2017 IEEE Symposium on Privacy-Aware Computing

LiLAC: Lightweight Low-latency Anonymous Chat

John P. Podolanko, Revanth Pobala,
Hussain Mucklai
Dept. of Computer Science and Engineering
University of Texas at Arlington
Email: {john.podolanko | revanth.pobala |
hussain.mucklai}@mavs.uta.edu

George Danezis
Dept. of Computer Science
University College London
Email: g.danezis@ucl.ac.uk

Matthew Wright
Center for Cybersecurity
Rochester Institute of Technology
Email: matthew.wright@rit.edu

***Abstract*—Low latency anonymity systems, like Tor and I2P, support private online communications, but offer limited protection against powerful adversaries with widespread eavesdropping capabilities. It is known that general-purpose communications, such as web and file transfer, are difficult to protect in that setting. However, online instant messaging only requires a low bandwidth and we show it to be amenable to strong anonymity protections. In this paper, we describe the design and engineering of LiLAC, a Lightweight Low-latency Anonymous Chat service, that offers both strong anonymity and a lightweight client-side presence. LiLAC implements a set of anonymizing relays, and offers stronger anonymity protections by applying dependent link padding on top of constant-rate traffic flows. This leads to a key trade-off between the system's bandwidth overhead and end-to-end delay along the circuit, which we study. Additionally, we examine the impact of allowing zero-installation overhead on the client side, by instead running LiLAC on web browsers. This introduces potential security risks, by relying on third-party software and requiring user awareness; yet it also reduces the footprint left on the client, enhancing deniability and countering forensics. Those design decisions and trade-offs make LiLAC an interesting case to study for privacy and security engineers.**

***Keywords*-instant messaging; anonymous chat; onion routing; dependent link padding; private communication**

I. Introduction

Instant messaging (IM) involves sending short messages in real-time to one or more intended recipients. Some of its uses involve sensitive information that needs to be quickly transmitted back and forth between parties (e.g. military and intelligence services, political activists, journalists and whistleblowers). Yet, today's IM systems are inadequate in providing those with strong security and anonymity guarantees. While modern IM systems offer end-to-end encryption and perfect forward secrecy, they provide no protection of the key *meta-data* of who is chatting with whom and when.

One system that provides some protection is TorChat, a chat service running over the Tor anonymity network. TorChat, however, suffers from known vulnerabilities in Tor: powerful adversaries that could observe a significant fraction of Tor traffic may deanonymize users. These attacks have been thoroughly explored, assuming a fraction of malicious Tor nodes [1], interception at Internet Exchanges [2], observed Autonomous Systems [3], [4], and adversaries actively manipulating Internet routing to enable interception [5]. Some techniques protect against strong adversaries, such as Dependent Link Padding (DLP) [6], [7], but these are typically too expensive for bursty general purpose communications. However, given the low traffic requirements of IM, we show that such techniques may be used at reasonable cost.

Contributions. We make several contributions around the design and evaluation of LiLAC, a Lightweight Low-latency Anonymous Chat service offering both strong anonymity and a lightweight client-side footprint. The novel LiLAC uses a directory server and circuits, much like Tor, but stronger anonymity through using DLP over of constant-rate traffic flows. In LiLAC there is a key trade-off between the system's bandwidth overhead and the end-to-end delay along the circuit. Thus, we experimentally tune LiLAC to minimize the overheads and keep delays at an acceptable level for interactive chat.

LiLAC was designed with usability and ease of deployment in mind. It operates over a web interface that works in typical browsers, and is very similar to a standard web chat service. A key contribution of this paper, is to examine the security trade-offs of this deployment choice, such as the reduction of the footprint on the client and the increased risks in relying on third-party software (the browser).

The rest of our paper is organized as follows: background is presented in Section 2. Section 3 contains the attack model assumed by LiLAC. In Section 4, we present the LiLAC's architecture and subsystems. Our experimental design and results are presented in Section 5. Open problems are outlined in Section 6. Related works are compared in Section 7, and we conclude in Section 8.

II. Background

We first provide a brief overview of prior work in secure communications, focusing on design elements that we combine and extent to design LiLAC.

978-1-5386-1027-5/17 $31.00 © 2017 IEEE
DOI 10.1109/PAC.2017.14

A. Chat Content Confidentiality

Off-the-record (OTR) messaging [8] introduces *perfect forward secrecy* (PFS) to secure chat. PFS uses frequent re-keying thought authenticated ephemeral Diffie-Hellman key exchanges. Thus compromizes of long-term secret keys do not allow an adversary to obtain symmetric keys used to encrypt past messages. Alexander and Goldberg improve this process of mutual user authentication [9] through the Socialist Millionaire's Protocol (SMP) [10]: two parties can derive a fresh strong secret key through a previously shared weak shared secret, such as a pass phrase. A number of IM clients, such as CryptoCat [11], deploy OTR with SMP. We also use SMP for end-to-end secrecy in LiLAC, but also provide anonymity guarantees.

B. Onion Routing

Tor [12], the popular anonymity system used by millions[1], implements onion routing [13]. Onion routing enables interactive anonymous communications with connections we now call *circuits* that pass through multiple relays on the path from the client to the server. In a circuit, the client establishes shared keys with each proxy (or *onion router*) on the path, and uses these keys to build secure tunnels. The tunnels are layered inside of each other—hence the name onion routing—and this prevents any one party from cryptographically linking the client with the destination server. LiLAC uses such onion-routing-style circuits for routing IM messages, and strengthens their security against passive traffic analysis through padding with cover traffic.

C. Dependent Link Padding

Anonymity systems, including onion routing, can be protected from traffic analysis by using *padding*, which is traffic generated in order conceal patterns. A well-known method for effective padding is Dependent Link Padding (DLP) [6], [14]. Consider a proxy (e.g. an onion router) that relays multiple circuits. DLP ensures that, if a message arrives on one of the circuits at time t, it is scheduled to be sent at the latest at time $t + \delta$ along with padding on each of the other circuits. If a message arrives on a second circuit at time $t' < t + \delta$, that message replaces the padding for that circuit, thus reducing bandwidth overhead. The parameter δ can be tuned to trade delay along the circuit for volume of padding. We note that this is particularly suitable for chat, since users can tolerate a moderate amount of delay (e.g. a few seconds), and that increased delay lowers overhead. A further extension is Reduced Overhead DLP (RO-DLP) [7], which takes advantage of the topology of the whole system to further reduce overhead. We apply RO-DLP in LiLAC.

III. The LiLAC Attacker Model

In this section, we briefly outline the attacker model for LiLAC. We aim to protect users from a global passive adversary (GPA), who can observe any packet being sent and received, including the source and destination addresses and when it was both sent and received. In vanilla onion routing observing both ends of the circuit is enough to perform a *traffic confirmation attack* and link the sender and receiver [12]. The GPA can essentially perform this attack on every sender-receiver pair to trace communications. While Syverson argues that the GPA is infeasible [15], the Snowden revelations suggest that it may be more relevant than once thought. Recently discovered applications of attacks on Internet routing to breaking anonymity systems make even a somewhat limited attacker as potent as the GPA [5].

The adversary may also attempt some limited active attacks. For example, a local eavesdropper or ISP may attempt a man-in-the-middle attack. The attacker may attempt to compromise one or more nodes in the system, or perhaps simply trick the people responsible for the system into accepting his freely volunteered computing and bandwidth resources. LiLAC assumes that at least one of the nodes on the circuit is honest. We also assume, in our prototype, that the *directory server* must be fully trusted. We explain this in Section IV-A, and note that this assumption can be eliminated though the use of standard techniques for distributing the directory.

Finally, we do not address long-term observation and intersection attacks in our system. The attacker can watch when users join and leave the system and look for patterns that link multiple users over time. The general form of this is called the *statistical disclosure attack* [16]–[19]. If the adversary can directly link two users with a single connection, as is possible in Tor, then this kind of longer-term attack is not required. Addressing this attack may be possible using the method of Wolinsky et al. [20] but leave this extension for future work.

IV. Design of the LiLAC Anonymity System

In this section, we present the details of LiLAC's design. Our primary objective is to design and implement a system that is secure, anonymous, and easy to deploy and use. LiLAC is composed of four components: the directory server, the presence server, relays, and clients. The directory server monitors changes in the network topology and maintains information about active relays and presence servers in the network. The directory server also hosts the LiLAC website and delivers content to the clients. The presence server maintains information about online clients and is the intermediary responsible for connecting two clients who wish to communicate. The relays operate much like relays in Tor [12] and use RO-DLP to prevent traffic analysis.

[1] https://metrics.torproject.org/

We examine each of these components as well as how they support our goals in the following subsections.

A. The Directory Server

The directory server is a trusted server that maintains the information required for LiLAC to operate as a system, consisting of information about the LiLAC relays and presence servers. When one of these nodes comes online, they must first register themselves with the directory server and provide their IP address, public key, and their role in the system – relay or presence server. The operator of the directory server should maintain a white list of nodes that she has determined can participate, and the system should check against this list before registering the node. As nodes go up and down in the system, the directory server maintains the list of which ones are online. LiLAC currently supports a single presence server, and so the directory server operates on a first-come-first-served basis. In other words, the first presence server to register itself becomes the active presence server; any additional servers become backups.

The directory server also serves content to LiLAC's clients. A LiLAC user enters the domain name or IP address of the directory server into a modern web browser, and the directory server responds with the web pages which includes the JavaScript code that makes up the LiLAC client. The directory server also provides this client the details of the current presence server the relays currently online. We chose the NodeJS framework on which to build our application. While NodeJS is commonly used for server-side rendering of JavaScript, we built the directory server so that the only JavaScript rendered server-side is that which is required for the directory server to function. To ensure a rogue directory server doesn't compromise anonymity and security for the clients, we allow for client-side rendering of JavaScript for key generation, circuit building, and conducting handshake protocols and SMP checks.

Unfortunately, the directory server must be trusted, which is the primary trade-off of using a web-based design. Since the Web server has all the client-side code, it could also modify all the code in arbitrary ways before delivering it to the client. Tor mitigates trust in the directory server by using a multi-party signature, in which multiple parties must agree to the same set of relay information. There is no point in doing this in LiLAC, however, since the web server would deliver the code to check this signature as well. The only way we see to mitigate this is to use a separate client application that could store the public keys of multiple entities and validate both code updates and directory updates using a multi-party signature, like Tor does. For now, the directory server remains the weakest point in the system.

To overcome this, at the cost of having smaller anonymity sets, a group of LiLAC users could run their own directory server and point their browser to the public domain or IP address on which their directory server is running. LiLAC pages can be served from any preferred framework (i.e. Flask, Django, Spring, etc.). This directory server could point to its own set of relays or piggyback on the existing relays and presence server, in which case the operator just downloads the `creds.json` file from the original LiLAC site. This file consists of the information of the location of relays and their public keys. The structure of `creds.json` is shown below:

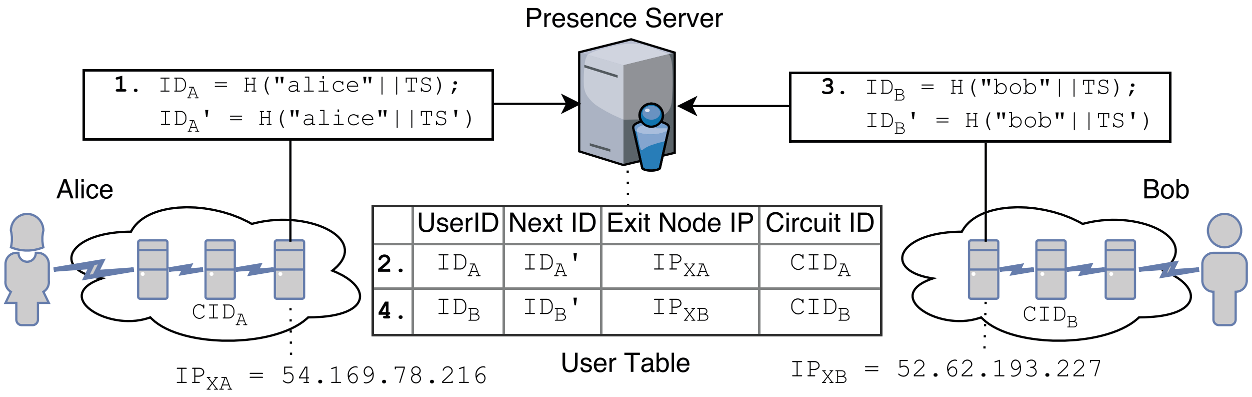
```
[{"host":"52.62.193.227",
"port":8091,
"publicKey":"5113daa74e909bef24e93f..."},
{"host":"54.169.78.216",
"port":8091,
"publicKey":"37bd39bce2fdb145e8e185..."}]
```

B. Relays and Creating a Circuit

Relays are volunteer-based onion routing servers that perform the same functionality in LiLAC as relays do in Tor [12] – anonymizing traffic. The client chooses three relays from the list, `creds.json`, provided by the directory server to build a circuit. We do not currently address bandwidth disparities, guards, or other path selection features currently used in Tor or proposed by others. With a large set of relays, these would be important considerations. Guards seem especially important for security [21]. In Tor, guards are implemented by retaining state at the client. In LiLAC, since no state is retained, we could instead use a consistent hash of the user's self-selected ID to seed a random number generator for selecting the guard. We leave detailed consideration of this issue for future work.

Once a client randomly selects three relays, it begins building a circuit. First, symmetric keys are generated for each of the relays starting with the entry relay. LiLAC implements the ACE protocol, which is an efficient adaptation of the Diffie-Hellman key exchange algorithm that provides one-way key authentication [22]. Upon establishing a symmetric key with the entry, the client can use this to encrypt and authenticate packets, thereby creating a tunnel. Through this tunnel, the client can then establish a symmetric key with the second relay using the ACE protocol, creating a tunnel within the first tunnel. The same process is repeated to extend the circuit to the exit (final) relay. Additionally, the client establishes a symmetric key with the presence server through the circuit once it is created.

During the period that the circuit is built, a loading screen is displayed on the client's web browser. Once the circuit has been established and extended to the presence server, the client is able to use the system and is prompted to enter a username and password.


Presence Server
1. ID_A = H("alice"||TS);
ID_A' = H("alice"||TS')
3. ID_B = H("bob"||TS);
ID_B' = H("bob"||TS')
Alice
Bob
CID_A
CID_B
IP_{XA} = 54.169.78.216
User Table
IP_{XB} = 52.62.193.227

	UserID	Next ID	Exit Node IP	Circuit ID
2.	ID_A	ID_A'	IP_{XA}	CID_A
4.	ID_B	ID_B'	IP_{XB}	CID_B

Figure 1. Connecting to the LiLAC presence server

C. *The Presence Server*

We now turn our attention to the presence server. Once the client's circuit is built, it establishes a shared symmetric key with the presence server and all communication between the client and presence server passes through the circuit.

1) Functionality: Unlike the directory server, it does not serve content to the clients, but it still renders its own functional JavaScript. The presence server's central role is to establish connections between chat partners. To do this, the presence server performs two functions: it maintains a list of online clients; and it responds to requests to connect clients. When a client accesses the LiLAC web application, it identifies itself by entering a username. This username is used to mark the client as online for a certain period of time, the *epochs* described below (§IV-C2). The client then sends its username (or a pseudonym derived from its username as described below) to the presence server which stores it in memory for future use.

To begin communicating, the client must submit a request to the presence server to start a chat by providing the username of the intended chat partner. The presence server receives this identifying username along with the username of the requester at which point the presence server searches for the recipient in its records. If found, it forwards the request with the requester's username to the intended recipient. To protect the recipient's privacy, no feedback of this part of the process is sent to the requester. In other words, if the client does not get connected to the recipient, then the client is not able to determine if the recipient is offline or simply ignoring the chat request.

2) Epochs and Time-Stamps: The presence server is critical to the design of LiLAC. To increase assurance, we aim to treat the presence server as adversary from the client's perspective. Therefore, the client divulges as little information as possible to this server, and none of that information can be used to ascertain the real identity of the client so long as their username is not indicative of their real identity. Even still, allowing the presence server to see the username for every client in plaintext may not be acceptable.

To avoid this threat, the LiLAC system operates on time periods known as epochs, and derived epoch specific pseudonyms for each username. An epoch is the minimum length of time for which a client would be marked online and open to receive chat requests. In our implementation, we used an epoch length of 15 minutes. We further define a timestamp TS as the number of epochs which have elapsed since the 1st of January, 1970.

When a client registers with the presence server, it hashes its own username concatenated with the timestamp such that $ID_A = H(\text{“}alice\text{”}||TS)$ where “$alice$” is the selected username and TS is the current timestamp. Before the client sends a chat request to the presence server, it similarly computes the chat partner's presence ID for this epoch, $ID_B = H(\text{“}bob\text{”}||TS)$, and makes a request to connect to this ID to the presence server. This request is covered in more detail below (§IV-D). To address moderate time synchronization issues, a client registers with the presence server by sending two identifying names – one for the current epoch and one for the next. The presence server receives both identifying names and maintains two collections of names: one for the current epoch and one for the next. This process is shown in Figure 1.

We note that this approach to private presence online is not perfect. The username acts as a key, and if the user selects a guessable username, then a curious presence server could apply a dictionary attack on any given epoch. Then, given one successful guess, it could search all other epochs for that username and corresponding timestamp. A better approach to private presence was proposed by Borisov et al. [23]. However, this approach relies on clients pre-sharing cryptographic keys, and incorporating this into LiLAC is future work.

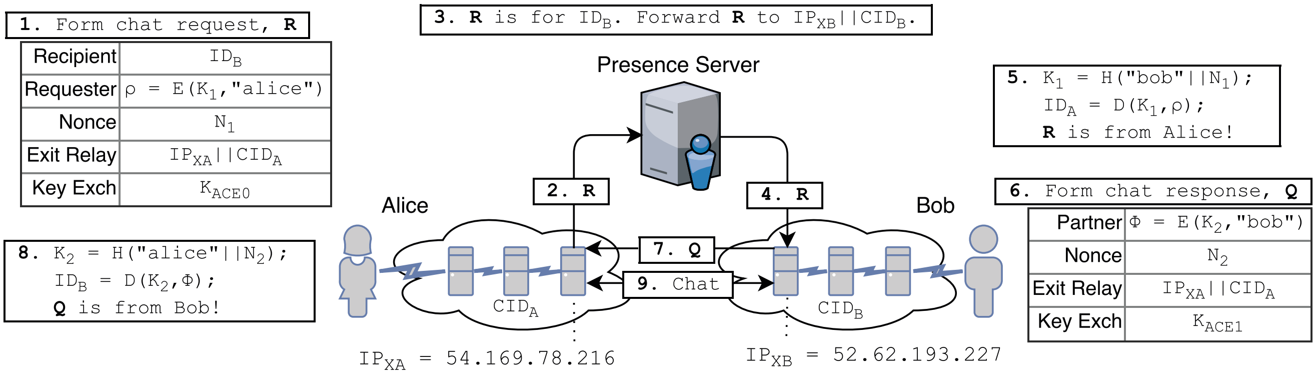

1. Form chat request, R
Recipient | ID_B
Requester | ρ = E(K_1,"alice")
Nonce | N_1
Exit Relay | IP_{XA}||CID_A
Key Exch | K_{ACE0}
3. R is for ID_B. Forward R to IP_{XB}||CID_B.
Presence Server
5. K_1 = H("bob"||N_1);
ID_A = D(K_1,ρ);
R is from Alice!
6. Form chat response, Q
Partner | Φ = E(K_2,"bob")
Nonce | N_2
Exit Relay | IP_{XA}||CID_A
Key Exch | K_{ACE1}
2. R
4. R
Alice
Bob
7. Q
9. Chat
8. K_2 = H("alice"||N_2);
ID_B = D(K_2,Φ);
Q is from Bob!
CID_A
CID_B
IP_{XA} = 54.169.78.216
IP_{XB} = 52.62.193.227

Figure 2. Connecting to a LiLAC chat partner

D. Connecting to a Chat Partner

Once two clients have registered themselves with the presence server and want to communicate, one of them initiates the conversation by submitting a chat request to the presence server. The key challenge is for users to find each other through the presence server. The ideal mechanism here is something private and secure like DP5. For our purposes, we developed a simpler mechanism based on a hash of the username that we could easily prototype which is shown in Figure 2. Let's say that two users, Alice and Bob, are registered with the presence server and that Alice wants to submit a chat request to Bob. This request R consists of five fields: the chat recipient field ID_B, the chat requester field ρ, a nonce N_1, the exit relay field, and the key exchange field. The chat recipient field is just Bob's ID as described above (§IV-C2). The chat requester field is the encryption of Alice's username using a key K derived from the hash of Bob's username with the nonce N_1, i.e. $K_1 = H(\text{"}bob\text{"}||N_1)$, and $\rho = E(K_1, \text{"}alice\text{"})$. Note that protecting the unlinkability of Alice and Bob relies on Bob's username being a secret from the presence server; again, a stronger private presence technique like DP5 could strengthen this mechanism. The exit relay field contains the IP address and circuit ID of Alice's exit relay, denoted by $IP_{X_A}||CID_A$. While the presence server can determine this information from its own connection with Alice's exit relay, Bob will still need this information in order to connect directly to Alice's circuit after accepting the chat request. Finally, the key exchange field will contain the first message in establishing a shared symmetric key via the ACE protocol [22], denoted by K_{ACE_0}. This request message is summarized in Table I.

Alice sends R to the presence server. The presence server extracts ID_B and uses it to verify Bob's online presence. If ID_B is found in its online records, then the presence server forwards R to Bob via his circuit. Once Bob receives R, he computes $K_1 = H(\text{"}bob\text{"}||N_1)$ and uses it to decrypts and read Alice's username. Bob will discover that R came from Alice (or someone posing as her) and will decide whether or not he wants to accept her request.

Chat Recipient:	$ID_B = H(\text{"}bob\text{"}||TS)$
Chat Requester:	$\rho = E(K_1, \text{"}alice\text{"})$
Nonce:	N_1
Exit Relay:	$IP_{X_A}||CID_A$
Key Exchange:	K_{ACE_0}

Table I
DATA STRUCTURE OF A CHAT REQUEST MESSAGE R

To accept request R, Bob sends a chat acceptance message to his exit relay. This acceptance message, denoted by Q, consists of four fields: the chat partner field ϕ, a nonce N_2, the exit relay field, and the key exchange field. For the chat partner field, Bob computes a key $K_2 = H(\text{"}alice\text{"}||N_2)$ and then uses the key to encrypt his username, $\phi = E(K_2, \text{"}bob\text{"})$. This field is required for Alice to verify that Bob (or someone posing as Bob) is the sender of the message. The exit relay field contains the information about Alice's exit relay, $IP_{X_A}||CID_A$, which was part of R and is needed by Bob's exit relay. The key exchange field holds the response – specified by the ACE protocol – for Alice to compute the shared symmetric key, denoted by K_{ACE_1}. This acceptance message Q is summarized in Table II.

Chat Partner:	$\phi = E(K_2, \text{"}bob\text{"})$
Nonce:	N_2
Exit Relay:	$IP_{X_A}||CID_A$
Key Exchange:	K_{ACE_1}

Table II
DATA STRUCTURE OF A CHAT ACCEPTANCE MESSAGE, Q

Bob sends Q through his circuit to his exit relay, which extracts IP_{X_A} and uses it to forward Q to Alice's exit relay

X_A. In turn, X_A extracts CID_A and identifies the correct circuit through which it will send Q to Alice. Once Alice receives Q, she computes $K_2 = H(\text{"}alice\text{"}||N_2)$ and uses K_2 to decrypt Bob's username. Alice will discover that Q came from Bob, at which point she uses K_{ACE_1} compute the shared symmetric key. Now, Alice is ready to communicate with Bob. She immediately sends an acknowledgment message to Bob to confirm the establishment of the conversation. Alice's exit relay now forwards traffic to Bob's exit relay until it receives a "Disconnect" message and vice-versa.

E. Security Features

We now discuss two security features of LiLAC: heartbeat padding and the implementation of the Socialist Millionaire's Protocol (SMP).

1) Heartbeat Padding: We aim to prevent traffic analysis in LiLAC by the use of padding. As discussed in §II-C, we apply RO-DLP to protect links between nodes. This does not, however, protect the timing information of real messages sent. An observer who sees a user sending messages followed by another user sending messages in a back-and-forth pattern can see that the two parties are chatting with each other. To cover this activity, we additionally queue all messages and send them out at a constant-rate, using random ciphertext as padding whenever a real message is not available to be send. Padding messages are generated to be indistinguishable from real messages: they consist of a random string S encrypted under a random fresh key K using AES (thus $M_H = E(S, K)$). We call this the *heartbeat* traffic in LiLAC, and we use a value of 300 ms as the default heartbeat interval. A shorter heartbeat interval decreases the amount of delay that a message suffers before being sent, while a longer heartbeat interval reduces the bandwidth overhead due to padding when real messages aren't available. We explore this trade-off in our experiments (§V).

Because of this heartbeat, the method for sending outbound messages needs to be adjusted. Rather than sending outbound messages immediately, they are placed in a queue. There is a separate queue kept for each outbound connection. After the passing of a heartbeat interval (i.e. every 300 milliseconds), the node removes the first message from every queue and sends it to the respective recipient. If a queue is empty, the node generates a dummy heartbeat message and sends it to the queue's corresponding connection. In this instance, it is worth noting that the term "node" can refer to clients, relays, and the presence server.

The implementation of cover traffic is useful in protecting LiLAC and its clients from statistical analysis of its traffic flow [18]. However, selecting the heartbeat interval involves a potential trade-off. Selecting a very short heartbeat interval may increase congestion in the network whereas selecting a longer heartbeat interval can increase the latency experienced by the clients – especially considering the fact that messages will be queued for the respective heartbeat interval at each relay. The overheads of operating the system using several different heartbeat intervals are discussed in our experimental section.

2) The Socialist Millionaire's Protocol: LiLAC uses the Socialist Millionaire's Protocol (SMP) [9], [10] to assure both parties that they are speaking with their intended chat partner, and derive a shared key between them for secure and authenticated communications. In short, one party asks a question such as "What is the pass phrase?" or "What is the name of the bar we met at on our second date?", and the other party answers the question. If the answer matches the answer given by the asking party, the protocol will succeed and a key will be shared. Else, the protocol will fail, and the users will be alerted. In LiLAC, SMP occurs after the full circuit and shared key between the chat parties are established. We use the first shared key that is established as part of the input to the answer, which assures both parties that they have both been involved in the communications so far. SMP needs to be initiated by one of the two chat partners. Before SMP is invoked, the user interface is designed to warn the users that their chat is insecure and that they should click the SMP button. We leave exploring the design, usability and effectiveness of this warning for users as future work.

For a malicious user, Mallory, to successfully execute a MITM attack, she would need to intercept traffic between Alice and Bob and establish different symmetric keys with both of them. Alice would falsely believe the symmetric key she shares with Mallory is shared with Bob. Similarly, Bob would falsely believe the symmetric key he shares with Mallory is shared with Alice. However, that's only possible if Mallory knows the challenge answers for both Alice and Bob. When Mallory receives the SMP request from Alice, she has two options. She may interfere and provide her own answer reply to the SMP request which will cause the SMP to fail assuming Mallory does not know the secret pass phrase entered by Alice. Alternatively, Mallory could let the SMP take place between Alice and Bob uninterrupted which will also fail because Alice and Bob possess different symmetric keys. Therefore, Mallory has no method of forcing the SMP to evaluate to true, and Alice and Bob will detect the MITM attack.

3) Application Footprint: LiLAC is lightweight such that it can be run from any computer with a web browser and an Internet connection, and it requires no special software, plug-ins, or client installation. It is entirely browser-based and allows a user to access the chat service without leaving any footprints on any trusted computer. Once a user is finished using LiLAC, the user must simply delete the browser history and clear the cache to remove any evidence that the computer was ever used to access the LiLAC website.

We encourage users to access LiLAC in the web browser's incognito or private browsing mode, so manual deletion of browser history and cache becomes unnecessary.

F. The Back-end

LiLAC is designed to be run on popular operating systems such as Windows, macOS, and most flavors of Linux. It is written in JavaScript and utilizes the NodeJS framework to run its processes. LiLAC can easily be deployed using cloud and container services such as Amazon Web Services (AWS), Heroku, and Docker. The only software dependencies required to run the LiLAC servers are as follows:

- Node.js
- Node Package Manager (NPM)
- Python 2.7
- Stanford JavaScript Crypto Library (SJCL)
- Google Crypto Library
- Bootstrap
- ecc2519 (to generate private and public keys)

To demonstrate LiLAC's capabilities, we have deployed a directory server and presence server on AWS using EC2 instances. Relay servers are placed in AWS EC2 instances, Heroku containers, and on physical computers in our lab in Arlington, TX. The complete network map of the LiLAC network is shown in Figure 3 where the red markers indicate relay nodes and the green marker indicates the presence server.

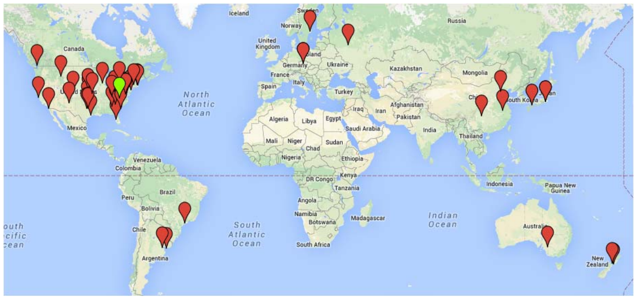


Figure 3. Deployed LiLAC Network

The source code for LiLAC has been made available to the public, and it can be found at https://github.com/revanthpobala/Lilac.

V. Experiment and Results

To evaluate the LiLAC design, we devise and execute a set of experiments using a real-world Internet deployment. Our experimental design consists of running LiLAC with a single directory server, a single presence server, and twelve relays. We then measured the performance of the system under the load of 1,000 clients. These clients were run on ten systems – one hundred clients per system. To achieve more accurate results, we had our clients and relays geographically dispersed. We used Amazon Web Services (AWS) to run most of our clients, relays, and our Presence Server. AWS provides server instances in nine geographic locations – namely Northern Virginia, Oregon, Northern California, Ireland, Frankfurt, Singapore, Tokyo, Sydney and São Paulo. These nine locations, along with our lab in Arlington, Texas, allowed us to globally distribute our system for experimentation. The summarized architecture is shown in Table III.

Node Type	# of Servers	Memory (GB)	Location(s)
Directory Server	1	4	Arlington, TX
Presence Server	1	32	Frankfurt
Relay	12	16	Global
Client	10	8	Global

Table III
Summary of Experiment Architecture

The experiment was run several times with varying heartbeat intervals. In order to conduct these experiments effectively, we had to automate the client-side code to automatically create a circuit and register with the presence server with a random string. The code base for the relays was not altered, but the behavior of the presence server was altered to maintain a variable, called `lastRegistered`, for the last client registered. The variable was empty when initialized. When the presence server received a registration request, it would check the value of `lastRegistered`. If it was empty, the presence server would set the value as the username it received in the request. If `lastRegistered` was not empty, the presence server would return its value to the client making the request and subsequently setting `lastRegistered` to empty. If the client received a username from the presence server, it would immediately send a request to the presence server to begin a chat with this client. If not, the client would wait for a chat request. Using this scheme we are able to continuously initiate communication channels between random clients. This is better illustrated in the following steps in which we shall refer to the two simulated clients as Alice and Bob:

1) The presence server initiates with $lastRegistered = null$.
2) Alice registers with username “$alice$”.
3) The presence server sets $lastRegistered =$ “$alice$”.
4) Bob registers with username “bob”.
5) The presence server sends back “$alice$” to Bob and sets $lastRegistered = null$.
6) Bob sends chat request for “$alice$” to the presence server, and the connection is established.

To measure system performance, the automated clients first create a circuit and record the circuit build time. Next, the clients establish a connection with a chat partner as described above. Clients would then send and receive messages continuously with their respective chat partners. A client sends a message and record the time the message was sent. When the chat partner received a message, it would calculate the time elapsed since the last message was sent and record this calculated value in a log file. The client then sends another message, records the time it was sent, and continues in this loop indefinitely. The finalized log file contains the time taken to build the circuit, followed by a list of times taken for a message to be sent to a chat partner and returned to the sender – or, simply put, round-trip times (RTT). The message send time, which is the time it takes for a message to reach its recipient, are effectively half the RTT. We ran preliminary experiments using heartbeat intervals of 150, 200, 300, 400, 500, 600, and 1000 milliseconds. Intervals of 150 and 200 ms led to high overheads at the servers that clogged the network, while intervals 600 and 1000 milliseconds led to very slow message delivery. As there are no hard and fast rules on how long we can make users wait (see, e.g. Nah's work on Web waiting times, which discusses wait times ranging from 2 seconds to more than one minute [24]), we relied on our own perceptions to rule out larger intervals. We thus ran the experiments twice using a heartbeat interval set to 300 and 500 ms, respectively.

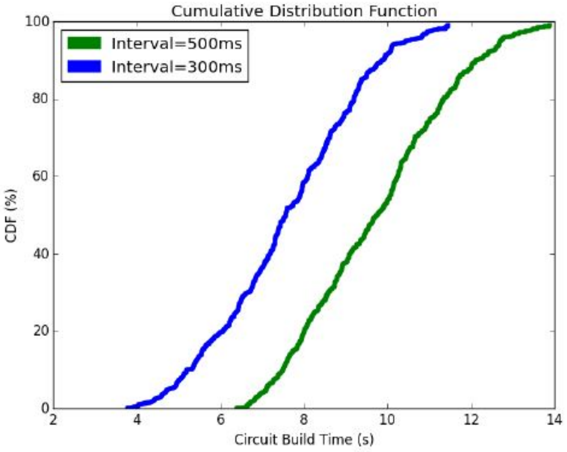


Figure 4. Cumulative distribution function for circuit build times

As shown in Figure 4, circuit build times are kept to under 10 seconds in most cases. Note that circuit creation messages are also forced into the heartbeat time restrictions, and this slows the process considerably. While this delay could be seen as lengthy, this is a one-time cost for the user, which appears like loading the page. The message RTT, as shown in Figure 5, has a median value of 3.0 seconds for a 300ms heartbeat interval and 4.6 seconds for a 500ms interval. This means that most messages reached their recipient in 1.5 or 2.3 seconds or less, respectively. We believe that this delay is reasonable for interactive chat.

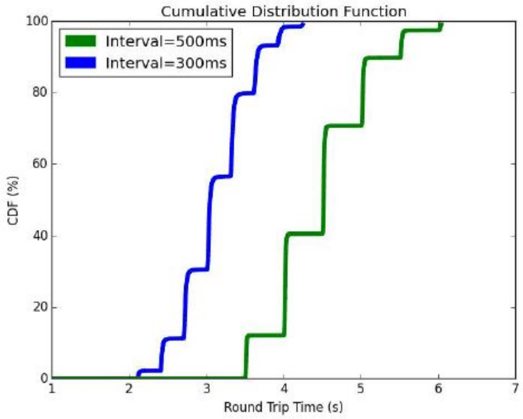


Figure 5. Cumulative distribution function for message round-trip times

VI. Discussion and Future Work

To increase the functionality of the LiLAC system and satisfy more of its users' requirements, there is still more work to be done. One feature would be the addition of group chat into the system. In implementing group chat, we need to consider the potentially negative effect on performance and anonymity that such a feature could cause. For example, a naive approach would be to have all users operate multiple circuits all the time, enabling multiple parties to be contacted at once without leaking evidence of the multi-party chat. This would reduce the number of concurrent users that the system could handle. Another approach would be to require messages to be multiplexed over the single circuit, but this creates contention and might only work for very small groups.

We would also like to incorporate DP5 [23] into LiLAC. DP5 provides strong privacy for presence information and supports high-integrity status updates to facilitate update and rendezvous protocols. DP5 is designed so that secrets are not long-term and perfect forward secrecy can be used in the event of compromise. On the other hand, DP5 requires additional back-end infrastructure and pre-shared key material between users.

DP5 does not entirely prevent attacks based on users' presence in the system, however, since an eavesdropper can see when the user connected to the system by direct observation. This means that a remaining open threat in LiLAC is the statistical disclosure attack [16]. This can be partially prevented by encouraging users to remain connected to LiLAC even when they are not actively using the chat service. Longer connection times make it harder to isolate users that are chatting with each other. Wolinsky et al. [20] propose grouping users and forcing them to be effectively online at the same times by blocking posts from the whole group while any group members are offline. This method is interesting, but it forces difficult trade-offs with communication availability that would need to be evaluated in our setting.

Another feature to be included in future versions is file transfer. Given the DLP dummy traffic, file transfers could be a considerably slow process as larger files are being transferred, so it would need to be fine-tuned, optimized, or bandwidth would need to be expanded. We would also need to limit file transfer sizes so other users aren't inconvenienced in the event another user tries sending large files through a relay they happen to share.

To expand upon the file transfer feature, we could add voice messaging to LiLAC. This would not be real-time voice over IP due to quality of service concerns, but we could allow the users to record voice messages of an arbitrary length – let's say 10 or 30 seconds – and transmit the packaged message to the other user as they would with any other file. The user would then play the entire voice message after it has been received and reconstructed.

We would, however, need a larger backbone to accommodate file transfers and voice messaging. LiLAC's current backbone consists of only one presence server backed by few load balancers. The presence server also currently acts as the bottleneck in terms of scalability; a thousand clients required almost 32GB of memory. As such, we will need to implement a protocol to utilize multiple presence servers which must be geographically distributed. One approach would be to have multiple regional presence servers which are subordinate to a global presence server. In other words, if there are five regions, then we deploy a single presence server to each of the five regions as well as a sixth presence server to govern them all. If a user in region 1 is looking for a user in region 3, the region 1 presence server would query the global presence server which would query the user in region 3 and establish the connection between users. This, however, still leaves a single point of failure. We propose that we cut out the middle man – the global presence server – and just establish communications links between all the regional presence servers through which they can broadcast queries for individual users not found in their region. When a regional presence server governing the requested user sees this query, it would respond with connection information.

Future versions should ideally do something to mitigate LiLAC's vulnerabilities. Having centralized directory and presence servers essentially creates multiple single points of failure for the entire system, not to mention a centralized authority implies trust. Any powerful DDoS attacker could essentially bring LiLAC down with little effort. Further, compromise of the directory server should not lead to complete privacy exposure.

Lastly, LiLAC will require volunteers to run relays and servers in support of a strong anonymity service for chat, much as Tor relies on altruistic volunteers [25].

VII. Related Works

There have been various approaches to designing and implementing anonymous communication systems. One such approach was proposed by Chaum [26], in which anonymity was provided by relaying messages through a series of nodes referred to as *mix nodes*. Mix systems require multiple public key operations to relay every message. A mix node is a server that accepts incoming connections from multiple clients, forwards them to their appropriate destinations in such a way that an eavesdropper is not able to ascertain relationships between inbound and outbound connections. Considering the negative implications of a single compromised mix node, messages are relayed through multiple nodes to increase the strength of the anonymity. LiLAC's architecture implements a variant of Chaum's mix nodes which are derived by grouping a cascade of nodes that accept messages from multiple senders. They then shuffle these messages and send them out in a random order to a subsequent set of mix nodes or to the circuit's exit node.

Similar systems, such as ISDN-mixes [27] and the Java Anon Proxy (JAP) [28], have been previously implemented. ISDN-mixes and JAP attempt to anonymize phone conversation traffic and web traffic respectively and have been designed to provide real-time anonymous communication. However, the synchronous manner in which they work is not well suited for today's bi-directional, asynchronous TCP/IP networks [29]. In contrast, LiLAC is designed to be a low-latency system that overcomes this drawback while still enabling real-time, anonymous chat.

Systems such as Mixmaster [30] and Mixminion [31] provide re-mailer anonymity for sending and receiving email messages. Email messages are sent to re-mailer servers in fixed-sized encrypted packets, with instructions on where to send the packet to next. This architecture has proven to be practical and effective where delays in data transmission are not critical to the application's use. It does not, however, suit the needs of real-time systems such as IM.

Onion routing, such as Tor [12], only requires public key operations to build circuits, and a circuit can be maintained for the duration of the chat session. Tor comes with an anonymous IM system that uses Tor hidden services [12] as its underlying network known as TorChat. One of the main advantages of using TorChat is that it supports file transfers which is currently unsupported in LiLAC. Viigipuu provides a good overview of how it works and a detailed analysis of its security [32]. With respect to anonymity, TorChat is vulnerable to attacks on Tor and Tor hidden services in particular. Kwon et al. demonstrate that an attacker observing a significant fraction of traffic from clients to guards can link clients to the hidden service they contact with just 20 Tor cells (fixed-sized units of data in Tor) [33]. More generally, powerful attackers who observe a large amount of Tor traffic entering and exiting the network can deanonymize many Tor connections, including both ends of TorChat sessions. Moreover, the Tor network is becoming more congested due to the growing number of users and the increased network traffic from different applications.

Cryptocat [11] is a browser-based IM system that provides security to the users by implementing SMP and onion routing similar to LiLAC. Like Tor Chat, Cryptocat also supports file transfers. However, Cryptocat fails to provide anonymity to the users which still makes LiLAC the first of its kind.

VIII. Conclusion

We have proposed a system for Lightweight Low-latency Anonymous Chat (LiLAC) that is customizable and open source, and we have publicly deployed this platform on the Internet. Our work aims to provide a simple and efficient architecture dedicated to anonymous IM. In this paper, we have discussed the anonymity and security benefits and other complementary features inherent in LiLAC's design that make it unique to its class. We have also addressed the many trade-offs in performance, the trust model, and feature set that leave room for improvement. We have shown through experimentation that the performance of LiLAC is reasonable for use as a real-time chat medium with an average round-trip time of 3.5 seconds and an average circuit build time of 7 seconds.

Acknowledgment

The authors would like to thank the reviewers for providing great feedback. We thank the University of Texas at Arlington and the Department of Education for supporting us with a Graduate Assistance in Areas of National Need (GAANN) fellowship. This work was also supported in part by NSF awards number CNS-1423163 and CNS-0954133 as well as Rochester Institute of Technology and the Signature Interdisciplinary Research Areas grant.

References

[1] B. N. Levine, M. K. Reiter, C. Wang, and M. K. Wright, "Timing attacks in low-latency mix systems," in *Proc. International Conference on Financial Cryptography (FC)*, 2004.

[2] S. J. Murdoch and P. Zieliński, "Sampled traffic analysis by Internet-exchange-level adversaries," in *Proc. Workshop on Privacy Enhancing Technologies (PET)*, 2007.

[3] M. Edman and P. Syverson, "As-awareness in Tor path selection," in *Proc. ACM Conference on Computer and Communications Security (CCS)*, 2009.

[4] A. Johnson, C. Wacek, R. Jansen, M. Sherr, and P. Syverson, "Users get routed: Traffic correlation on Tor by realistic adversaries," in *Proc. ACM Conference on Computer and Communications Security (CCS)*, 2013.

[5] Y. Sun, A. Edmundson, L. Vanbever, O. Li, J. Rexford, M. Chiang, and P. Mittal, "Raptor: Routing attacks on privacy in Tor," in *Proc. USENIX Security Symposium*, 2015.

[6] W. Wang, M. Motani, and V. Srinivasan, "Dependent link padding algorithms for low latency anonymity systems," in *Proc. ACM Conference on Computer and Communications Security (CCS)*, 2008.

[7] C. Diaz, S. J. Murdoch, and C. Troncoso, "Impact of network topology on anonymity and overhead in low-latency anonymity networks," in *Proc. Privacy Enhancing Technologies Symposium (PETS)*, 2010.

[8] N. Borisov, I. Goldberg, and E. Brewer, "Off-the-record communication, or, why not to use PGP," in *Proc. ACM Workshop on Privacy in the Electronic Society (WPES)*, 2004.

[9] C. Alexander and I. Goldberg, "Improved user authentication in off-the-record messaging," in *Proc. ACM Workshop on Privacy in the Electronic Society (WPES)*, 2007.

[10] F. Boudot, B. Schoenmakers, and J. Traore, "A fair and efficient solution to the socialist millionaires problem," *Discrete, Applied Mathematics*, vol. 111, no. 1, pp. 23–36, 2001.

[11] N. Kobeissi and A. Breault, "Cryptocat: Adopting accessibility and ease of use as security properties," 2013.

[12] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The second-generation onion router," in *Proc. USENIX Security Symposium*, 2004.

[13] D. Goldschlag, M. Reed, and P. Syverson, "Onion routing," *Communications of the ACM*, vol. 42, no. 2, pp. 39–41, 1999.

[14] P. Venkitasubramaniam and L. Tong, "Anonymous networking with minimum latency in multihop networks," in *Proc. IEEE Symposium on Security and Privacy (Oakland)*, 2008.

[15] P. Syverson, "Why im not an entropist," in *Proc. International Workshop on Security Protocols*, 2009.

[16] G. Danezis, "Statistical disclosure attacks," in *Proc. IFIP International Conference on Information Security: Security and Privacy in the Age of Uncertainty*, 2003.

[17] N. Mathewson and R. Dingledine, "Practical traffic analysis: Extending and resisting statistical disclosure," in *Proc. Workshop on Privacy Enhancing Technologies (PET)*, 2004.

[18] N. Mallesh and M. Wright, "Countering statistical disclosure with receiver-bound cover traffic," in *Proc. European Symposium On Research In Computer Security (ESORICS)*, 2007.

[19] G. Danezis and C. Troncoso, "Vida: How to use Bayesian inference to de-anonymize persistent communications," in *Proc. Privacy Enhancing Technologies Symposium (PETS)*, 2009.

[20] D. I. Wolinsky, E. Syta, and B. Ford, "Hang with your buddies to resist intersection attacks," in *Proc. ACM Conference on Computer and Communications Security (CCS)*, 2013.

[21] M. K. Wright, M. Adler, B. N. Levine, and C. Shields, "Passive-logging attacks against anonymous communications systems," *ACM Transactions on Information and System Security (TISSEC)*, vol. 11, no. 2, p. 3, 2008.

[22] M. Backes, A. Kate, and E. Mohammadi, "Ace: An efficient key-exchange protocol for onion routing," in *Proc. ACM Workshop on Privacy in the Electronic Society (WPES)*, 2012.

[23] N. Borisov, G. Danezis, and I. Goldberg, "DP5: A private presence service," *Proceedings on Privacy Enhancing Technologies (PoPETs)*, vol. 2015, no. 2, pp. 4–24, 2015.

[24] F. F.-H. Nah, "A study on tolerable waiting time: How long are Web users willing to wait?" *Behaviour & Information Technology*, vol. 23, no. 3, pp. 153–163, 2004.

[25] S. Chakravarty, A. Stavrou, and A. D. Keromytis, "Identifying proxy nodes in a tor anonymization circuit," in *Proceedings of the 2008 IEEE International Conference on Signal Image Technology and Internet Based Systems (SITIS '08)*, 2008.

[26] D. L. Chaum, "Untraceable electronic mail, return addresses, and digital pseudonyms," *Communications of the ACM*, vol. 24, no. 2, pp. 84–90, 1981.

[27] A. Pfitzmann, B. Pfitzmann, and M. Waidner, "ISDN-mixes: Untraceable communication with very small bandwidth overhead," in *Kommunikation in verteilten Systemen*. Springer, 1991, pp. 451–463.

[28] O. Berthold, H. Federrath, and S. Köpsell, "Web MIXes: A system for anonymous and unobservable Internet access," in *Designing Privacy Enhancing Technologies*. Springer, 2001, pp. 115–129.

[29] R. Böhme, G. Danezis, C. Diaz, S. Köpsell, and A. Pfitzmann, "On the PET workshop panel Mix cascades versus peer-to-peer: Is one concept superior?," in *International Workshop on Privacy Enhancing Technologies*. Springer, 2004, pp. 243–255.

[30] U. Möller, L. Cottrell, P. Palfrader, and L. Sassaman, "Mixmaster protocolversion 2," http://www.mixmin.net/draft-sassaman-mixmaster-XX.html, 2003.

[31] G. Danezis, R. Dingledine, and N. Mathewson, "Mixminion: Design of a type III anonymous remailer protocol," in *Proc. IEEE Symposium on Security and Privacy (Oakland)*, 2003.

[32] R. Viigipuu, "Security analysis of instant messenger TorChat," University of Tallinn MS Thesis, http://kodu.ut.ee/~arnis/torchat_thesis.pdf, 2013.

[33] A. Kwon, M. AlSabah, D. Lazar, M. Dacier, and S. Devadas, "Circuit fingerprinting attacks: Passive deanonymization of Tor hidden services," in *Proc. USENIX Security Symposium*, 2015.