Semester Project 2022 - Secret Race Strolling: A scalable privacy competition platform

Tom Demont, supervised by Bogdan Kulynych for SPRING, EPFL

June 2022

1 Introduction

SecretStroll is a CS-523 course project where students design a privacy-oriented toy application. The main purpose of the latter is to have a client-server point of interest (PoI) query system for different user locations. At the end of the project, students obtain an application with a client-server architecture going through the Tor network. See Figure 1 for a schematic representation of the application.

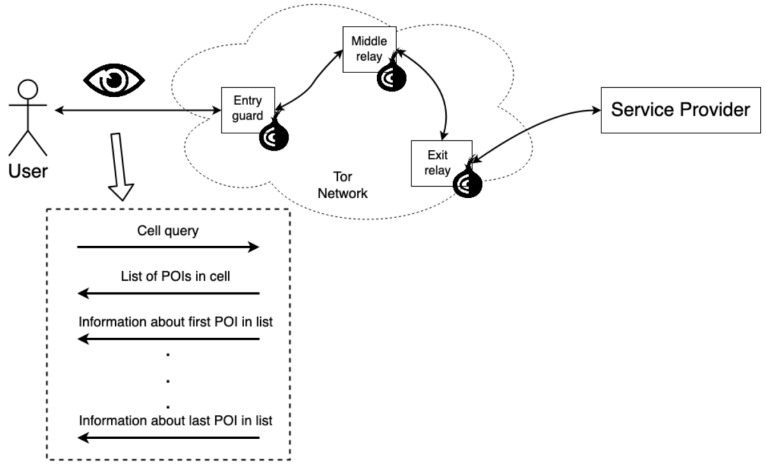

Figure 1: SecretStroll system at the last part of the project

In this context, students should:

- Run the application and collect the generated network trace (in `.pcap` files for example).

- Parse, clean and transform this network trace into feature vectors that associate features to a grid cell id (a discretized location on a map) queried by the client.
- Create a machine learning model to feed these feature vectors to.
- Evaluate the performance of their model.
- Discuss the website fingerprinting features that make the queries distinguishable and propose counter-measures.

Our project stems from this last task. Indeed, to enable students to further explore website fingerprinting and its counter-measures, we aimed to develop a platform where:

1. Students can upload a network trace generated from the client-server system implementing the counter-measures.
2. Students can automatically receive feedback on network wise utility metrics (packet volume and time delay) of their solution by an analysis of the uploaded trace.
3. Matches between teams can automatically be generated and students can download train sets and test sets representing the network trace generated by the Secretstroll implementation of teams they should attack.
4. Students can upload a probability classification for every trace of the test sets they downloaded, and get meaningful metrics that reveal the privacy leakage their classifier can unveil.
5. Students can be ranked in a class-wise leader-board and compare their results to their peers.

The course project name together with the competitive aspect of this addition yield the latter's title: Secret Race Strolling (SRS), recalling the race walking discipline.

Some aspects of the desired platform intersect with other well known machine learning competition platform such as Kaggle [6] or AICrowd [1]. The major difference with our concept is our desire allow student vs. student competition which such platforms do not propose.

As such, the project was focused on the design and implementation of an application fulfilling these goals while meeting constraints on scalability

and code approachability. The core of the contribution can be found on the code repository of the project [12]. The details to run the projects are described in the `README.md` file. This report aims to detail some design decisions, explain the trade-offs and state the development ideas for further enhancement of this project.

2 Design and architecture decisions

2.1 High-level design choices

2.1.1 Timeline

To ensure a clear vision for our implementation, we developed a theoretical timeline of the competition. We envisioned 2 main approaches: the fully interactive one, where students can update their attacks and defence freely at any time during the competition; and the segmented one, where the competition is paced by rounds of attack and defence. The Table 1 summarizes this trade-off.

	Fully interactive	Rounds
Interactivity	Yes, provides responsive insight on privacy leakage	Delayed by rounds, reduces opportunities for improvement
Scalability	Unpredictable and unbounded downloads/uploads	Paced and predictable uploads/downloads
Adapted to CS-523 students	Demanding and potentially overwhelming	Clear schedule

Table 1: Comparison of both timeline approaches

Considering these aspects, we opted for a round-based approach. Through our desire to create a stateful and interactive competition, we prioritised rounds to reduce interference and better allow students to observe their progression. Additionally, students have a limited timeframe for the competition, for which two rounds seems most appropriate. Note that our proposed application is however fully adapted to a higher number of rounds. The deployability of the project, given the limited resources, was also facilitated by this approach. The final timeline is illustrated through Figure 2

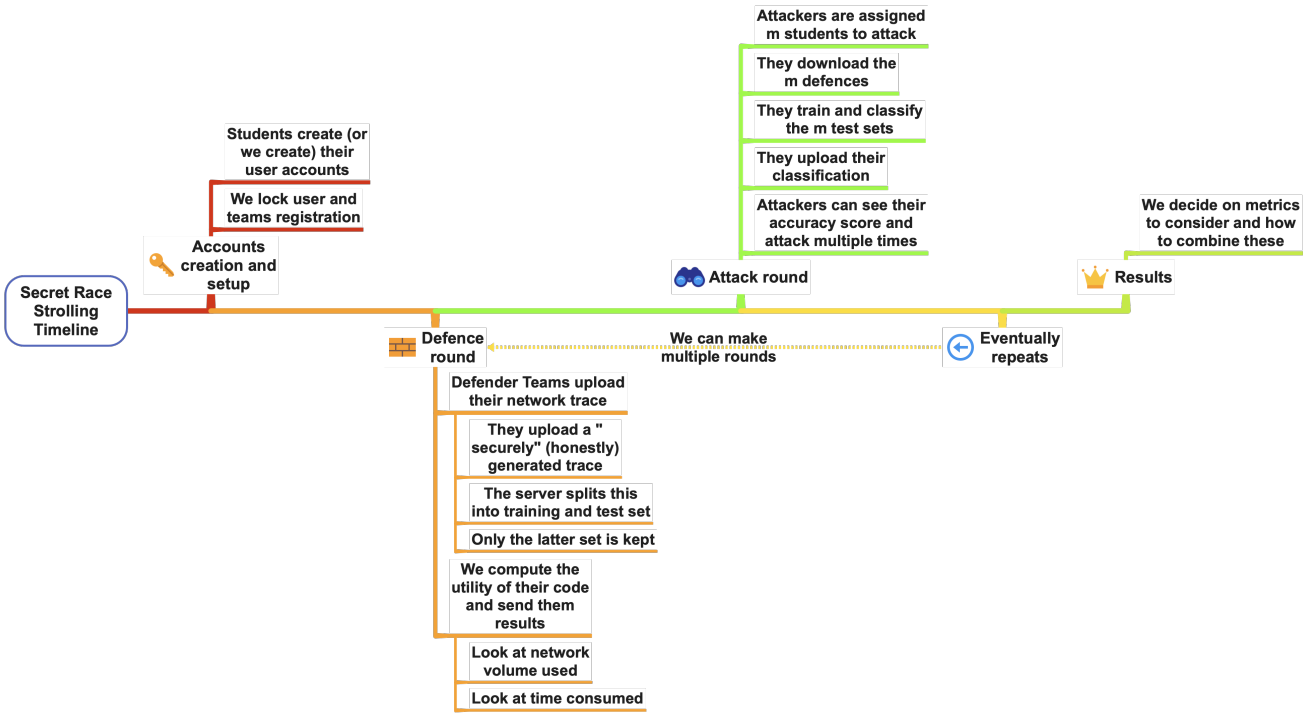

Figure 2: Timeline of SRS

2.1.2 Defence data

In order to evaluate the students' fingerprinting counter-measures, we had to decide between either, asking them to upload a generated trace (close to the one they would use as training/set sets) or ask them to upload their client-server code and run the trace collection on the server side. Table 2 summarizes our thought process for this decision.

As always, there is no free lunch. In our case, the "honest student" assumption is reasonable given that this competition will not influence grades and serves only as a learning tool. We therefore decided to opt for the student side trace generation. Students should upload a csv file with columns as described in Table 3, identifying each exchanged packet over all queries.

The alternative server-side option can of course be set up under a small modification of the appropriate celery tasks. Evolution of this decision is nonetheless totally possible and would require just modification of the celery tasks used to treat student uploaded code for defence phase.

	Student Self generated trace	Server side trace generation
Scalability	Only one collection per team but takes time as collected via real Tor	Collections for all teams but potentially on a faster Tor simulation
Verify correct implementation	Very hard to verify and assert correctness on student's side (could produce passing dummy trace)	We have all freedom to check system's correct implementation
Deployment	Could be deployed in the current form of the code produced by the project and scale correctly	Deploy a Tor network simulation for Secrestroll that supports parallelization and isolation
Fairness	As we cannot prevent cheating, fairness is hard to provide	Providable under determinism of outputs

Table 2: Comparison of both trace collection methods

cell_id	rep	direction_size	timestamp
Location for which PoIs are queried	Query identifier for this location	Signed (+ for upload, - for download) size of the packet in bytes	Packet timestamp

Table 3: Defence columns

2.1.3 Attack data

In order to evaluate students on their classifier, we look at their classification results. Our methodology for this was inspired by that of Kaggle and AICrowd. Notably, we expect uploads under a csv format. Nonetheless, in order to be able to compute more interesting metrics (not only accuracy but also Receiver Operating Characteristic - Area Under the Curve), we decided to ask for the full classifier's output probability for every class. Therefore, instead of having one column with the hard output classification, we ask for 100 columns (the 100 classes of Secretstroll project) holding the classifier's output probability for each class. See Table 4 for the detailed format expected by student for each trace they should classify.

team_id	capture_id	proba_class_1	proba_class_i, $i \in \{1..100\}$
Attacked team	Attacked trace	Classification probability for cell 1	Classification probability for cell i

Table 4: Attack columns

2.1.4 Score and leader-board

Finally, we decided on the metrics to extract from the attack and defence data-frames.

In order to account for utility, we considered all recorded PoI queries in the uploaded trace, and computed the maximum, minimum and mean number of bytes it takes for PoI query resolution. Upload and download traffic were considered separately. These metrics give an indicator of the general network volume overhead. We also kept the maximum volume to notice solutions with specifically high overhead on some queries. For simplicity and readability, we did not use all of these metrics for the final utility score but instead used: $64/(\log_{10} med_{in-volume} * med_{out-volume} * med_{time})$. We used 64 as a magic number and this logarithmic scale to obtain a readable numerical value between 0 and 10 for the default implementation of Secretstroll, while being inversely proportional to the number of bytes and time used by the solution, consistent with a utility metric.

For the attack performance, we initially considered the accuracy. However, the "detector" nature of the classifier inclined us to include more complex metrics accounting for the false positive rate. Therefore, we used the full probability classification output by students to compute the Receiver Operating Characteristic Area Under the Curve (ROC-AUC score)[7] that brings an insightful result on the classifier quality and links to key concepts in machine learning evaluation for privacy purposes. Our attack performance score used for ranking is $10 * roc\_auc\_score$, which also gives a result between 0 and 10 while being proportional to classifier quality metric. For aggregating over all attacks performed by a team, we computed the average of this score for the latest attacks over all matches made in the round. We made the choice to only compute the score over one round, making each round a new opportunity for students to learn from their mistakes without being penalized.

To aggregate these scores and decide on students ranking, we computed a total score by a simple ratio between the two metrics:

$$\frac{64 * \sum_{i \in \{latest\_attack\_\forall\_matches \in round\}} 10 * roc\_auc\_score_i}{(nb\_matches \in round) * \log_{10} med_{in-volume} * med_{out-volume} * med_{time}}$$

2.2 Technical design

2.2.1 Flask and Celery

Early in the project, we decided to implement the system following a web framework. We quickly decided to go with Flask [15], the extremely pop-

ular web server gateway interface. Indeed, this one fits with many of our requirements:

- Based on Python, it comes with its inner readable style and its popularity making it approachable.
- Very light, this microframework [8] is easy to deploy, which is crucial in the case of on-premise deployment that we may want in the context of the CS-523 course.
- It comes with many extension module that allow to easily develop some key aspects (Web Forms, Database interaction, Bootstrap templates rendering, Authentication, Mail support and logging) and facilitates addition of extensions.
- The popularity of the framework yields a very rich and diverse documentation on the web which eases a lot the development and debugging.

Nonetheless, we had to keep in mind the nature of the tasks our server should run: verifying and evaluating students uploaded network trace, and evaluating the scores performed on classification. The collected traces for the Secretstroll project gives files of around 50MB: the deep inspection of such a dataframe can takes a few seconds which does not scale with the expected time response of our web interface, nor with the growth of users. In order to cope with this, we decided to go with a distributed task queue architecture for handling the longer jobs asynchronously. Also, this could allow us to extend our application by developing the server-side trace generation 3.1 as a new asynchronous task.

For this, we considered two popular options: Celery [2] and ZeroMQ [16] frameworks. Table 5 gathers the arguments we had for either proposal.

	Celery	ZeroMQ
Python popular and endorsed module	Yes (3k stars on GitHub)[4]	Yes (19k stars on GitHub)[9]
Integrates well to Flask	Very well integrated, developed in Flask documentation [3]	Some examples exist, but not widely developed

Table 5: Comparison of both distributed tasks queues frameworks

The excellent integration with our web server framework raised Celery as our preference, along with a Redis [10] message broker for its easy deployment (note that the message broker can fluidly be changed, the user

should just change parameters in the environment variable, see the `README` instructions [12]). This smooth integration makes adding of new tasks easy in Python by only requiring a decorator `@celery.task` to generate a task that can be launched asynchronously.

The development and tests were run with the following versions: Python 3.9.13, Flask 2.0.3 and Celery 5.2.6. See the `requirements.txt` file of the coding repository [12] for more details.

2.2.2 Data model

As our project requires to store information about users, we defined a data model containing these.

The timeline discussed in Part 2.1.1 gave us a clear view of the entities involved:

- User: the personal accounts, representing a student or a CS-523 staff
- Team: represents the pair of students that developed Secretstroll together
- Defence: represents an uploaded network trace of a team's implementation of counter-measures. Mostly stores the associated team and utility score performed
- Match: a pair of teams, namely an attacker and a defender. The attacking team should download the train and test set of the defender team and classify the test traces
- Attack: represents an uploaded classification of an attacker team involved in a match

As the relations between entities were pretty clear to define, we decided to go with a relational database model, see Figure 3

To smoothly manage the database entities with our Python code, we made use of the Flask-SQLAlchemy package [5] that provides an access for the Flask app to the toolkit and Object Relational Mapper for Python: SQLAlchemy [13]. One last note is that the usage of SQLAlechmy lets us easily choose a database back-end, just by changing the database url parameter (see the `README` instructions [12]). This has the huge advantage of letting us choose an easily deployable (as SQLite) or more scalable (like PostgreSQL) database without modifying a line of the application code.

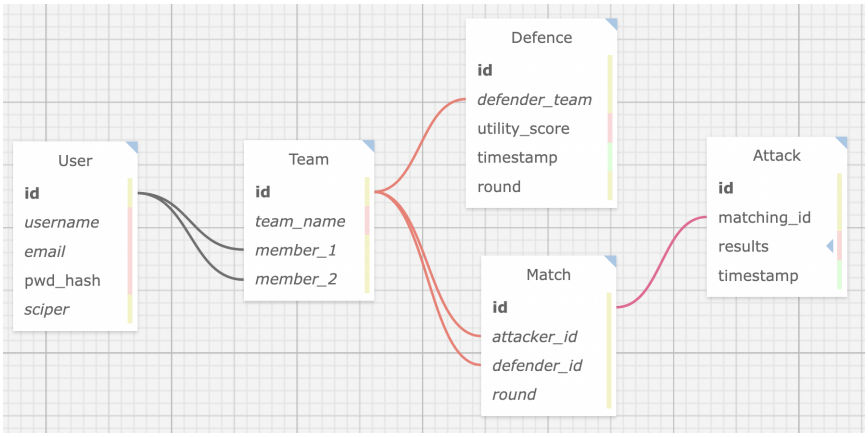

Figure 3: Relational Model of SRS

3 Development continuation and improvements

3.1 Server-side trace generation

As we have seen, the server side trace generation is an extension that could raise the platform to another level and make the competition easier and more enjoyable for students, allowing them to focus on the core of Privacy Enhancing Technologies (PETs) rather than the time consuming trace collection.With this in mind, we have already began steps to integrate a network simulator, Shadow [14], to our application. This solution avoids having to pass through a real Tor network as this is often slow and also to gain control of the network randomness. With this extension, student would therefore not upload their generated csv file but their Python code for client and server, that would be tested for correctness from which we would extract a trace under Shadow network simulation.

We currently met deployment issues in testing the software: build fails for the docker image and in virtual machines on arm64 M1 chip device. We managed to build on an amd64 machine but found out that, in order to generate a tor network, we would have to use TGen [11] network configuration generator tool and make a working install of the tor software. Meeting these issues too late in the project development, we could not integrate it to SRS but we believe it is the best path to manage server side trace collection.

3.2 Automated testing extensions

Another further work could be the development of automated test cases for the different tasks and routes provided by the application. The model methods already have tests in the `test.py` file, we could develop automated tests in the same vein for other components.

4 Conclusion

The currently developed platform proposes a working implementation allowing students having developed the Secretstroll project to match against each other with their fixed client-server code and evaluate the privacy and utility impact of their implementation. This tool could be deployed and used by the future CS-523 team to let students go further and pursue this fascinating project providing an interactive activity on PET implementations and pitfalls. Finally, this Secret Race Strolling is developed with in mind its continuation and improvement that could make it an excellent platform for generic network fingerprinting attack-defence competitions, even handling tor network trace generation on a scalable server side if shadow can be integrated.

References

[1] *AIcrowd*. en. URL: `https://www.aicrowd.com/`.

[2] *Celery - Distributed Task Queue — Celery 5.2.7 documentation*. URL: `https://docs.celeryq.dev/en/stable/index.html`.

[3] *Celery Background Tasks — Flask Documentation (2.1.x)*. URL: `https://flask.palletsprojects.com/en/2.1.x/patterns/celery/`.

[4] *celery/celery*. original-date: 2009-04-24T11:31:24Z. June 2022. URL: `https://github.com/celery/celery`.

[5] *Flask-SQLAlchemy — Flask-SQLAlchemy Documentation (2.x)*. URL: `https://flask-sqlalchemy.palletsprojects.com/en/2.x/`.

[6] *Kaggle: Your Machine Learning and Data Science Community*. en. URL: `https://www.kaggle.com/`.

[7] *Measuring Performance: AUC (AUROC)*. en. Feb. 2019. URL: `https://glassboxmedicine.com/2019/02/23/measuring-performance-auc-auroc/`.

[8] *Microframework*. en. Page Version ID: 1084284183. Apr. 2022. URL: https://en.wikipedia.org/w/index.php?title=Microframework.

[9] *PyZMQ: Python bindings for ØMQ*. original-date: 2010-07-21T07:20:37Z. June 2022. URL: https://github.com/zeromq/pyzmq.

[10] *Redis*. en. URL: https://redis.io/.

[11] *shadow/tgen*. original-date: 2019-02-12T18:40:19Z. June 2022. URL: https://github.com/shadow/tgen.

[12] *spring-epfl/spring22-TomDemont*. en. URL: https://github.com/spring-epfl/spring22-TomDemont.

[13] *SQLAlchemy - The Database Toolkit for Python*. URL: https://www.sqlalchemy.org/.

[14] *The Shadow Network Simulator*. URL: https://shadow.github.io/.

[15] *Welcome to Flask — Flask Documentation (2.1.x)*. URL: https://flask.palletsprojects.com/en/2.1.x/.

[16] *ZeroMQ*. URL: https://zeromq.org/.