Lightnion: seamless anonymous communication from any web browser

Wouter Lueks
SPRING lab, EPFL
wouter.lueks@epfl.ch

Matthieu Daumas
matthieu@daumas.me

Carmela Troncoso
SPRING lab, EPFL
carmela.troncoso@epfl.ch

***Abstract*—Privacy-enhancing technologies often rely on anonymous communication systems to hide users' identities on the network layer. Such systems are difficult to integrate. Privacy-friendly applications therefore rely on other tools such as the Tor Browser to provide anonymous communication. This approach places a burden on users: They must install and use these tools correctly. We present Lightnion, an anonymous communication library that can be easily integrated into web applications to enable seamless network anonymity for its users. Lightnion uses the existing Tor network. It consists of a Javascript library that runs in a user's web browser, and a proxy that facilitates the communication between the browser and the Tor network. In this short paper, we present Lightnion's architecture, its threat model, use cases, and the first performance measurements of our research prototype.**

I. Introduction

Anonymous communication networks (ACNs) are essential for privacy-enhancing technologies to maintain user anonymity. For example, messaging systems use ACNs to hide communication metadata from network observers. Anonymous authentication systems use ACNs to hide the user's network identity from service providers.

Over the years, application have integrated privacy-enhancing technologies and cryptographic solutions to provide seamless security for users. Messaging applications such as Signal and Whatsapp use strong cryptography to provide end-to-end encryption without any user involvement. Similarly, the collaborative editing platform CryptPad[1] integrates strong cryptography to ensure that CryptPad servers cannot read the documents its users edit.

Network anonymity, on the other hand, still relies on external tools such as the Tor Browser. If users do not use such a tool, their privacy may be affected. For example, if users do not use an anonymous browsing tool, the CryptPad server learns which users edit which documents (even though the server cannot see the content). Similarly, if a secure email client requests GPG keys from a key server before encrypting an email, that key server learns who is communicating with whom based on the requested keys.

[1] https://cryptpad.fr

Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb) 2019
24 February 2019, San Diego, CA, USA
ISBN 1-891562-60-6
https://dx.doi.org/10.14722/madweb.2019.23xxx
www.ndss-symposium.org

Using Tor to browse the internet used to be difficult [1], [7], however, with the introduction of the Tor Browser, usability of Tor has improved a lot [4], [6]. The Brave browser[2] might make using Tor even easier. However, using Tor is not *effortless* for ordinary users. They have to install software, and use it correctly. Application providers might therefore not be willing to ask or require users to use it.

In this paper we present Lightnion, an anonymous communication library that can be seamlessly integrated into web applications to enable them to make anonymous web requests from the user's browser. Lightnion requires no special actions from users. Opening the web application in a normal browser is sufficient.

Lightnion is not always an alternative to the Tor Browser. Lightnion enhances a user's anonymity when using *trusted* web applications. Unlike the Tor Browser, Lightnion cannot protect against malicious web sites. However, there are many scenarios where the web application is not the adversary and users require anonymity only for particular actions or against particular third parties.

Consider the secure collaborative editing platform CryptPad. Users trust the software provided by CryptPad. At the same time, they do not necessarily want the CryptPad server to know which files they edit: users require anonymity for the action of editing the file. To achieve this anonymity, the CryptPad application can use the Lightnion library to perform document edits via an anonymous channel. Similarly, the secure webmail client can provide anonymity against the third-party key server by making all requests for recipients' GPG keys using the Lightnion library. The key server can then no longer relate key requests to real users.

Lightnion uses the existing Tor network to provide anonymity for users. It consists of a small Javascript library that trusted providers can embed in their websites. Websites can then take make anonymous connections to untrusted servers using the Lightnion library. To do so, the library connects to the Tor network via an untrusted proxy, and then sets up an anonymous connection via the Tor network to the untrusted server.

In this work, we make the following contributions:

✓ We analyze the trust assumptions required for an anonymous communication library in the browser.

[2] https://www.brave.com

✓ We apply this analysis to derive a minimal Lightnion library that uses the Tor network and protocols.

✓ We constructed a prototype implementation of Lightnion and analyze its performance.

✓ We present use cases that can benefit from Lightnion.

II. Background

Tor [2] is an overlay anonymity network consisting of around 6000 volunteer Tor nodes, or onion routers. It is the most widely used anonymous communication network. At any point in time, millions of users use Tor.[3]

Users communicate anonymously with servers by routing their traffic through three Tor nodes. This sequence of three nodes is called a *path*, the nodes are called the guard, the middle, and the exit node respectively. Each node on the path only knows the identity of the two adjacent nodes. As a result, none of the Tor nodes know both the user and the destination server. Traffic between the Tor client, running on the user's machine, and nodes in the Tor network is onion encrypted. Each subsequent node along the path removes one layer of encryption, until finally the exit node removes the last layer and forwards the traffic to the destination server.

The Tor network consists of many Tor nodes, each with different capabilities and available bandwidth. A group of nine Tor directory authorities periodically publish a *consensus* of all the nodes in the network. Tor nodes themselves publish *descriptors* describing their keys and other properties. Tor clients use the consensus and descriptors to select paths and to authenticate Tor nodes.

To create an anonymous connection to a destination server, the Tor client proceeds as follows.

1) The client retrieves the latest consensus and verifies the signatures by the trusted directory authorities. It also retrieves descriptors for many Tor nodes.
2) The client uses the consensus information to compute a path consisting of three nodes: the guard, middle and exit nodes.
3) The client makes a TLS connection, called a link, to the guard node. The client authenticates the guard based on its descriptors and derives a shared key. The guard and the client use this shared symmetric key to encrypt subsequent traffic between them. The client starts building a circuit.
4) The client then requests the guard node to extend the circuit to the middle node. The client authenticates the middle node using information from the node's descriptor and derives a shared key. The client and the middle node use this key to encrypt traffic between them encrypt their traffic.
5) The client requests the middle node to extend the circuit to the exit node. The client then authenticates the exit node and derives a symmetric key as before.
6) Finally, the client requests the exit node to open a TCP connection to the target server.

III. System

Figure 1 shows a high-level overview of the Lightnion architecture. Lightnion consists of two parts. The first part is a Javascript library that runs inside the user's browser. Websites can use this library to make anonymous web requests via the Tor network. To do so, the Lightnion Javascript client connects to a Lightnion proxy, the second Lightnion component, via a Websocket connection. The proxy relays Tor cells it receives over the websocket connection to a corresponding TLS connection that it maintains with guards in the Tor network.

The Lightnion Javascript library authenticates the nodes on the circuit, deriving the correct encryption keys for the symmetric encryption channel, and handles all the necessary cryptographic operations. The proxy simply relays traffic to the corresponding Tor nodes.

A. Parties and trust assumptions

The Lightnion Javascript library is provided by the website that the user visits. We call this party the *service provider*. We assume that the service provider is semi-trusted. In particular, we assume that it will not intentionally try to send malicious code to the user. This trust assumption can be weakened if users can verify the code they receive before running it.

The web application may need to communicate anonymously with one or more *end points*. These end points are untrusted and may try to identify the user. The goal of Lightnion is to ensure user anonymity with respect to this end point. If the end point and the service provider coincide, we still assume that the service provider does not send malicious code to the user. In the case of CryptPad, the end point is the CryptPad server itself, whereas for the secure webmail client the endpoint is the key server.

The *Lightnion proxy* facilitates communication between the Lightnion Javascript client and the Tor network. We assume that the proxy is available, but we do not trust it for privacy. There can be many such proxies. Ideally, the Lightnion proxies would be co-located with Tor nodes.

A non-collusion assumption. We assume that the Lightnion proxy does not collude with the end point. If they collude, they can perform a time-correlation attack to deanonimize users. This assumption is the same as Tor's assumption that the guard node and the destination server do not collude.

IV. Design and architecture of Lightnion

In this section we describe the design and architecture of Lightnion.

A. A simple Tor client

Lightnion aims to implement an as simple as possible, but fully functional, Tor client that can run in the user's browser. Unfortunately, the full Tor client is quite complex. Users use Tor for a large variety of purposes, e.g., to browse websites, to share files, to send email, to use instant messaging services, or to visit hidden services (websites that are not accessible via the normal Internet).

[3]https://metrics.torproject.org/userstats-relay-country.html

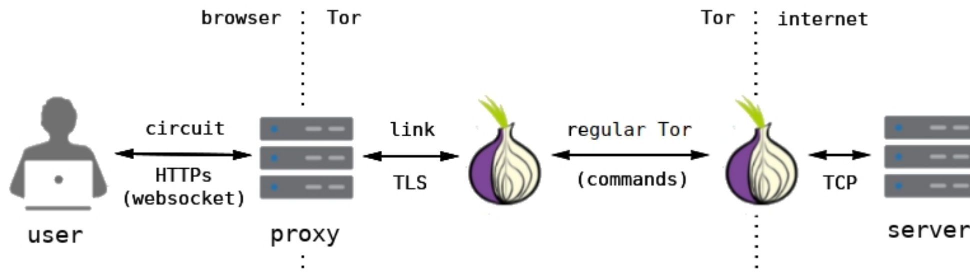

Fig. 1. High-level architecture of Lightnion. The Lightnion Javascript library runs in the user's browser. To communicate anonymously with an untrusted end point (the server at the right) the Lightnion Javascript library connects to it via the Tor network. The Lightnion proxy translates between a Websocket on the user's side and a TLS link on the Tor side.

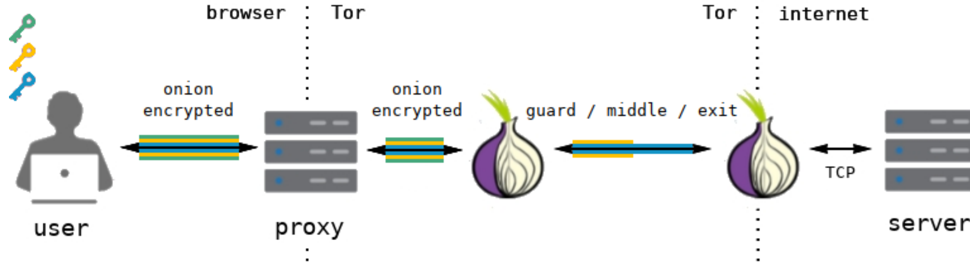

Fig. 2. Interaction of Lightnion Javascript client running in the user's browser and the Tor network via the Lightnion proxy. Note that the Lightnion proxy only sees onion encrypted traffic, it can therefore neither read nor modify it. The Lightnion Javascript client holds the keys with which Tor traffic is onion encrypted.

Moreover, these different applications have widely different characteristics. E-mail requires low bandwidth and can tolerate a lot of latency, and messaging is low bandwidth and tolerates some latency. Both require only a few TCP connections. On the other hand, file sharing and web browsing, especially when visiting streaming websites, require a large number of concurrent TCP connections, some of which require high bandwidth and low latency.

The Tor client has been optimized to work well in all of these scenarios. It supports a large number of possibly long-term and high-bandwidth connections, and in addition to the normal protocols to communicate anonymously with existing servers it also supports protocols to enable hidden services. All these features complicate the Tor client.

We envision Lightnion for scenarios, see Section VI, that do *not* require this complexity. They do not use hidden services, do not make many simultaneous connections, nor do the connections that they do make require high bandwidth or very low latency. This observation has given us the freedom to implement a lean Tor client that offers only essential functionality: to setup one TCP connection that can be used to transfer small amounts of data anonymously.

B. Circumventing browser limitations

Implementing a simple Tor client directly in the browser is challenging. To connect to the Tor network, a Tor client opens a TLS connection to the guard over TCP/IP. However, browser scripts can only create HTTP and Websocket connections. Pure TCP/IP connections are not allowed.[4]

[4]There exist some hacks such that circumvent these restrictions on some browsers, for example using Flash plugins or Java applets. However, none of these are portable or supported in a wide range of browsers. WebRTC is supported by most browsers, but we prefer Websockets as they are simpler.

Instead, in this work we use a simple, untrusted Lightion proxy that translates between a protocol that is available to the Javascript Lightnion client, such as HTTP or Websockets, and the TLS connection to the Tor nodes. Needing this proxy is unfortunate. Ideally, this proxy would be unnecessary because the Tor nodes themselves or pluggable transports they host provide the required functionality.

Note that the TLS connections between the Lightnion proxy and the Tor nodes terminate in the proxy. As a result, the Javascript Lightnion client does not need to simulate a TLS client in software.

C. Setting up a Tor circuit

In Section II we identified six steps that a normal Tor client must perform to set up an anonymous communication channel via the Tor network. The Lightnion Javascript client retrieves and validates the consensus and Tor node descriptors, and selects a path. Then, it requests the Lightnion proxy to open a TLS connection to the guard, the first node in the path. See also Figure 1.

Thereafter, the Lightnion Javascript client performs a handshake with the successive guard, middle and exit nodes. These handshakes authenticates the nodes and derives symmetric encryption keys that are then used to onion encrypt traffic between the Lightnion Javascript client and the exit node. Note that the handshake authenticates the connection to the guard despite the fact that the proxy is a man in the middle for the TLS connection.

Older Tor clients used to use a fast key handshake protocol for the guard node because it already authenticated it on the TLS layer. Obviously, the Lightnion client cannot use this optimization because the TLS channel terminates in the Lightnion proxy, not in the client itself.

```
// create a channel through the proxy
lln.open('proxy.example.net', 4990, function(channel)
{
    // Callback interface (skip intermediate states)
    if (lln.state != lln.state.success)
        return

    // Handle response of request
    var handler = function(data) {...};

    // Send HTTP GET request to api.ipify.org
    tcp = lln.stream.tcp(channel,
            'api.ipify.org', 80, handler)
    tcp.send('GET / HTTP/1.1\r\n' +
            'Host: api.ipify.org\r\n\r\n')
})
```

Listing 1: Sample usage of the Lightnion browser script to make an HTTP request to an external service. We omitted the handling of the response for brevity.

After authenticating the three nodes, the client has established a layered encrypted channel from the users browser, via the proxy, to the exit node. See Figure 2. Note that the Lightnion proxy only sees encrypted traffic. Finally, the Javascript client requests the exit node to open a TCP connection to the destination server. 1

1 WL: Sending traffic is actually a bit tricky. As we need to format our own HTTP/S requests.

V. Evaluation

We created a research Lightnion prototype. We are currently working on turning this prototype into a robust and stable implementation that can be used in a large variety of settings. We report here on the initial research prototype and its performance.

A. Implementation

We first implemented a minimal Tor client in Python to identify the different components needed. We then reimplemented this client in two parts: the Javascript Lightnion client and a Python Lightnion proxy. The Lightnion client uses the Javascript libraries sjcl[5] and tweetnacl-js[6] to perform the cryptographic operations.

While we have working code for each of the six steps described in Section II, the current Lightnion Javascript library does not integrate the code to verify Tor's consensus and node descriptors, nor does it pick paths itself. Instead, it relies on the proxy to provide a path. We hope to change this soon.

The Javascript library does do all the cryptographic tasks: authenticating Tor nodes, deriving symmetric keys, and sending encrypted traffic. Moreover, it can setup anonymous connections to any web server. The Javascript client offers a convenient interface to do so. This interface hides the complexities of communicating via the Tor network from the developer. See Listing 1 for an example.

All our code is open source and will be made available before the start of the workshop.[7] We aim to publish a fully integrated and performing prototype implementation.

[5] http://bitwiseshiftleft.github.io/sjcl/

[6] https://tweetnacl.js.org/

[7] https://github.com/spring-epfl/lightnion

TABLE I. Experimental results of latency in ms of sending packages to an echo server. We sent packages at a rate of 250 per second. We compare the baseline implementation featuring the regular Tor client to the in browser Lightnion Javascript client. We also measure the length of the total experiment.

	Round-trip Time		Total Time	
#pkts	baseline	in browser	baseline	in browser
50	101±18	135±30	200±0	376±10
500	119±77	452±462	2000±0	2493±277
2500	133±94	1397±835	10000±0	11353±522

B. Performance

We performed a simple performance evaluation of the current implementation of the Javascript Lightnion library in conjunction with the Python proxy. For these tests we used a small test Tor network constructed using chutney. This network runs 8 Tor nodes (or onion routers, of which at least 2 guards and 5 exit nodes). All tests were performed locally, they therefore do not take network latency into account. Despite using a test network, we expect the Lightnion overhead to be similar when using the real Tor network. The cost of parsing and verify the consensus (which we did not yet integrate, nor measure) does increase when using the real Tor network.

First, we measured the round-trip latency of our implementation using a simple echo server. We created a simple Javascript application running in a browser that uses Lightnion to send packages to this echo service. We measure the time between transmitting the package and when it returns to the Javascript application. We compare the performance of this setup with the baseline of a simple client sending packages using the regular Tor client. All clients, Tor nodes and the echo server run on the same machine. Hence, these experiments measure computational overhead only.

We send packages of 498 bytes (so that they fit within a single Tor cell) at a rate of 250 packages per second. We noticed that the proxy suffers from congestion. Therefore we compare the effect of sending packages for a longer time: 50 package in total (i.e., 200 ms), 500 packages in total (i.e. 2 sec), and 2500 packages in total (i.e., 10 sec). See Table I.

The table shows that longer connections have no significant effect on the latency for data sent using the regular Tor client. However, whereas the Lightnion Javascript client initially keeps pace with the regular Tor client, sending more packages causes a significant increase in latency. We suspect that mismanaged buffers in the Lightnion proxy are the cause of this. We aim to improve the proxy in the future.

We also profiled our Javascript implementation. About 40% of the time is spent encrypting and decrypting traffic, about 5% is spent performing integrity checks, about 20% is spent on garbage collection, and about 19% is spent on converting between different low-level Javascript data structures. We aim to speed up the Javascript library by handling garbage collection and type conversion better.

VI. Applications of Lightnion

We present four use cases where Lightnion can be used to provide anonymity for specific actions or against untrusted

third party servers.

Secure webmail. Consider a secure webmail client for encrypted e-mail. Users trust the provider of their webmail client. However, to find the GPG keys of e-mail recipients, the webmail client requests the keys from an untrusted key server. The key server then learns the user's communication patterns by virtue of seeing the key requests. Instead, the webmail application can use Lightnion to make this request to the key server anonymously to protect the user's privacy.

Accessing medical resources. Consider doctors accessing the results of a patient's medical tests using a trusted online platform. To provide extra information about the results, the trusted platform refers to untrusted external websites. When a doctor visits these external sites, these sites might learn which doctors have patients suffering from certain diseases by relating accessed resources to the doctor's network identity. The trusted platform can instead use Lightnion to retrieve the extra information without the untrusted external websites learning which doctor makes the request, thereby protecting both the doctor's and patient's privacy.

Privacy-preserving collaborative editing. Consider a privacy-preserving collaborative editing service such as CryptPad. This software uses cryptography to ensure that the platform does not know the content of the documents. Users trust or verify the software provided by the editing platform. However, the editing platform learn who edits which sensitive files based on incoming requests from the user's browser. Users may trusts the software provided by the editing platform, but do necessarily trust the provider to know which files they edit. To increase the user's privacy, the CryptPad application can make edit requests anonymously via Lightnion. As a result, the CryptPad server no longer learns who edits which files.

Anonymous questionnaire website. Consider an online questionnaire dealing with sensitive personal questions. To ensure the privacy of participants and to comply with regulations such as the GDPR, the questionnaire platform does *not* want to be able to link the answers to individual users. This linking can occur in two ways: (1) via data submitted back to the server, and (2) via information implicitly sent at the network layer as users submit answers. The platform can easily ensure that no explicit identifiers are submitted together with the answers. However, the network layer identifiers remain. To remove these identifiers, the questionnaire platform can submit the results anonymously via Lightnion. In this way, the questionnaire platform ensures that it cannot link answers to individual users.

A. *Why not just use a proxy?*

In all these scenarios users interact with both trusted platforms and untrusted end points. One way to ensure that users can anonymously access the untrusted server seems to be letting the service provider operate a centralized and trusted proxy server. This proxy server proxies the communication between the user and the untrusted end points, thereby anonymizing users with respect to the untrusted end points.

Using Lightnion instead of a trusted proxy has many benefits. One, Lightnion can be used in scenarios where a trusted proxy cannot. Consider the last two scenarios. There, the untrusted end point and the trusted proxy would coincide, violating the trust assumptions, and resulting in no anonymity for users. Two, operating a trusted proxy requires infrastructure and expertise, Lightnion would provide a decentralized generic solution instead of a custom centralized one. Three, when using Lightnion, users can reduce their trust assumptions in the service provider. Instead of fully trusting the proxy operated by the service provider, users have the option to verify the service provider's code.

VII. Related Work

Snowflake [5] aims to solve a different problem than Lightnion. Snowflake is a censorship circumvention system and is a successor to Flashproxy [3]. Snowflake assumes that censors will not block WebRTC connections as they are also used for direct voice and video communication between clients. To leverage this, Snowflake asks volunteers to run an in-browser Javascript *proxy*, the Snowflake, that bridges WebRTC connections from censored users to the Tor network. Censored users, however, still use the Tor Browser (or a normal Tor client) to connect to the Snowflake proxy.

The node-Tor[8] project shares Lightnion's goal to run a Tor client in a user's browser. It is used in demo applications iAnonym[9], an in-Browser alternative to the Tor browser, and Peersm[10] for peer to peer file sharing. Just like Lightnion, these projects use a proxy to translate between WebSockets and the Tor protocol. However, these projects seem stale (the open source code was last updated 6 years ago, the closed source library was last updated 4 years ago), and do not have a clear trust model. It is unclear if they are still operational (the Peersm demo did not work, and the iAnonym does not seem to have a public demo). Lightnion, instead, aims to provide well-documented open source code with a clear trust model that targets specific applications where a small in-browser Tor client for anonymous requests makes sense.

VIII. Conclusions and Future Work

We presented Lightnion, a system to enable seamless anonyous communication from a user's browser. The use of Lightnion is invisible to users and requires no active involvement from them. We created an initial research prototype that confirms that Lightnion's approach of operating a small Tor client is viable and seems to perform well enough for many scenarios.

We aim to turn the research prototype into a well-performing, robust piece of software that can be used in a number of scenarios. We furthermore hope to convert parts of the Lightnion proxy into a pluggable transport that can be operated by existing Tor nodes, allowing any web application to use Lightnion.

References

[1] J. Clark, P. C. van Oorschot, and C. Adams, "Usability of anonymous web browsing: an examination of tor interfaces and deployability," in *Proceedings of the 3rd Symposium on Usable Privacy and Security, SOUPS 2007, Pittsburgh, Pennsylvania, USA, July 18-20, 2007*, L. F. Cranor, Ed., vol. 229. ACM, 2007, pp. 41–51.

[8] https://github.com/Ayms/node-Tor

[9] http://http://www.ianonym.com/

[10] http://www.peersm.com/

[2] R. Dingledine, N. Mathewson, and P. F. Syverson, "Tor: The second-generation onion router," in *Proceedings of the 13th USENIX Security Symposium, August 9-13, 2004, San Diego, CA, USA*. USENIX, 2004, pp. 303–320.

[3] D. Fifield, N. Hardison, J. Ellithorpe, E. Stark, D. Boneh, R. Dingledine, and P. A. Porras, "Evading censorship with browser-based proxies," in *Privacy Enhancing Technologies - 12th International Symposium, PETS 2012, Vigo, Spain, July 11-13, 2012. Proceedings*, ser. Lecture Notes in Computer Science, vol. 7384. Springer, 2012, pp. 239–258.

[4] K. Gallagher, S. Patil, B. Dolan-Gavitt, D. McCoy, and N. Memon, "Peeling the onion's user experience layer: Examining naturalistic use of the tor browser," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1290–1305.

[5] S. Han, "Snowflake technical overview," Jan 2017. [Online]. Available: https://keroserene.net/snowflake/technical/

[6] L. N. Lee, D. Fifield, N. Malkin, G. Iyer, S. Egelman, and D. A. Wagner, "A usability evaluation of tor launcher," *PoPETs*, vol. 2017, no. 3, p. 90, 2017.

[7] G. Norcie, K. Caine, and L. J. Camp, "Eliminating stop-points in the installation and use of anonymity systems: a usability evaluation of the tor browser bundle," in *5th Workshop on Hot Topics in Privacy Enhancing Technologies (HotPETS)*, 2012.