, 0.339...
9996	96.0	201006.json	[543.0, -1.0, -543.0, 1.0, 4077.0, -1.0, -1.0,...	[0.0, 0.0, 0.367, 0.367, 0.369, 0.369, 0.369, ...
9997	97.0	201006.json	[543.0, -1.0, -543.0, 1.0, 4077.0, -1.0, -1.0,...	[0.0, 0.001, 0.328, 0.328, 0.329, 0.329, 0.329...
9998	98.0	201006.json	[1057.0, -1.0, -1057.0, 1571.0, -1.0, 543.0, -...	[0.0, 0.001, 0.129, 0.131, 0.131, 0.14, 0.141,...
9999	99.0	201006.json	[543.0, -1.0, -543.0, 1.0, 4077.0, -1.0, -1.0,...	[0.0, 0.0, 0.381, 0.382, 0.383, 0.383, 0.383, ...

10000 rows × 4 columns

Fig. 9: Example of a pandas dataframe created from all the json files

- Instead of having the number of messages exchanged as the dimension on axis 1, we process it as a value for every sample.
- We compute the percentage of messages in the sample that come from the server side over the total number of messages in the sample.
- The individual message lengths are captured with 3 values: average message lengths the sample, the greatest and smallest message lengths. Due to the latter having small complexity, we also added the lower quantile message length.
- Finally we added the average maximum and minimum time between messages, as well as the total time length of the exchange.

Thus our final input dimension is $10'000 \times 10$.

4) Classifier: For our classification, we decided to use a Random Forest Classifier using the `scikit-learn` package, which is based on constructing multiple decision trees, and the output is the aggregation of all the trees in the forest.
We have also tried a Neural-Network with Cross Entropy loss, but we have achieved worse results with longer computation time than with the random forest classifier.
While using 200 trees for our classifier, the parameters for the forest have been selected using a grid search using one of the folds in order to fine-tune the model. The final parameters are shown in Table II.

The model provides the weights of the features for the decision and we notice, that the most important factor is the number of packets for the query. We plot the distribution of the number of packets in figure 10 and indeed the distribution is rich. In addition, by increasing the number of trees, the accuracy does not improve, suggesting the feature engineering made the model under-fit the data. We also notice that only the total time for the query is important for the model, we thus apply the model with the raw data features of packet lengths, aggregated with the total time of the exchange, thus the input dimension becomes $10'000 \times 803$. Note that with the different data, we changed the parameters: number of trees = 500, maximum depth = 20 and max features = *sqrt*.

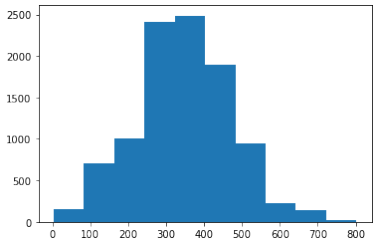

Fig. 10: Distribution of packet sizes.

n_estimators: 200	max_features: 0.7	max_depth: None
min_samples_leaf: 1	min_samples_split: 5	Bootstrap: False

TABLE II: Hyper-parameters of our Random Forest Classifier.

B. Evaluation

With a 10-fold validation using processed data with 10 features, with obtained an average accuracy of 59.6, with the following results for every Fold:
60.7 61.6 60.2 63.8 62.4 59.7 55.9 55.9 57.0 58.6

Using 803 features containing the total time of exchange and individual packet lengths and origin, resulting in an average accuracy of 68.76, with the following for every Fold:
69.8 69.8 68.7 72.1 78.0 71.8 62.8 69.0 59.4 66.2.

C. Discussion and Countermeasures

In conclusion, we have created two models, that both perform reasonably well. The first one, requiring only 6s to perform one fold, achieved an average accuracy of 59.58%, while the second requiring 56s per fold achieved a round average of 68.76%. The performance was firstly influenced by the number of possible cells and the accuracy would certainly drop if we considered a wider area. But the 100 cells assumption is still interesting because an average user will most likely query cells that are closer to his location, therefore within the range of 100 cells around him.

There exist many possible countermeasures to fingerprinting. One of them is caching, which we already discussed as a defense when talking about (De)Anonymization user trajectories in this report. Here the caching would limit the number of available samples for the training of the classifier, reducing its' accuracy. Another solution would be to obfuscate the average size of the packets by padding every exchanged packet. While being an effective solution for privacy, this would increase the communication latency and general overhead of the network between the client and the server.
One last solution could be to add multiple random queries to the one made by the client, or general random packets for noise, and send them at the same time or with small delays. This would greatly impact the three main parameters the classifier privileged for its guess: the order of packets, their size, and the delays between them.

REFERENCES

[1] D. Pointcheval and O. Sanders, "Short randomizable signatures," Cryptology ePrint Archive, Report 2015/525, 2015, https://eprint.iacr.org/2015/525.