Student Privacy Protection in Online Classrooms

Pierre Thévenet
SPRING Lab
Supervised by Wouter Lueks and professor Carmela Troncoso

January 10, 2020

1 Introduction

Massive open online courses (MOOCs) became popular among students and educators, by providing students access to open, interactive and distant teachings . MOOCs platforms collect large amount of data on students, enabling teachers to gain useful feedback on their courses. Student activity feedbacks, such as answers success rate or response times, give the opportunity for teachers to improve the learning experience. However this collection of data brings significant privacy concerns. The information collected on students may be very sensitive, and could be misused. For instance, the platform could leak results from a personality test, which could encourage discrimination or bullying of students.

Our goal is to prevent the data collected by the platform to be personally identifiable. Doing so would protect the students' privacy while preserving the utility of the data for educators. Network-level and application-level anonymity are needed to prevent student activities on the platform to be identifiable. In this contribution, we focus on ensuring network-level unlinkability, to provide anonymous communications between students and the platform. For application level unlinkability, anonymous credentials [13] can be used to protect students' identity, while enabling access control on the platform.

In this work, we want to modify an existing online classroom platform, FROG [1], to let students use the platform anonymously. We seek a solution that is easy to use, with no software installation or configuration needed from students. Moreover this solution should be portable to other platforms.

We built a prototype solution on top of Lightnion [11], a JavaScript library that provides anonymity at the network layer for simple HTTP re-

quests. Lightnion does not support the WebSocket protocol, which is used by FROG. Hence, we extended Lightnion to support WebSockets so that all communications between students and FROG could be anonymous.

2 Background

2.1 Tor

In this work, we extended the Lightnion library, which uses Tor [6] to provide network anonymity.

Tor enables anonymous communications at the network level between a Tor client and an application server, running on top of TCP. To communicate with an application server anonymously, an application has to implement a Tor client. The Tor client exposes a TCP socket to the application, and routes this socket's messages through the Tor network, implementing onion routing [8] to provide anonymity. A Tor client randomly selects three Tor nodes, forming a circuit from the first to the last. It then establishes a TLS session with the first node, and uses this session to establish another one with the second node, by encapsulating the second session into the first. It proceeds similarly for the third node. At the end of the third node is exposed a real TCP session, between the third node and the application server. This routing technique makes it difficult to determine both ends of the logical connection by a passive adversary.

Tor protects against network surveillance and traffic analysis but does not protect against a global passive adversary capable of sniffing the client's and server's network. Tor also protects from successive connection linkability. While constructing the circuit, Tor nodes are chosen randomly, therefore successive connections cannot be linked to the same client from the application server.

2.2 Lightnion

In this section we explain what Lightnion [11] is and what we extended it with.

Lightnion enables simple browsers to access webpages through the Tor network. It is a JavaScript library that can be shipped in web applications wanting to provide anonymity for their users. This way, users do not need to install the Tor browser or configure their browser's settings to gain anonymity.

The Lightnion library does not connect directly to the Tor network, as browsers do not allow making plain TCP sockets programmatically. Tor clients, however, need to open TCP connections to Tor nodes to follow the protocol. To circumvent this limitation, the Lightnion library implements a modified Tor client that runs on the users' browser. Since this client cannot directly connect to Tor nodes, its messages are encapsulated over a WebSocket connection to a Lightnion proxy. This Lightnion proxy then receives the WebSocket messages, translates them from WebSocket to TCP, and forwards them to Tor nodes. When the Lightnion proxy receives messages from Tor nodes, it proceeds oppositely. Messages from Tor nodes are encapsulated into the WebSocket connection, forwarded to the Lightnion client, that deciphers and delivers them to the web application. The Lightnion proxy is therefore seen as a Tor client by the other Tor nodes. However this proxy does not hold the cryptographic keys needed to decipher the Tor messages it receives, so the confidentiality of the user's connection is kept the same.

To web applications running on the browser, the Lightnion library offers a TCP-like socket, enabling the implementation of anonymized TCP applications. It is already possible to fetch HTTP webpages using Lightnion, but more complex web applications often requires WebSockets. Consequently, in this work we implemented WebSockets over Lightnion.

2.3 WebSocket

The WebSocket protocol is a full-duplex communication protocol working on top of TCP. WebSockets are message-oriented, and provide binary and text communications, with support for fragmentation. The protocol is supported by the major web browsers (since Chrome v16, Firefox v11, ... [5]) and standardized by the RFC 6455 [7]. Full-duplex communications were previously emulated by HTTP streaming, long polling [10], or other mechanisms.

To our knowledge, there exist no WebSocket protocol implementation in JavaScript for the browser. We therefore implemented the WebSocket protocol in JavaScript, to be used in the browser and emulating the standard WebSocket.

3 Setting

To introduce the privacy properties we seek to implement, we consider the scenario of an online classroom, see Figure 1. Users (students) use a web application hosted on an endpoint (the platform), to perform tasks written and gathered by an administrator (the professor). Task results contains

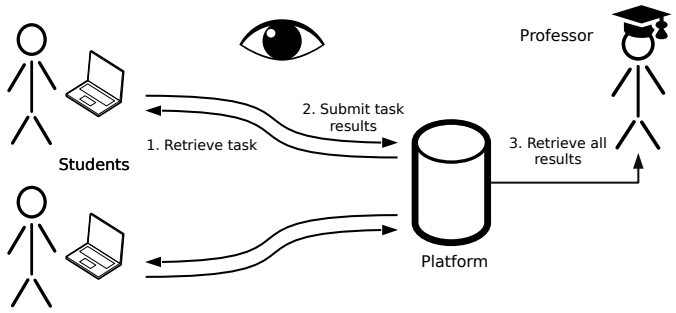

Figure 1: High level setting. Students access the platform, retrieve tasks and submit task results. The professor then gather all results from students. Traffic between the students and the platform can be anonymized using Lightnion.

user data (such as questionnaire results) along with metadata linked to the user activity (access times, task duration, network address, ...). The administrator creates tasks to be done by the users, and can retrieve the task results from the endpoint.

3.1 Privacy Goals

Sender anonymity [12]. Users connecting to the endpoint should be anonymous. Explicitly, the endpoint, administrator and other users should not be able to identify a user from the set of all possible users. If the access to the endpoint is limited, we want authorized users to be anonymous in the set of authorized users.

Activity unlinkability [12]. Successive connections from users should be unlinkable. Explicitly, if a user connects successively to the endpoint, the endpoint or administrator should not be able to link these two connections to the same user.

3.2 Privacy Non-goals

If the professor has access to the local network used by students, he may perform traffic analysis to deanonymize students. For example, if a single

student accesses the platform and submits a task, then the professor could deanonymize this student. Having access to the local network of students, the professor could filter out the only student connected to the platform and infer that this is the student having submitted the task. This simple example shows that deanonymization of students is possible if the professor has access to the local network of students. We therefore assume that the professor does not inspect the traffic of the students. To avoid this assumption, anonymous networks with stronger guarantees could be used, for example Atom [9].

Moreover, we do not seek to avoid privacy leaks performed by users themselves. We do not prevent a user from leaking information on data it submits, for example by giving its real name when completing a task.

3.3 Threat model

We consider a practical threat model that can be applied to a real world scenario. It puts little trust on the administrator and endpoint.

Administrator as a Covert Adversary. We assume the administrator is covert, that is, he may behave malisciously if the chance of getting detected is very low. This assumption implies that the administrator will always point the users to use a privacy-respecting version of the web application, e.g. by pointing to a deployment of our solution.

Endpoint as a Covert Adversary. Similarly, we assume the endpoint is a covert adversary, that may be audited by administrators or users.

4 System

In order to anonymise the users of FROG, we need to ensure that all web communications between them and the FROG platform are anonymous. Specifically, FROG uses both HTTP and WebSockets to communicate with its users. Lightnion [11] already provides a way to anonymise HTTP connections, with the convenience we aim to achieve. So we based our solution onto the previous work done in Lightnion [11]. Using Lightnion for an application-level protocol requires a JavaScript implementation of that protocol, for the browser. Hence, we added support for WebSocket redirection to Lightnion.

Our solution completes the Lightnion system described in the original Lightnion proposal [11]. We added a new party to the system because of

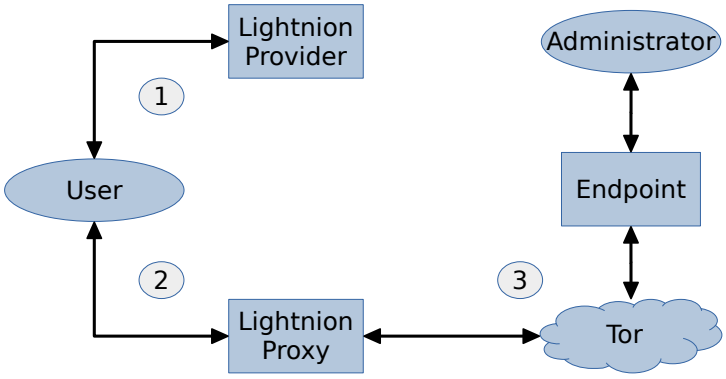

Figure 2: Lightnion System for untrusted endpoint.

the different threat model: the Lightnion provider. The Lightnion provider is a server hosting the Lightnion client JavaScript code. The user's browser should connect first to the Lightnion provider, retrieving the client code. In the original Lightnion proposal [11], the provider and endpoint are the same entity, however, in this model, the platform is not trusted. Therefore we shifted the trust assumptions to the new entity.

We describe our solution, see Figure 2. Let us suppose a user has been requested to visit the web application hosted at the endpoint, via the Lightnion provider's URL.

First (1), the user enters the URL of the Lightnion provider on its browser, and retrieves the webpage with the bundled Lightnion client code. Second (2), the user's browser establishes a connection to the Lightnion proxy. Finally (3), the user connects to the endpoint by specifying its URL in the webpage. In this step, the Lightnion client hosted on the user's browser setups a Tor circuit, through the Lightnion proxy, by choosing random Tor nodes. This circuit is then used to tunnel HTTP requests and WebSocket messages to the endpoint.

Trust assumptions. Having added two parties, we need to clarify the tradeoffs that were made in order to offer the privacy and convenience that we are looking for.

The Lightnion provider and the Lightnion proxy should not collude with the administrator nor the endpoint. In particular, this is necessary to avoid traffic correlation attacks. The Lightnion provider can either be trusted to give the correct Lightnion client code or untrusted if there is a way for clients

to verify the code received from it.

In a practical setting, these assumptions are realizable if, for example, the Lightnion provider and the Lightnion proxy are hosted by a different entity than the teaching team and the platform host.

Privacy properties. We argue that the privacy properties hold.

User Anonymity. This property is provided by Tor. The use of the Lightnion proxy does not weaken the anonymity of the users [11]. The Lightnion proxy knows the set of users that are connected to him, but nothing about the destination or contents of the packets it receives. It's role is analogous to the entry node of a Tor circuit, because it knows its connected users and their next-hop in the circuit, but nothing more. The communications between the user and the provider only leak the access times of the user, because they download the same webpage and code every time.

Activity Unlinkability. The Tor exit node chosen by the Lightnion client is random and changes at each connection. The endpoint (or administrator) sees the exit nodes used by all the users, and may link connections if the exit node does not change for a user between two connections.

4.1 WebSocket Redirection

To provide a Tor tunnelling of WebSocket messages, we implemented the WebSocket protocol in JavaScript. To provide a Tor tunnelling of WebSocket messages, we had to implement the WebSocket protocol in JavaScript. We implemented it on top of the Lightnion TCP socket, but it should be usable in other conditions, in particular with Node.js raw TCP sockets.

The websocket library implements a good part of the standard, aiming to be a drop-in replacement for usual WebSockets clients:

```
<script src="lws.bundle.js"></script>
<script>
    let url = new URL("ws://echo.websocket.org");

    // replace this following line
    let ws = new WebSocket(url, protocols = []);

    // by this line
    let ws = new LWS(url, protocols = [], lightnionHost = "
        127.0.0.1", lightnionPort = 4990);
</script>
```

The Lightnion WebSocket library was developed respecting the standard as closely as possible. The library includes tests to verify the correctness of the implementation, namely on the protocol and the browser interface. We did not implement subprotocols [7, s. 1.9] nor extensions [7, s. 9], as there are not needed for our current application. However our library can be extended to support them in the future.

5 Integration and Evaluation

5.1 Practical integration

As a proof of concept and for future use, we tried to implement the websocket redirection into an educational web application named FROG [1]. Description of all steps taken to redirect FROG's websocket are detailed in the project repository [3].

FROG uses the meteor framework [2] for deployment, which uses its own WebSocket when serving a web application to users. To redirect this WebSocket, modification to the framework itself is required. We tried to patch the framework, unfortunately dependencies issues prevented us to fully implement our solution.

In summary, we successfully implemented WebSocket redirection into FROG, but not into its deployment framework. We believe the recipe to deploy our solution is valid.

5.2 Experimental setup

To evaluate the JavaScript WebSocket redirection through Lightnion, we compared it to a non-private browser-provided WebSocket, as well as to a browser-provided WebSocket proxied through Tor. Moreover, we compared using our redirected WebSocket implementation to using raw TCP sockets redirected through Tor, that Lightnion already provides. We will denote by "WS" the use of the standard WebSocket client, "WS+Tor" the use of the standard WebSocket client proxied through a Tor network, "LTCP" the use of the Lightnion redirected TCP socket client, and "LWS" the use of the Lightnion redirected WebSocket client. We used Firefox 71.0 (64-bit) for the evaluation.

The experimental setup consists of four entities, as shown on Figure 3: a browser, a Lightnion proxy, a local Tor network and an endpoint. The Lightnion proxy will proxy the Lightnion WebSockets and TCP communications through the Tor test network. The client browser will initiate the

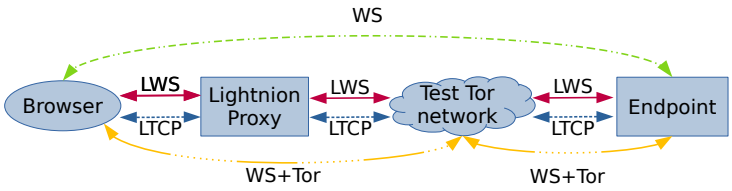

Figure 3: Experimental Setup

connections to the endpoint, using Lightnion WebSockets, Lightnion TCP sockets or browser-provided WebSockets, that are proxied or not through the Tor test network. The local Tor network is constructed using chutney, and runs 8 Tor nodes (with at least 2 guards and 5 exit nodes). The endpoint implements several protocols, above the TCP and WebSocket sockets.

We measured four different quantities in order to evaluate our implementation. First, the round trip time, computed using echo WebSocket or TCP servers as the endpoint. In the browser, it is the time between the sending a message and the reception of the reply. Once the socket connection opened, we successively sent and received 10000 frames. Second, the connection establishment time, that is, the time taken by the browser to open a socket connection to the endpoint. Third, the time to first message, which is the time taken to open a connection, have the browser send a message and have the server receive it. Both previous tests were measured by opening 100 successive socket connections to the endpoint. Finally, we computed throughput measurements, having first the browser upload to the endpoint, and then the endpoint upload to the browser, at different times. Throughput was computed by sending 10MB of data with messages of 1kB.

All measurements where computed using the browser's internal clock, except for the time to first message that implied using the endpoint clock. We measured the time shift between both clocks to be under 2 milliseconds. The browser clock has a precision of one millisecond.

5.3 Evaluation results

As the numbers in Table 1 show, our solution does not compete performance-wise with using a Tor enabled browser. The use of Lightnion induces overhead, by having the browser perform Tor cryptographic operations, protocol encapsulation, and adding network latency by introducing a proxy. More-

	WS	WS+Tor	LTCP	LWS
Round trip time (ms)	0.2476 (0.004588)	0.2272 (0.006583)	23.65 (0.09555)	24.99 (0.09880)
Connection establishment time (ms)	6.110 (0.08750)	6.440 (0.07563)	334.3 (4.430)	480.7 (5.615)
Time to first message (ms)	5.490 (0.2259)	5.930 (0.1373)	312.7 (2.665)	471.4 (4.329)
Download throughput (MB/s)	32.89	28.17	1.484 *	0.5636 *
Upload throughput (MB/s)	17.61	19.57	0.7762 *	0.3758 *

Table 1: Experimental results of evaluation, measuring the mean and standard error (in parentheses). * Due to high CPU utilization during the throughput experiments, results may be worse than they are in practice.

over, our implementation of the WebSocket protocol introduces significant performance losses compared to using plain Lightnion TCP socket.

During throughput tests, we observed high CPU utilization with intervals of 100% when evaluating our WebSocket implementation and Lightnion TCP sockets. Therefore numbers in Table 1 may be worse than they are in practice. When testing for other measurements, we did not notice such behavior and CPU utilization was low. This leads us to think that our solution is fit for low-throughput applications.

6 Conclusion

WebSockets became a ubiquitous tool in web applications, and supporting them creates more opportunities to use Lightnion. We now hope to cover most web application dependencies, by providing both HTTP and WebSocket redirection through Tor. Our work also shows that Lightnion can be extended for arbitrary application protocols running on top of TCP. We believe our system, once production ready, could be deployed easily in a real-life scenario, by taking example of our deployment of FROG.

Our implementation does incur large performance tradeoffs, but we believe it is still useful for low-throughput applications, like FROG. We hope

the performance can be improved, nevertheless, our solution is by design slower than plain browser-provided WebSockets.

Few important missing features would be necessary in order to improve our prototype. In particular, WebSocket subprotocols and extensions are lacking. Moreover, secure WebSocket (WebSocket over TLS), are not currently available. They could be easily implemented by taking example on the support for HTTPs by Lightnion, but requires the Lightnion proxy to provide secure WebSockets support.

References

[1] FROG. `https://chilifrog.ch/`.

[2] Meteor. `https://www.meteor.com/`.

[3] Project Repository. `https://github.com/spring-epfl/fall19-PierreThevenet`.

[4] Service Worker API. `https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API`.

[5] The WebSocket API (WebSockets). `https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API`.

[6] Roger Dingledine, Nick Mathewson, and Paul F. Syverson. Tor: The Second-Generation Onion Router. In *Proceedings of the 13th USENIX Security Symposium, August 9-13, 2004, San Diego, CA, USA*, pages 303–320, 2004.

[7] I. Fette and A. Melnikov. The WebSocket Protocol. RFC 6455, IETF, December 2011.

[8] David Goldschlag, Michael Reed, and Paul Syverson. Onion routing for anonymous and private internet connections. Technical report, Naval Research Lab Washington DC Center for High Assurance Computing Systems, 1999.

[9] Albert Kwon, Henry Corrigan-Gibbs, Srinivas Devadas, and Bryan Ford. Atom: Horizontally scaling strong anonymity. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 406–422. ACM, 2017.

[10] S. Loreto, P. Saint-Andre, S. Salsano, and G. Wilkins. Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP. RFC 6202, IETF, April 2011.

[11] Wouter Lueks, Matthieu Daumas, and Carmela Troncoso. Lightnion: seamless anonymous communication from any web browser. In *Workshop on Measurements, Attacks and Defenses for the Web (MADWeb) 2019*, 2019.

[12] Andreas Pfitzmann and Marit Hansen. A terminology for talking about privacy by data minimization: Anonymity, Unlinkability, Undetectability, Unobservability, Pseudonymity, and Identity Management. `http://dud.inf.tu-dresden.de/literatur/Anon\_Terminology\_v0.34.pdf`, August 2010. v0.34.

[13] João Pedro Pinto Silveira. A browser-based anonymous questionnaire system with user-controlled linkability. dec 2019.

A HTTP Redirection

A current limitation of the HTTP redirection provided by Lightnion is that it does not follow browser's fetches. For example, if a user clicks on a link on a webpage received with Lightnion, the browser will follow the link and fetch the linked page directly, without using Lightnion. This is a threat to the privacy of the users of the webpage, since they may think being protected.

To redirect HTTP fetches, we implemented a prototype of an imperfect solution. A trivial solution would be for the web application to avoid fetches to other HTTP resources, but we believe this to be a too large constraint to support. We found a partial solution that relieves a part of this constraint from the application.

It consists in a ServiceWorker [4], served by the Lightnion provider, that intercepts HTTPs fetch requests. Those requests are first blocked by the ServiceWorker, preventing the user from being exposed to the target domain. The requests are then modified to be Tor encapsulated and redirected to the Lightnion proxy. Responses to these requests are then deciphered and processed as normal responses.

Using this solution, some constraints are still imposed on the web application. ServiceWorkers can only intercept fetch that are to the same origin. In this case the origin is the Lightnion provider, resources pointed from the webpage using relative references are considered same origin by

the browser. Hence resources fetch from other websites, such as clicking on a link to another website, cannot be redirected. Cooperation is still needed from the platform to not include those resources into the application. An unexplored solution to this problem would be to process all responses and convert absolute references to appear as same origin.