CS-523 SecretStroll Report

Malo Perez, Maxime Sierro

I. Introduction

The goal of this project was to implement an attribute based credential system and integrate it to a location based application, as well as evaluating the computation and communication costs of the the system. The second objective was to perform a privacy evaluation of this application and then propose and implement a defense. Finally, we have run a data collection on user queries in order to implement a cell fingerprinting attack.

II. Attribute-based credential

The Attribute Based Credentials system was implemented following the protocol seen in class, and by implementing ourselves the non-interactive zero knowledge proofs.

Secret Stroll was mapped to the previously implemented Attribute Based Credential system, by considering each possible subscriptions type as a possible credential, creating a public and private key of size number of possible credentials + 1 (for the users secret key).

Here, when a client registers and prepare the issuance request, it only considers its secret key as a "user-attribute" and thus making sure the server signs it without seeing it. The server than produces the credentials and valid signature for the requested subscriptions, which corresponds to the "issuer-attributes". A limit of our implementation is that the server always needs to create the same number of credentials, so the rest of the signing and computation works. This means that if the client doesn't request all possible subscriptions, the server will have to create some "dummy" credentials corresponding to empty strings to make sure the right number of credentials is sent to the client.

When making a request to the server, the client creates a disclosure proof, choosing only the requested subscriptions types as disclosed attributes (so not showing to the server its own secret key or subscriptions the user has credentials for but are not part of the request). This ensures that the user only reveals to the server the minimal amount of information possible: the subscriptions needed for the request. The server then checks that the signature on disclosed credentials and query is indeed correct, and proceeds to computing the service.

To make the zero knowledge proofs non-interactive when creating issuance request or the disclosure proof, the challenge was created by concatenating the server's public key, the commitment, the prover's commitment R, and in the case of the disclosure proof, the message corresponding to the location request. We then used the hashlib libaray to create a SHA-256 digest, which served as the challenge for the rest of the protocol.

A. Test

The system was tested gradually in `test_ABC.py`, first by testing each building block from `credentials.py` as we went, and then creating different scenarios to test `stroll.py` in its entirety. One scenario was a correct one, and two others were "bad" scenarios, were the client were either asks for a service for a subscription they have no credentials for or asks for a service which is not part of the possible subscriptions. A good way to assess how effective the tests are would be using coverage as a metric, either by making sure every line of the code is executed at least once, or by checking all paths are executed at least once (every combination of branching paths). Also exploring corner cases, as well introducing some form of randomness is important to cover a good range of scenarios.

B. Evaluation

Evaluation of the communication and computation costs was done be running the whole protocol from SecretStroll (key generation, issuance, signing, and verification) 25 times, and for each iteration logging the number of bytes sent between the server and client, as well as the time taken, for each of the 4 steps. This was done for five numbers of attributes (1, 5, 10, 20 and 50). Each of the possible subscriptions names is generated as a random string of 8 characters.

Fig. 1 shows us the computation costs for each of the four steps. We can see that the showing of credentials (creating disclosure proof) is the step that takes the less time among all four, with the three others growing linearly with the number of attributes. The standard deviation doesn't evolve much with the number of attributes, except when verifying credentials, where having more attributes makes the variance grow.

We can see on Fig. 2 the communication costs for Key Generation, Issuance and showing of the credentials. We did not include communication costs of the verifying credentials step, as no message is exchanged between the client and server at this step. The standard deviation is not displayed either, as the amount of bytes sent was constant across all 25 runs of each step for each number of credentials. We can see here as expected that the number of attributes has the biggest impact on the key generation generation and sharing. Indeed, it grows linearly with the number of attributes. For the other two steps, the effect is not as important, as the messages sent between the client and server are less dependent on the attributes (issuance requests, zero knowledge proofs and

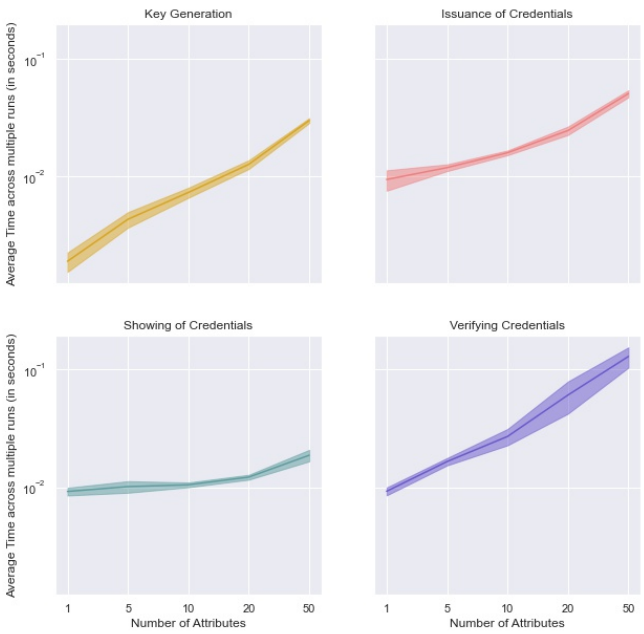

Fig. 1: Average time taken to execute each part of the protocol (in seconds) depending on the number of attributes. (log scale)

.

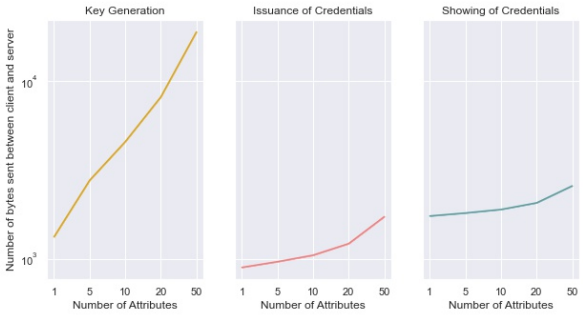

Fig. 2: Number of bytes exchanged between the server and client depending on the number of attributes. (log scale)

.

location request)

III. (De)Anonymization of User Trajectories

A. Privacy Evaluation

We assume the adversary is the service provider and that it is semi-honest : it tries to learn as much as possible but follows the protocol. The analysis is thus also useful in the secondary case where another malicious party is able to access the data of a fully honest service provider. The general idea is that such an adversary has access to the same information contained in the two CSV files, but does not try anything else outside of inferring information from them.

The specified assumption also needs to be clarified/extended. We not only assume that no IP address can be used by different people, but also assume the stronger fact that every user will always have the same IP address during the studied 20 days. Depending on the interpretation, this is not necessarily true in the specified assumption but we felt it made sense to assume as much since this first part should not yet include users who actively try to defend their privacy. With the same thinking, we assume that the location data has not be tampered with and is the exact position of the user when they makes a request.

Because of those assumptions, the information collected proves to be extremely valuable and as such is a serious threat to the privacy of users, even if the service provider follows the protocol. The Jupyter Notebook contains examples of attacks using the collected queries. The first thing a semi-honest service provider should do is translate the ”timestamp” field values into a weekday/hour format : this way, it can learn habits depending on the day of the week as well as the time of day. Finding out where a user lives as opposed to where they work is automatically easier since it can differentiate weekdays from weekends. Finding out about the interests of a user is also easy as long as this interest is part of the offer proposed by the service. The provider can look up the frequency of particular searches (e.g. ”Club”) as well as the timings : learning that a person likes to go clubbing every Monday is a good example of a problematic breach of privacy. Additionally, it is safe to assume that the searches reflect a real interest and not mere curiosity since they require a paid subscription. An example of a clever attack could be to identify a person thanks to its precise address and workplace inferred from the data collected, and threaten them to tell their employer that they frequently search for bars in the middle of their work hours during the week. If said user uses the app while inside a club, its location would also confirm that he actually went there and was not querying bars for later. This kind of tactic could be used as part of a ransomware.

As illustrated in the Jupyter Notebook `privacy_evaluation/Part 2.ipynb`, those attacks can easily be mounted simply by filtering each row to fit particular criteria. This is very inexpensive and could be made even more efficient using SQL queries and data modelling for example. This kind of data is after all extremely valuable compared to how lightweight it is : each query contains valuable information and is only a couple of bytes. The location and timestamp recorded are also needlessly precise for such a service. Because of this ”lightweightness”, attacks on a very large number of queries should not be problem and open new kinds of privacy breaches : an avid user which frequently sends queries could have its actual movement tracked as well as its sleep patterns, etc.

Inferring relationships between users seems hard however. In a realistic setting, only one person of a group should be using the app to look for a destination and this seems to be reflected in the data. Moreover, even if we find out that multiple people carried out the same search at the same time around the same place is not necessarily proof of a relationship : many people could simply be looking for a restaurant at 1pm which is perfectly normal. This could however show that multiple employees of a same company share the same schedule for example : the location context is crucial in many cases. The experiments conducted comforted us in the conclusion that attacks which focus on one user at a time by filtering queries are the best investments of resources for potential attackers.

B. Defences

For a client-side defence, the naive option would be to introduce randomness per query to hide precise information : delay the sending of the query, add noise around the position, etc. However, those would highly impact the utility of the service for users negatively. The only case where adding random noise does not impact utility is in the case of the location, by making sure that the sent location stays in the correct cell, however this is not allowed in this exercise as is switching IP addresses which would also improve privacy for users without any utility drawbacks. We thus need a different kind of strategy : instead of modifying the queries, a user could simply ”hide” them along many additional randomised queries so that the service cannot distinguish which ones are real.

Such a solution could obviously be automated, so that the user only needs to manually take care of the ”real” queries whose results they are interested in. There are many reasons why such a strategy is suited to this particular context. First, the light weight of both queries and results means that sending and receiving them is not expensive for the user's device. The results to these ”fake” queries may even be ignored by the device in order to alleviate the workload. Moreover, since the user pays a monthly subscription, it is fair to assume that the service provider is not going to inflict a penalty on a frequent usage of the service, especially when the aforementioned light weight of the data transfers is taken into account. Ideally, the randomised queries that are periodically sent by the user's device should be realistic so that a potential attacker cannot distinguish them from real, manual queries. For example, a query for clubs at 14:00 in a rural area would be ignored by a smart attacker and should probably not be generated by a good defence mechanism. True randomness would also be bad for the location data : an attacker could single out a location which is the exact source of multiple queries and assume that it is not random and is the house/workplace/etc of the user, since it is extremely unlikely for true random queries to have the exact same location multiple times. Here too, the defence mechanism should aim at ”realistic randomness” and intentionally produce many fake queries originating from a set of fake locations, so that an attacker could mistake them for the home or workplace of the client. Many other factors such as this should be taken into account to achieve this realistic randomness and be effective versus a smart attacker.

With this strategy, the advantage is that a user will always be able to make a real query and get relevant results without worrying about hiding information inside the query. This is why it can be argued that the utility of the service is not affected for the client, at least not directly. Moreover, if it is impossible for an attacker to distinguish the real queries from the fake ones, every attack should be extremely expensive to mount as long as the attacker is still semi-honest : the data becomes almost useless to them ! The downside is how much resources the user is willing to spend to ”drown” their real queries in a large quantity of randomised ones. Because of the aforementioned light weight, this strategy could be taken very far but limits do exist : sending a random request to the server every second would probably be overkill for example. A naive mistake would be for those requests to be separated by a constant time : a smart attacker could single them out easily and obtain the real queries. Sending many fake requests during night time would also be a poor spending of resources since there are less true queries to hide (unless the user wants to hide its sleep schedule ?). Another drawback is that hiding interests may require subscriptions which do not matter to the user, which costs money. However, this drawback is not inherently caused by this strategy and should be a problem for any defence mechanism unless the user illicitly acquires interests he did not pay for, which is not a desirable scenario. Each user should judge how much extra money they are willing to spend to hide their interests.

More interesting dilemmas occur when we assume that every user is going to use this defence, since the increased volume of queries could become problematic for the service provider. As a response, it could introduce penalties depending on the volume of queries, such as prices which scale with the client's usage frequency. Users will again need to choose how much they are willing to spend for their privacy. Heavy users will need correspondingly more randomised queries to hide their real ones and will need to pay more to achieve the same level of privacy. In the case of a different kind of penalty, such as a limit on how many queries are allowed per day (100 for example), the user will need to think about the utility-privacy trade-off : are they willing to do 10 real queries today if this will only leave 90 other queries to be randomised to hide the real ones ?

Assuming a strategic adversary which is aware of the existence of the randomised queries, our quantification of the user's privacy is expressed as the ratio of the number of its randomised queries to the number of all of its queries. This represents the likeliness that an attacker is wrong in suspecting that a particular query is a real one, and thus

assumes that the randomised queries are "ideally realistic", meaning that they are individually indistinguishable from real ones to any actor other than the user itself. Note that without the defense in place, this privacy level would equal 0, which may seem like an extreme number but is fair considering how valuable the information contained in each query is, as was seen in the previous part. Also note that if two users generate the same number of randomised queries, the user which uses the app more often will have a lower privacy with this formula, which is intuitive. Achieving a privacy of 0.9 for every client would require 10 times more queries, that is a total of $20'443 * 10 = 204'430$ queries with our experimental CSV data. For a privacy of 0.99, the number would rise to $2'044'300$. We can see that this privacy would require considerable increases on the workload of the server and the clients' devices, which is our utility drawback. Since we focus on the privacy of individual users and that the randomised queries can have their location set anywhere in the area, the points of interest details are not relevant in our case.

The Jupyter Notebook also contains an experimental implementation of such a defense, though in a simplified form to fit the scope of the project. We cannot realistically implement the aforementioned "realistic" randomness which must take a huge number of factors into account, as well as analyse enormous quantities of existing manual queries. Our implementation achieves a privacy level of 0.75 while focusing on the location privacy of every single user. The biggest takeaway is that the strategy is especially effective versus adversaries which cannot verify if individual queries are real or not. For attackers with this capability, the defense multiplies the work they must do, but increases the workload on users' devices and on the server by the same factor. Since creating one fake query of a couple bytes should realistically be much cheaper than verifying its correctness as an attacker, we think that the additional workload should generally be worth it.

IV. Cell Fingerprinting via Network Traffic Analysis

A. Implementation details

The goal of this part is to use machine learning techniques on network traffic in order to identify which cell was queried when given a network trace of the query. In this part, every query is done using Tor, meaning every query is encrypted using TLS. The key is to used the meta-data of these queries in order to perform the cell identification attack.

The first step was to capture a large amount of network traffic in order to construct a dataset. This was done using the `tcpdump`, a command-line packet analyzer. We have written a script, `script.sh`, running the following protocol on the virtual machine: For each of the 100 cells, we start a network capture using `tcpdump`, start a query for the cell using the `grid` command and using the Tor option, and once the query is complete we stop the network capture. The trace is then saved in a pcap file. We do this 20 times for each of the 100 cell, leaving us with a total dataset of 2000 pcap files.

The next step was figuring out what meta-data to extract from these traces that would be good features for our classifier. After looking through the pcap files, we have noticed that depending on the cell, the number of packets exchanged by client and server, the number of bytes sent by the server and the total time to execute the query were quite different (and stayed relatively consistent throughout the 20 runs for each cell). This is quite logical considering each cell has a different number of PoI, the server's response to a query will differ in terms of size and time taken depending on the queried cell, even though it is encrypted using TLS. These information seem like features a classifier could leverage in order to identify a cell.

The extracted features for each network traces were the following:

- The number of packets exchanged between client and server for the query.
- The total time elapsed between the query being made by the client and the end of the response by the server.
- The total amount of bytes sent by the server (done by filtering the packets containing strictly more than 54 bytes, which is the size of the ACKs packets sent by the client).
- The mean time elapsed between two packets sent by the server.
- The standard deviation of the time elapsed between two packets sent by the server.

This makes for a total of five features extracted for each of the 2000 pcap files.

To build the classifier, we have used, as suggested, a Random Forest Classifier, as it seemed the most adapted to our task. An interesting property of this classifier is that it takes care of selecting the best features to perform the classification, meaning that if a feature is useless and doesn't have much effect on the label (our cell ID) then the classifier will simply not use it. This means that it is possible to leverage this in order to get feature importance and see what feature is the most useful to our classifier, which we will discuss later. To select the best number of estimators to use in our Random Forest Classifier, we have used 10-fold cross validation, which we will detail in the next section.

B. Evaluation

We have used 10-fold cross validation to evaluate the performance of our classifier for each of the tested number of estimators. The chosen metric to evaluate how good the classifier is doing is accuracy: out of all network traces in the test set, what proportion of the predicted cell for each is correct. We then do the mean accuracy for all 10-folds, and choose the number of estimators that gives us the largest mean accuracy across all its folds. This method of selecting the best parameter gives us the following results, displayed in Table I. The number of parameters doesn't have a huge impact on the accuracy (starting from 80 estimators), we still choose to

Nb of estimators in RFC	**10**	**20**	**50**	**80**	**100**	**200**	**300**
Accuracy of classifier	65.45%	66.4%	68.8%	68.9%	68.8%	69.05%	68.95%

TABLE I: Network's Characteristics

use 200 estimators as it yields the highest mean accuracy : 69.05% of the cells predicted by the classifier for the test set is correct. This is an impressive result, almost 7 times out of 10, the attack is successful in detecting which exact cell out 100 is being queried! Even though the number of classes is very high, and even though the number of features extracted is relatively low (only 5 features to split on), it seems like the server's response depending on the cell has a pretty unique fingerprint that an attacker can leverage.
It would also be possible to generate even more data by running a longer network traffic collection, and by having more than 20 examples for each cell (2000 records is not very large for 100 classes).
In the the next section we'll discuss what factors could make the network traces so unique for each cell, as well as different measures that could prevent an attacker for performing so well.

C. Discussion and Countermeasures

To get a better idea of what information our classifier is taking the most advantage of, we have run a feature important analysis. After having trained our classifier (using 200 estimators), we go through all of the said estimators and get the mean (and standard deviation) feature importance for each of the 5 features we extracted from the network traces.

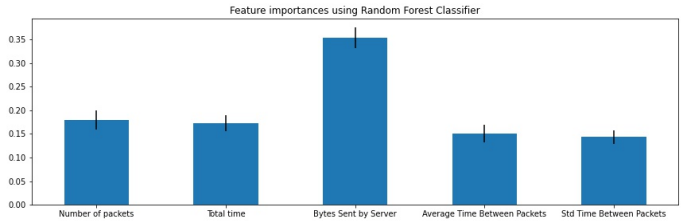

Fig. 3: Feature importance for each of the 5 features used by our classifier

.

The mean feature importances are plotted in Fig. 3. We see that the feature that has the largest feature importance is the "Bytes sent by the server" feature (around 0.35). This is in line with what we had hypothesized. Since the list of all PoI is unique for each cell, the number of bytes (not content!) containing the information for each PoI in the cell seems like the clearest way to uniquely identify the correct cell. The number of bytes sent by the server seems pretty constant throughout the 20 queries for each cell, meaning it's a very valuable information that the classifier takes advantage of. The features importance for the other four features are lower but quite similar (between 0.12 and 0.18), which is non-negligible. This shows us that all 5 features extracted are useful information to our classifier. The combination of all 5 information from the network trace for one query, makes it possibly to uniquely identify the queried cell with very high accuracy.

One simple way to defend against this attack would be to try to mitigate the obvious problem with the server's response: the variable length depending on the queries cell. It is already humanly possible to see the differences in network traces used to differentiate the cell, so it's obvious a classifier will have even less problems taking advantage of it. The simplest solution to this would be to pad the server's response, so every response to a query has the same length. This would make sure that the number of bytes sent by the server is relatively similar regardless of the queried cell ID. This would also have a repercussion on the number of packets exchanged and on the total time, as having a response of the same size regardless of the cell would make those two features consistent across queries. This would likely not have much impact on mean and standard deviation time between packets sent by the server, but this feature alone is not very discriminating with regards to the cell, and would not be very useful to a classifier if only these features were provided. This would however have a effect on the overhead, as we would have to make sure that every response from the server is as long as the longest possible answer. This affect performance quite heavily, both in terms of computation and communication costs, and is a serious hit to usability, as every query would take much longer.

On the other hand it doesn't seem like time related metadata is very valuable information to leverage, so introducing some delay to the message or similar defenses would not have much impact on the attacker (as the number of bytes sent would still be the same), but would impact the utility, as the client would have to suffer various delays.