CS-523 SecretStroll Report

Benoit Knuchel (270320), Bradley Mathez (260413)

***Abstract*— We implemented an attribute-based credentials for a location-based application. We then made a privacy evaluation of a dataset leak. Finally, we analyzed the network traffic metadata to implement a cell fingerprinting attack using machine learning.**

I. Introduction

This project is named SecretStroll. We developed a location-based application that allows users to look for nearby points of interest (POI). However, we wanted this app to be more private than getting the exact position of each user for every requests. We also wanted our app to be a paying one, this is why we want to be able to know if a request has been issued by a registered user. For that part, we built an anonymous authentication mechanism using attribute-based credentials. The whole cryptography is implemented using the library *Petrelic*.

We started by writing the different parts to manage the credentials using a *Pointcheval-Sanders* (PS) *attribute-based credentials* (ABC) scheme. Implementation details will be explained in the first part of that report.

We then simulated a dataset leak and did a privacy analysis from different attacks and also proposed a way to defend users against some of them.

In the last part of this project we used machine learning to be able to recognize from which cell a request has been made. This means that we can approximately locate the users based only on the metadata transmitted.

II. Attribute-based credential

We used the PS scheme to be able to recognise if a request has been made by someone who is allowed to obtain the results because she paid for it. We proceeded as follows.

The user subscribes to the server using the PS scheme by signing a list of one attribute (we took a part of the client's secret key as single attribute of the list) and we sent in plain text the subscription list from the client to the server. Then, the server uses this list to sign the subscription request and sends the result to the client. The client uses the server's response to derive credentials that will be used to sign his/her requests.

The attributes are the possible subscriptions items (e.g. *bar*, *dojo*). We have associated an index and a *Big number* (i.e. `Bn`) to each of them in order to be able to compute the cryptography needed for this project.

The Fiat-Shamir heuristic has been used to make the zero-knowledge proofs (ZKP) non-interactive. It computes a hash of the public attributes as well as the the previous commitments and takes the result as a challenge to be sent along with the packet. Then we will recompute the challenge using the provided values and verify that the one that has been sent is equal to the one we have recomputed.

We used it for both the disclosure proof (created by the client to send a query to the server which verifies it) and for the registration process. The client generates the proof and the server verifies it. We are first going to discuss about the registration process, then we will explain the queries' signature's creation process.

We will start by explaining the first ZKP (the one used to register a new user). We used this to prove that the user did correctly commit to its attributes . We assume that the server's public and secret keys (`srv_pk` and `srv_sk`) have been already generated. Based on the lecture note about the PS scheme, we obtained `srv_sk = ((x, X, y), valid_sub)` and `srv_pk = (g, Y, gt, Xt, Yt)`. The registration request contains `(commit, (Rnd_t, Rnd_is, challenge, s_t, s_is))`. We defined them as:

$$
\begin{aligned}
commit &= g^t \cdot \prod_{i \in is} Y_i^{user\_att_i} \\
G1elem_i &\in G1 \ , \forall i \text{ (i.e. random elements from } G1) \\
t &= G1elem_1 \\
rnd\_t &= G1elem_2 \\
Rnd\_t &= g^{rnd\_t} \\
rnd\_is &= [(i, G1elem_{3i})] \ , \forall i \in is \\
Rnd\_is &= [(i, Y_i^{rnd\_is_i})] \ , \forall i \in is \\
s\_t &= g^{rnd\_t} + challenge \cdot t \\
s\_is &= [(i, rnd\_is_i + challenge \cdot user\_att_i)] \ , \forall i \in is \\
challenge &= \texttt{Bn}(\texttt{absolute\_value}(hash(Rnd\_t) \\
&\quad + hash(pk) + hash(Rnd\_is) + hash(commit)))
\end{aligned}
$$

where is is the indexes of the user attributes, but here the user's attributes are only composed of a part of the user's public key (i.e. $client\_sk\_x$) with index 0. $user\_att_i$ represents the user's attribute assigned to the index i. We used the Fiat-Shamir heuristic in the $challenge$ computation as described above. We chose SHA3-512 as a hash function and for the hash of pk and Rnd_is we have computed the sum of the hashes of each parts composing the objects.

Now, the server wants to verify if `sig1` is equal to `sig2`.

$$
\begin{aligned}
\texttt{sig1} &= (\prod_{i \in is} Rnd\_is_i) \cdot Rnd\_t \cdot (commit^{challenge}) \\
&\overset{(1)}{=} g^{y_0 \cdot rnd\_is_0} \cdot g^{rnd\_t} \cdot (g^t \cdot \prod_{i \in is} Y_i \cdot user\_att_i)^{challenge} \\
&\overset{(2)}{=} g^{y_0 \cdot rnd\_is_0} \cdot g^{rnd\_t} \cdot (g^t \cdot g^{y_0 \cdot cli\_sk\_x})^{challenge} \\
&= g^{rnd\_t + y_0 \cdot rnd\_is_0 + challenge \cdot (t + y_0 \cdot cli\_sk\_x)} \\
&= g^{rnd\_t + challenge \cdot t} \cdot g^{y_0 \cdot (rnd\_is_0 + challenge \cdot cli\_sk\_x)} \\
&\overset{(3)}{=} g^{s\_t} \cdot \prod_{i \in is} Y_i^{s\_is_i} \\
&= \texttt{sig2}
\end{aligned}
$$

In (1) we have written the *commit* under its definition form and developed the product to a single element due to the size of is. (2) and (3) are also due to the size of is. The variables' names in this development correspond to the ones present in our code.

The disclosure proof is more complex but keeps the same core idea. In the previous development we generated a random t (Rnd_t) and random user's attributes (Rnd_is). For the disclosure proof, we generate a random signature ($sigp = (sigp_1, sigp_2)$) from credentials obtained by deriving the registration request signed by the server. The hidden attributes are the user's attributes as explained before. We proceed as proposed in the ABC guide received for this project.

We also add a ZKP to the disclosure proof. We reused the same principle as the one described before. The Fiat-Shamir's principle has been taken into account during this procedure as well so that the ZKP is non-interactive. However, we replaced the hash of the commitment by the hash of the disclosed attributes and the hashes of both $sigp_1$ and $sigp_2$.

A. Test

We tested both files ('credentials.py' and 'stroll.py') separately. We decided to write tests step by step while developing the functions. We have tests for each big functions provided by the skeleton. We will begin by talking about credentials and then we will discuss about stroll.

1) Credentials tests: We started by testing the three basic functions (i.e. `generate_key`, `sign` and `verify`). We generated a given number of random messages that will be signed and then, we verified the signature.

Firstly, we tried the correct path without modifying anything. After that, we signed the messages and replaced the first element of the signature (i.e. a generator) by the neutral element of G1. In the third test, we signed the messages and replaced the content of one of them by a new random message and tried to verify the signature. Our last test generates a new random signature and tries to verify it.

We think that these tests are enough to detect many problems in our implementation for that first part, because we tested the main problems that could occur with the three simplest functions of credentials.

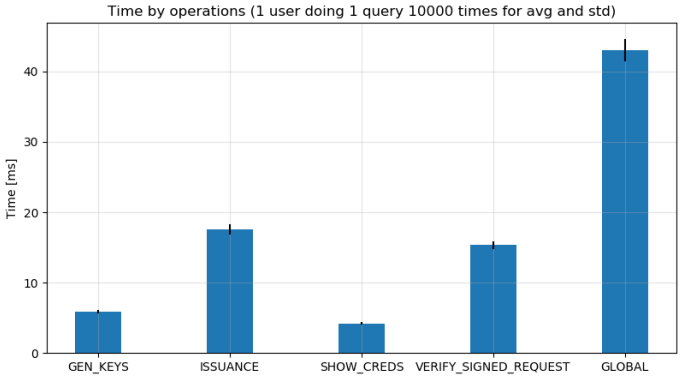

Fig. 1: Average time for each operation

Then we tested the issuance protocol. In this procedure, we have three operations to verify. They are `create_issue_request` (i.e. the user commits to the attributes wanted in the credentials), `sign_issue_request` (i.e. the issuer verifies the validity of proof with respect to the commitment and generates a blind signature) and `obtain_credential` (i.e the user checks the signature and verifies if it is valid for a PS scheme).

We begun by creating an issuance request. Then we tried to sign it normally. After that, we tried to sign a correct request with a new (bad) ZKP created with wrong user's attributes to verify that the signature was correctly verified. The last two tests for the issuance protocol test the *"unblinding of the signature"*. We tried to obtain credentials through the good path but also with a bad signature.

Finally the last part is the disclosure proof. For that part, we created and verified a normal disclosure proof, while verifying the length of the disclosed attributes list and that none of the disclosed values is in the hidden list.

2) Stroll tests: We begun with testing a standard workflow, without errors (i.e. the correct path). A server is initialized with a list of supported types. A client registers to the server with some of the supported types and requests POIs for a given position. The last check is the server checking the request's validity.

Otherwise, we tried to request a type of subscription which is not supported by the server or for which the client is not subscribed to (two different failure paths). We also tried the behaviour if two clients with different names registered to the same types requesting the same types could forge a new request using the signature of one of the user and the proof of the other. It must fail and the server should not able to verify the signature.

To conclude, we tested every possible cases we were able to think of. We think that it is not perfect but as long as we insert randomness in a lot of tests, we should be able to discover more issues than with simple static tests.

B. Evaluation

In order to evaluate our implementation, we took some of the tests we wrote and added timings and counted the bytes

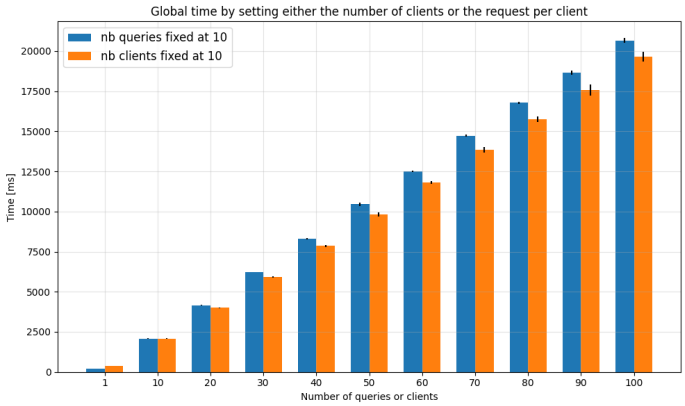

Fig. 2: Average global time with a fixed parameter (number of clients or queries per client)

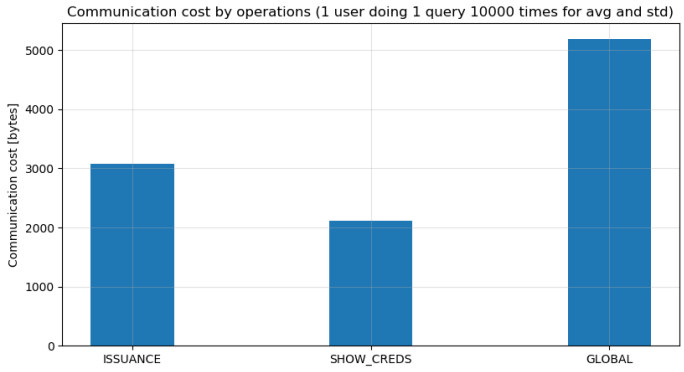

Fig. 3: Average communication cost for each operation

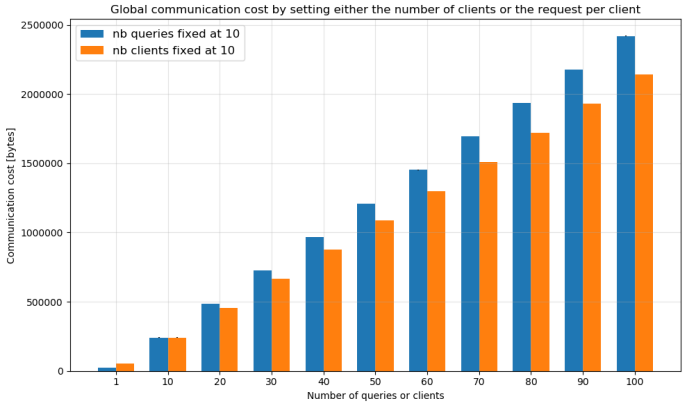

Fig. 4: Average global communication cost with a fixed parameter (number of clients or queries per client)

	Gen keys	Issuance	Show creds	Verify	Global
Time[ms]	0.25	0.73	0.21	0.56	1.62

TABLE I: Standard deviation of time for each operation

Time[ms]	Queries fixed	Clients fixed
n=1	1	6
n=10	10	10
n=20	21	27
n=30	20	39
n=40	49	58
n=50	97	120
n=60	43	75
n=70	67	185
n=80	69	174
n=90	120	345
n=100	151	288

TABLE II: Standard deviation of global computation time

	Issuance	Show creds	Global
Bytes	3	2	4

TABLE III: Standard deviation of communication cost for each operation

Bytes	Queries fixed	Clients fixed
n=1	8	15
n=10	70	83
n=20	141	127
n=30	307	221
n=40	337	320
n=50	373	328
n=60	507	576
n=70	707	399
n=80	805	766
n=90	973	590
n=100	985	812

TABLE IV: Standard deviation of global communication cost

transmitted. The file we used is 'stroll_evaluation.py'. Note that on Fig. 1, 2, 3 and 4 the black lines at the top of the bars represent the standard deviation.

We start by analyzing the computation time. From Fig. 1 we can see that 2 operations, the issuance protocol and the verification of the request, clearly take most of the computation time. This is not surprising since most of the cryptography are done in these parts. Also the total time it takes for a user to make a request, including the key generation, and get the result is less than 50ms. From Table I we can see that the standard deviation is small. From Fig. 2 we see that the total time grows linearly with both the number of queries or the number of clients. The standard deviations are in Table II, again we can say that they are small e.g. with 100 clients each making 10 requests the average time is approximately 20s with a std of 0.15s. The hardware on which we run the code will have a great effect on the computation time.

For the communication cost we did not take into account key generations and credentials verification since there is no communication involved. We can see the averages of the communication cost by operation on Fig. 3 and the standard deviations in Table III. So for a user to register and make

one query, it takes a bit more than 5000 bytes with a std of only 4 bytes. Note that the issuance protocol is more costly than showing the credentials (by about 1000 bytes). Now for the global communication cost, the averages are on Fig. 4 and standard deviations in Table IV. We see that the average and the standard deviation of the total number of bytes grows linearly with both the number of queries or the number of clients. Note also that the standard deviations are really small e.g. with 100 clients each making 10 requests the average number of bytes transmitted is approximately $2'500'000$ bytes with a std of 985 bytes. The hardware on which we run the code will have little to no impact on the communication cost.

III. (De)Anonymization of User Trajectories

A. Privacy Evaluation

To analyze the simulated data we used a Jupyter notebook, named 'privacy_evaluation.ipynb', that you can find in the code of the project. For the first 3 attacks, we assume that the adversary has access to the dataset 'queries.csv' and has no prior knowledge on users. This is the typical case of an adversary who wants to collect a maximum amount of private information that could be sold later on. We define the privacy as the probability that an adversary obtains private informations on users such as the ip address, home address, working place or interests. We define the utility as the probability that the user is located in the right cell.

For attack 4 we will take the same adversary as before but he also has prior knowledge on a person. His goal here is to find more precise and private informations about a person he knows already.

Attack 1

The goal of the adversary is to find the interests of some users. We defined the interests as 'gym' and 'dojo'. One could argue that bars and clubs are interests but we wanted to focus on hobbies. This is easily implemented by just filtering the interests. We can find these for 166 different users. This attack clearly reduces the privacy of the clients but it is not severe since after finding their interests the adversary cannot link them to a real person but only to an ip address.

Attack 2

The goal of the adversary is to find the home address of some users and their interests (for example to sell these informations to advertisement companies). Since having the interests of the users are easy (see Attack 1), the challenging part is to find their home. For this we assumed that requests done between midnight and 6am are from home. Also we removed requests for clubs or bars at these time since it is more likely that users are out. For example, if a user searches for a gym during night time, we can be quite sure that he did the query from his home. The size of the dataset under these conditions is 79 so we can just print the results and then analyze by hand. For most users there are multiple queries under these conditions and we can see that the positions differ slightly, which means that, fortunately, we cannot tell exactly in which house they live in but we know their neighborhood.

Attack 3

The goal of the adversary is to find users working place. For this we want to be sure that users are at their working place, so we select a time slot from 9am to 4pm and from Monday to Friday. We also filter the interests to gym, bar, cafeteria and restaurant because that's what people would like to do either with their colleagues (restaurant, cafeteria or bar) or alone (sport after work before heading home). This gives a dataset of 2046 different queries for 200 unique users. This means that in average there are 10 queries per users under these conditions and so we can safely say that the most appearing position is their working place.

Attack 4

Here the adversary has some prior knowledge on a person : he knows the working place of a specific user and also that this user is always looking for new places to eat at noon. His goal is to find his/her ip address. For this we select the queries made from coordinates '46.53294222140508', '6.591174086010503' with POIs 'restaurant' or 'cafeteria' with time from 11am to 1pm. This gives 5 different ip addresses. Which means that the adversary defined above did not exactly achieve his goal since he still needs to identify one address from 4 others. Nevertheless this attack can still be considered as severe since there is non-negligible probabilities that it deanonymizes a specific user.

Other attacks

We presented what we believe are the 3 big attacks possible with no prior knowledge on users. Things get more complicated with priors, there could be different attacks possible. Attack 4 was an example but we could imagine others such as : if an ISP gets access to the database they could deanonymize its clients, an insurance company could raise the premium if they see that a user already had alcohol problems but still continues to go in bars,...

B. Defences

As client side defence we chose spatial obfuscation : we added noise (from a laplacian distribution) to the location. Since the laplacian distribution is defined by its mean (obviously set at 0) and scale we just need to find the scale. After some testing we found that modifying the coordinates by a standard deviation of $1 \cdot 10^{-6}$ ($\sim$ 115 meters) results in a good utility loss - privacy gain trade-off. Since the standard deviation of the laplacian distribution is equal to $2b^2$ with b the scale, we find that $b = \sqrt{\frac{1}{2} \cdot 1 \cdot 10^{-6}}$.

This kind of defence will not have any effect on attack 1 since it does not depend on locations but for attack 2, 3 and 4 it can be efficient depending on the number of queries made by a user. Since we used zero mean noise, an adversary could average the results and get a good approximate.

For attack 3, in average we have 10 queries per user so we computed the mean of the coordinates and compared the results to the real dataset. We only did this for 3 users, just to have an idea of the efficiency of our defence for this attack : we got that the reconstructed coordinates are wrong by 50m to 330m. So we can conclude that since the goal of the adversary

is to find the working place of users, it does protect them pretty well. Note that this may change if the adversary has access to more queries. The same applies to attack 2 except that there will be less queries and so users are even more protected.

Attack 4 is not possible anymore since the coordinates have been modified. Note that even with a noise that is small the attack would not work since the adversary knows the exact location of the user.

With this setup, the grid accuracy is 15%, meaning that 85% of the time the users will receive the POIs from the right cell. The utility loss is not big since a user who is perceived in the wrong cell can just request again until the right POIs are received (note that the probability that a user has to request more than twice is less than 1%). The privacy gain is significant since attacks 2, 3 and 4 are not effective anymore.

IV. Cell Fingerprinting via Network Traffic Analysis

A. Implementation details

For the data collection part, we implemented a script 'capture_client.sh' using 'tcpdump'. For each cell on the grid, it will make 20 requests to the server and capture everything.

We used python, specifically 'scapy', to read the packets and extract the features (file 'extract_features.py'). From the course we saw a list of the most important features for website fingerprinting so we decided so start with some of them. We selected these 5 : the number of incoming packets, number of outgoing packets, number of outgoing packets as a fraction of the total number of packets, number of incoming packets as a fraction of the total number of packets and the total number of packets. As the sixth feature, we selected the length of the HTTP packets going from the proxy to the client. These packets contain the poi ratings, which are almost identifiers of pois, so we thought that our algorithm could benefit from using their size (there are 18 of them in each request).

In order to train the classifier we created the features and labels as follows : for each request of each grid we created an array of $5 + 18 = 23$ values containing the features and associated it to the grid number (the labels). This gives 20 labels (each with an array of 23 features) per grid so in total $20 \cdot 100 = 2000$ labels for our algorithm to learn from.

B. Evaluation

We used two different metrics to evaluate our classifier : first the accuracy, in %, and second Cohen's kappa coefficient $\kappa \in [0, 1]$. We chose the kappa coefficient as a way to measure how much better is our classifier than random predictions.

As results we get an accuracy of 100% and so $\kappa = 1$ since the prediction is perfect. We are surprised by these results and wonder whether we made something wrong. Nevertheless this accuracy really comes from the length of the HTTP packets received by the clients : the values are clearly different between each grid. Also if we remove these lengths as feature, we only get an accuracy of 15% and $\kappa = 0.15$.

C. Discussion and Countermeasures

We noticed that in 'server.py' before returning the list of poi ratings some padding noise is added but the noise level is quite small. If the noise level was much higher then the attack would not be as good. This could be a great defence.

Another defence possible would be to pad until every grid has the same number of poi. This way, an attacker loses the possibility to differentiate two grids by looking at the number of http responses (i.e. by counting the number of poi requested). To implement this solution the server could transmits some fake numbers as `poi_id` in the response of the first request and then transmits some random garbage that could be recognized as garbage by the client and not taken in account. However it will generate more traffic between the client and the server. An alternative to making every grid having the same number of poi may be to use the concept of *k-anonymity* among the number of poi in each grid (i.e. being sure that at least k grid have the same number of poi) and adapting k to the privacy needed.

A different idea to be used as defence would be to order the requests made by the client by their sizes (i.e. poi with smaller number of data requested first). Doing so will restraint the capacity of an attacker to distinguish two grids with the same number of poi and both have requests having the same sizes but in different order. For example, requesting $grid_1$ gives packets with sizes $(100, 200, 300)$ and requesting $grid_2$ gives $(300, 100, 200)$. Without reordering to obtain $(100, 200, 300)$ for both allows an attacker to know which grid has been queried. We could implement that solution by adapting the server to transmit indexes with the poi_id to request and the client to requests the poi_id following the order given by the server.

The last defence we thought about is to pad the packets' sizes to obtain a k-anonymity within these (i.e. making sure that at least k packets have the same length to be indistinguishable). The we will adapt k and the padding to obtain as much privacy as we want. Doing so, an attacker would not be able to distinguish two requests with not so close sizes. For example, if requesting $grid_1$ gives $(100, 200)$ as the size of the response and requesting $grid_2$ gives $(75, 213)$ without this defence an attacker would be able to predict which grid has been requested. With this defence, $grid_1$ would answer $(100, 213)$ and $grid_2$ would answer the same. Thus they would be indistinguishable.