CS-523 SecretStroll Report

Stuart Gueissaz, Kilian Lauener, Xavier Ogay

***Abstract*—We worked mainly together to grasp a full understanding of the project, also such that each member work equally.**

I. Introduction

The project aims to develop a privacy-focused location-based service application, addressing concerns around location tracking. The app provides users with details about nearby Points of Interest (POIs) only when requested, with anonymous authentication mechanism using attribute-based credentials avoiding continuous location tracking. This design emphasizes user privacy.

Each stage of the implementation will be carefully designed to address the identified privacy risks while maintaining the functionality and user experience of conventional location-based services. This approach aligns with modern privacy standards and user expectations for data security.

II. Attribute-based credential

We implemented Pointcheval-Sanders scheme into the system by carefully following the guide provided in the lecture notes. Which consisted mainly of translating the guide's formulas into Python code, and thinking about what to return from each function or how to define our types.

Among the six options of attributes given, we chose to use the first option:

- User attributes = [secret key]
- Issuer attributes = [all subscriptions; username]

First, we thought we needed to use option 4 as the project description advises us to use a secret key in the credentials. But after some reflection, we realized that the issuer attributes are revealed during the process so Option 4 makes no sense. Option 2 makes the user reveal no attributes, which is not considered good practice. We excluded Options 3 and 4 because putting all subscriptions in the user attributes makes it impossible to ensure the user accesses only subscriptions they paid for. Option 6 is not viable as the user name is part of the user and issuer attributes which can't be in a sane ABC. We hesitated to choose option 5 as it could be a viable option too but preferred to stay on option 1 as we wanted to give the same number of attributes to all users. So that signature have the same number of total attributes.
We reveal only the subscriptions the user decides to reveal ($-T$ option in the CLI). The secret key and username stay secret so the users are not identifiable based on the credentials they show.
All users have the same number of attributes hence, when two requests reveal the same attributes, they have the same number of hidden attributes as well. The subscriptions a user did not pay for are placed as empty strings in the signature, so they cannot cheat the payment system. The credentials we implemented have strong guarantees of anonymity. The data of the request itself may reveal more about the user though.

We used the Fiat-Shamir heuristic to make SecretStroll's zero-knowledge proofs non-interactive by building a challenge by hashing all the public information, then we built a Pedersen commitment to prove our signature was produced correctly. We helped ourselves by having a glimpse of the document "*Zero-knowledge Proofs of Knowledge for Group Homomorphisms*" from Ueli Maurer[1].
The challenge is built as $SHA256(commitment||pk)$ for the issuance request and as $SHA256(commitment||pk||msg)$ for disclosure proofs.
This challenge replaces the challenge we would expect from the verifier during a Pedersen commitment. The Pedersen commitment was adapted as follows in the system:
For each attribute i the user wants to include in their credentials, they select a random $r_i \in Z_q$ and produce the map $(i \to r_i - c * a_i) \forall i \in U$, they also include a $(-1 \to r_t - ct)$ to take into account the secret t mixed in the signature. c is the challenge produced before.

A. Test

We made some end-to-end tests and checked that the results and parameters were correct. We also thought of trying to compute manually some values to check if the results were similar but gave up as too much effort and we thought that if the end-to-end tests passed there would be a really small chance that a direct comparison would fail.

To assess the effectiveness of our tests, one ensures that our function pairs (e.g. sign and verify) work well in pairs. Negative tests were also used to guarantee some robustness in our functions. We mostly test at the API level of the $credentials.py$ file.

B. Evaluation

Benchmarks were run on the following config:
CPU: 12th Gen Intel i7-1265U (12) @ 4.800GHz
GPU: Intel Alder Lake-UP3 GT2 [Iris Xe Graphics]
Memory: 15429MiB

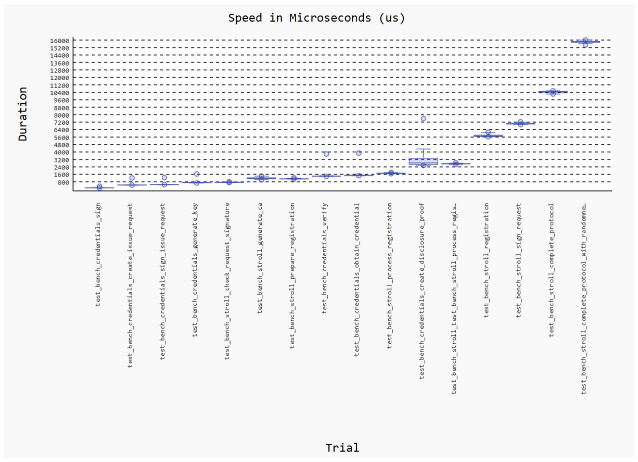

TABLE I: Performance of credentials functions

Function	Mean μs	Std μs
sign	154.0333	6.6702
create_issue_request	467.5993	44.7829
sign_issue_request	515.2935	57.0500
generate_key	711.0807	44.5075
verify	1,423.2856	122.9270
obtain_credential	1,496.4228	129.5842
create_disclosure_proof	2,991.3522	434.9541

TABLE II: Performance of stroll functions

Function	Mean μs	Std μs
check_request_signature	744.8391	12.6509
generate_ca	1,185.6771	83.4768
prepare_registration	1,127.9398	17.4776
process_registration	1,711.6663	23.3021
process_registration_response	2,737.6184	33.9735
sign_request	7,046.5992	72.7100

The issuance protocol part takes in mean time:
5,740.4179 μs ± 76.0788 (std)
The complete protocol with fixed parameters (username, subscriptions, and user subscriptions) takes in meantime:
10,433.8949 μs ± 104.3411 (std)
The complete protocol with randomness in parameters (username, subscriptions, and user subscriptions) takes in meantime:
15,785.2529 μs ± 130.7568 (std)

III. (De)Anonymization of User Trajectories

A. Privacy Evaluation

Given the simulated queries, the adversary would most likely be the server itself. The use of ABC is coherent with the data since users are authenticated anonymously. We suppose that user do not change IP addresses hence their data can still be grouped by IP. Another possible adversary would be one performing a passive man in the middle between users and providers. In all the strategies above, we have users not using Tor and only pseudo-anonymous given their IP.

Be it the malicious server or the man in the middle, the adversaries can gather quite a lot of information given the location data and the POI chosen. For the first and second attacks, both adversaries can execute them but at different scales, an adversary man in the middle could only attack to the scale of one IP while the adversary server could do it at the scale of every user. On the other hand, the third and last attack can only be executed by an adversary having access to all the queries performed (e.g. leak of the server query history or the server being malicious).

The first attack is leaking the favorite hobby of each IP by just searching the most queried type of POIs by IP.

TABLE III: Favorite Hobby

ip address	favorite hobby
0.98.248.97	dojo
10.229.150.53	dojo
100.255.65.73	dojo
101.193.212.180	gym
103.107.27.105	gym

The second attack consists of leaking the top three locations of the IP and inferring the type of location between Home, Work, and Activity. To be able to infer the type, we need to translate the timestamp of the queries into two categories: work time and free time. We decided to choose work time for the query sent during the interval of 9h00 and 17h00 from Monday to Friday and otherwise choose free time. From that we took the top localization during work time as the work localization, the top localization during free time as the home localization, and finally the second top localization during free time as the activity localization.

TABLE IV: Home localisation

ip address	home lat	home lon
0.98.248.97	46.510700	6.628843
10.229.150.53	46.558368	6.599673
100.255.65.73	46.555607	6.605922
101.193.212.180	46.535992	6.622526
103.107.27.105	46.538470	6.627223

TABLE V: Work localisation

ip address	work lat	work lon
0.98.248.97	46.546740	6.577377
10.229.150.53	46.546377	6.575353
100.255.65.73	46.527792	6.597571
101.193.212.180	46.542422	6.577282
103.107.27.105	46.546377	6.575353

TABLE VI: Activity localisation

ip address	activity lat	activity lon
0.98.248.97	46.513656	6.629130
10.229.150.53	46.556655	6.596498
100.255.65.73	46.549880	6.609449
101.193.212.180	46.537596	6.627838
103.107.27.105	46.535992	6.622526

The third attack is to infer the type of home, work, and activity of an IP by linking the localization to the closest POI of a possible type for it.

TABLE VII: Type of home, work and activity

ip address	home type	work type	activity type
0.98.248.97	appartment block	laboratory	restaurant
10.229.150.53	appartment block	laboratory	club
100.255.65.73	villa	office	dojo
101.193.212.180	villa	company	cafeteria
103.107.27.105	villa	laboratory	villa

The last attack consists of finding link between different IPs like neighbors, colleagues or people with the same activity by using, similarly as the third attack, the closest POI of the home, work and activity and searching people with the same POI ID.

TABLE VIII: Work colleagues

ip address	work colleagues
0.98.248.97	['138.53.90.242', '204.146.211.61']
10.229.150.53	['103.107.27.105', '139.251.47.207', ...]
100.255.65.73	['129.133.79.138', '200.20.52.81', ...]
101.193.212.180	['13.191.142.105', '237.144.218.252', ...]
103.107.27.105	['10.229.150.53', '139.251.47.207', ...]

TABLE IX: Home neighbors

ip address	home neighbors
0.98.248.97	[]
10.229.150.53	['203.24.85.254']
100.255.65.73	[]
101.193.212.180	['115.186.150.175']
103.107.27.105	['132.111.36.105']

TABLE X: Activity friends

ip address	activity friends
0.98.248.97	[]
10.229.150.53	['135.104.79.52', '233.228.129.122']
100.255.65.73	[]
101.193.212.180	['115.186.150.175']
103.107.27.105	[]

After executing our 4 attacks, we came to the conclusion that there are certainly way more leakages and exploits possible from an adversary server.

B. Defences

There 3 main leaks in privacy: the points of interest looked up, the location itself and the timestamp of the query.

Given the real-time nature of the app, there is nothing a user can do to hide the query time. The server knows when it received it and the user doesn't want to wait a random amount of hours before sending the query and getting an answer.

The point of interest is quite tricky as well. We could imagine something similar to `k-Anonymity`:
The user always queries for at least k POIs at the same time so the adversary cannot guess which one they really are interested in. Against a naive adversary, this would leave them a $1/k$ chance of guessing right. This solution has multiple issues, first, it would break the subscription system. Even if we ignore that, it will not work against a strategic adversary. If the $k-1$ other POIs are chosen randomly, an adversary will still observe a single POI that the user would query more than uniformly. If instead of choosing randomly, we had predefined categories, then either the element of the category shares semantic similarities e.g. bar, restaurant, coffee shop in which case they still leak information about the user habits. Or the categories do not share semantic similarities in which case the timestamp of the query could reveal which one the user really wanted (e.g. dinner time).

Finally, the location itself, we can apply laplacian noise on it. This is the most solid way of proceeding but still has caveats. First, there is a need for an appropriate noise level. The higher it is, the less relevant are the result for the user. Let us model the privacy and utility for secret-stroll:

The user looks for POIs in distance R of their position. By applying noise to their position, they approach the edge of this R-diameter circle around them. The bigger the noise, the more out-of-scope their result. We applied different levels of noise to determine how much of a utility loss and privacy gain it meant.

We defined utility as the difference between what answer the user would get on the original query and the one with added noise.

TABLE XI: Utility loss per noise level

Noise	**Accuracy**
laplace(0, 0.0001)	97.8%
laplace(0, 0.0003)	93.9%
laplace(0, 0.0005)	89.9%

The privacy gains are determined as follows, we consider the data inferred in our attacks as ground truth (best-case scenario). Then we perform the same analysis on the noised data and compare the changes in results, the more changes, the bigger the privacy gains. This noise mechanism offers deniability to the user.

TABLE XII: privacy gains per noise level

Category	**0, 0.0001**	**0, 0.0003**	**0.0005**
hobby	0%	0%	0%
home type	52%	52%	55.5%
home neighbours	96.5%	95%	96.5%
work type	20.5%	22.5%	27%
work colleagues	88.5%	93.5%	95%
activities	83.5%	85%	86%
hobby friends	65%	61.5%	62.5%

We can see that the hobby category is insensitive to noise which is expected given that it only relies on POIs asked in the query. As we explained above, this is unavoidable without changing the core of the app.

Some results above are surprising with some outliers like the hobby friends category. Overall the noise levels 0.0003 and 0.0005 both seem to offer fair privacy gains while keeping the utility of the data.

IV. Cell Fingerprinting via Network Traffic Analysis

A. Implementation details

To ease the trace collection process, we wrote a rather simple bash script meant to be run in the client docker. This script runs the following query for each grid i:
`python3 client.py grid $i -T restaurant -t`
while `tshark` runs in the background. Once the query returns, `tshark` is stopped, and the resulting trace is placed in `grid_i/c_j.pcap` where j is the j_{th} data collection. With this method, we retrieved 2600 traces (ie. 26 traces per grid point).

The traces were then read using Python. `Pyshark` was used to read them, and filter them such that we only had relevant packets left (ie. between Tor node and the user, no re-transmission packets, only TCP etc...). The following features were extracted per trace and used for the classifier:

- $Packet\_count$: *Total number of packets in the trace*
- $Outgoing\_packet$: *Number of packets sent by the user*
- $Incoming\_packet$: *Number of packets with user as destination*
- $Mean\_packet\_per\_second$: *The mean ratio of total packets per seconds*
- $Std\_packet\_per\_second$: *The standard deviation of the packet per second*
- $Ratio\_out/total$: *The ratio of outgoing packets compared to the total number of packets*
- $Ratio\_in/total$: *The ratio of incoming packets compared to the total number of packets*
- $Mean\_of\_seq$: *The mean of total number of the sequence numbers of packets*
- Std_of_seq : *The standard deviation of the sequence numbers of packets*
- Cap_size : *The total content size of the .pcap file*
- $Outgoing\_packet\_biggest\_len_i$: *The 5 biggest size of outgoing packet*
- $Incoming\_packet\_biggest\_len_i$: *The 5 biggest size of incoming packet*
- $Mean\_p/s\_i$: *The mean value of the packets seconds of the i^{th} of the 15 interval evenly spaced over the trace duration*

To see how we compute those features, take a look at the $compute\_stats.py$ file. The classifier is a RandomForest with 650 number of estimator.

B. Evaluation

Here is our evaluation of our classifier – the metrics after 10-fold cross-validation.

Metric	Average	Median	Std Dev
Accuracy	0.967692	0.965385	0.00753689
Precision	0.976388	0.975508	0.00560757
Recall	0.967692	0.965385	0.00753689
F1 Score	0.966766	0.964274	0.00764109

C. Discussion and Countermeasures

Considering that our predictive model operates within a grid of 100 possible outcomes, the performance achieved is notably impressive. This achievement is particularly significant when compared to the expected accuracy of a random classifier, which would have only a 1% (1 in 100) chance of correctly predicting a grid location. Now even with the usage of Tor we can correctly infer the grid location most of the time.

The network is an unreliable resource, packets may be dropped, reordered, or heavily slowed down in an unpredictable manner such that it can influence the performance of the classifier. All that causes noise in the capture. The usage of Tor makes it even worse as the path our packets take is longer and goes through machines of variable performance.

Our attacks rely mostly on the number of packets the server answers which is correlated to the queried grid (e.g. grid 1 may have 1 poi where grid 41 has 8). To implement countermeasures, we need to make the query and its answer independent from the view of an eavesdropper. One way is to have the server answer the same number of times and make sure each answer is of the same length.

References

[1] U. Maurer, "Zero-knowledge proofs of knowledge for group homomorphisms," *Designs, Codes and Cryptography*, vol. 77, pp. 663–676, 2015.