Hiding Metadata from Email Servers

Chaitanya Agarwal
ca2719

Shantanu Dahiya
sd4500

1 Introduction

Confidentiality and Privacy in online communication have gained a lot of traction recently. While achieving confidentiality in communication is relatively easy using common protocols like TLS, Privacy remains a much tougher challenge. To give an initial idea about this difficulty, protocols like TCP and UDP require the visibility of source and destination addresses of each packet for delivery, inherently violating privacy.

In more common modes of communication like email, or web chat, visibility is significantly reduced for an individual observer on the internet. However, the very platforms hosting these services have a clear view of the communicating parties. There are laws that *punish* using this data. Still, several companies have shown a history of transgressions even in the face of such laws. Even governments themselves have indulged in snooping people's private emails, or obtaining them forcibly using court orders. Thus, for ensuring privacy, users need verifiable mechanisms that *prevent* such snooping.

Tor [4] is an example of a system that satisfies this. It can, under very realistic assumptions, guarantee privacy in communication. It relies on several volunteers to host Onion Routers (ORs). Due to encryption, these ORs can only know the immediate previous, or the immediate next hop of communication. Given that packets are routed through more than one such ORs, and ORs cannot communicate with each other to combine information, its impossible to know the origin and destination of a packet. However, volunteers only want to provide very limited computing and storage resources. This prevents several, at times essential, services like abuse protection, spam-filtering, mail storage, etc. Indeed, the problem with Tor doesn't seem to be a technical one, rather an economic one, as it seems that Tor over a funded server can solve the aforementioned problems.

However, for hosting email servers to be commercially viable, providers require some level topic extraction for targeted advertisements, with the alternative being opting out with a fee. This now threatens confidentiality, as topic extraction would normally imply servers having full visibility of the email. Pretzel [5] addresses this by proposing a mechanism where the servers, in participation with the users, can provide spam-filtering, topic extraction, and keyword search, while hiding the contents of the email from servers. However, Pretzel doesn't provide anonymity to communicating users.

We've looked at two solutions that either provide anonymity, or provide confidentiality. In this paper, we present an email protocol that gives both, while removing the discussed weaknesses. Our protocol is very Tor-like and can be used in conjunction with Pretzel, when two parties are communicating using *two different*[1] mail servers. Further, our protocol provides additional security features like forward and backward secrecy. In addition, by working with a server-client architecture, it can provide several additional features to a user. We further show that it doesn't have considerable computational overheads, which can aide its deployability.

In this paper, we first more thoroughly present the goals of our protocols and the threat model under which those goals are achieved in Section 2. Then, we present the protocol itself in Section 3. We then present a few experiments on the protocol to highlight its overheads in Section 4. Finally, we conclude with a discussion of related work in this space in Section 5, along with open problems that still need to be addressed in Section 6.

2 Goals and Assumptions

As stated in the Section 1, we assume a scenario where two parties are communicating. We call them Alice and Bob for simplicity. We formally state our goals for a scenario where Alice sends Bob a single message. We call this event a 'transaction'. This can naturally extend to Alice and Bob exchanging any number of messages in either direction. We then state the threat model under which our protocol meets these goals.

[1] It is important to note that emails are often exchanged using the same mail server, a case we don't address in this protocol.

2.1 Goals

Confidentiality+Privacy: For a given transaction, only Alice and Bob should be able to read the contents of the message. Further, for a given transaction, anyone except Alice or Bob can either only know that Alice is sending someone a message, or they can only know that Bob is receiving a message from someone, they cannot know both.

Asynchronous Communication: For attempting a transaction, Alice should not require Bob to be online to send a message. This is something Tor isn't able to guarantee as it would require the last OR to store messages for Bob. Further, key exchange can also be a issue, as in Tor, session keys required for encryption are exchanged right before sending the message and all hosts have to be online.

Forward and Backward Secrecy: Most of internet communication relies on the Diffie-Hellman Key Exchange Protocol for establishing symmetric keys. Under this protocol, if a third party can obtain Alice's or Bob's private key, the third party can read all contents of the transaction. Even if such an event happens, the third party shouldn't be able to know the contents of any future or past transactions involving either Alice or Bob.

Value-Added Features: As discussed in Section 1, the addition of value-added features require that Alice and Bob communicate using a server, where mechanisms such as those implemented by Pretzel can provide these features. Thus, Alice and Bob should communicate using commercial servers.

2.2 Threat Model

Our protocol essentially asks every user to communicate using mail servers that can provide good computing and storage powers. In such a system, we don't assume the mail servers to be friendly to their users. We assume that the mail servers will actively try to read their client's emails, and try to find out their communicating partners. They can try to modify the exchanged packets, inject fake public keys, or impersonate an involved party. Our protocol defends against such attacks. Some attacks may cause a transaction not to be completed. We assume that if Alice and Bob talk out-of-band and know that a transaction could not be completed, it would bring bad publicity to the bad-acting server.

All exchanged packets, including public keys, go through the server. The server can compute the corresponding private keys to decode the communication. We assume that the server wouldn't have unlimited computing power and wouldn't try to do this expensive operation for obtaining the details of a single transaction.

It's important to note that Alice's server will always know which transactions Alice is involved in, while Bob's server will always know which transactions Bob is involved in. We assume that these servers will not share this information with each other. This is why we assume that Alice and Bob always communicate using separate servers. If they communicated using the same server, that server would know Alice is talking to Bob.

A few steps in our protocol require Alice/Bob verifying Bob's/Alice's certificates. This clearly violates anonymity as the authority can link Alice and Bob. For the purposes of this paper, we assume that the certificate authority is trusted and doesn't attempt to determine which users are communicating with each other.

Protecting against traffic analysis attacks is also a common theme in systems that tackle anonymity. We haven't analyzed our protocol under such attacks and for now assume that no party partakes in such activities.

3 The Protocol

The email protocol consists of communication among four entities. These are the sender (*Alice*), the sender's mail server (*alice-server.com*), the receiver (*Bob*) and the receiver's mail server (*bob-server.com*). All communication occurs over TLS links. Moreover, the endpoints *Alice* and *Bob* always send and receive packets via their respective mail servers. The endpoints encountered between the mail servers are not described as the contents of packets will be fully encrypted going through them, revealing no information about the communicating users or message contents. The protocol is divided into the four steps, each of which must occur before every communication. We denote encryption as $E_{key}(plaintext)$ and decryption as $D_{key}(ciphertext)$.

3.1 Step 1: Server Key Establishment

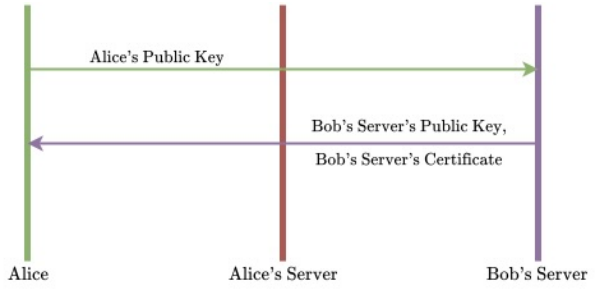

Figure 1: **Step 1**

Alice establishes a symmetric key (K_s)with *bob-server.com* using the Diffie-Hellman key exchange protocol. Since this key is established only for one message, this is ephemeral Diffie-Hellman key exchange, providing forward and backward secrecy for the sender's identity.

Importantly, when Alice's mail server *alice-server.com* sends out Alice's public key to *bob-server.com*, it excludes the identity of the sender, i.e. Alice, from the request, sending only the public key. This way *bob-server.com* does not know who it is exchanging keys with. This public key is used as an identifier when sending the message itself, so that *bob-server.com* knows which key to use to decrypt the receiver's identity in Step 3 (since it may be receiving messages from several senders.)

Additionally, the *bob-server.com* also sends its certificate which is verified by *Alice* to authenticate the receiving server, to ensure that it is not an impersonator. As noted before, we assume the certificate authority to be trusted.

3.2 Step 2: Client Key Establishment

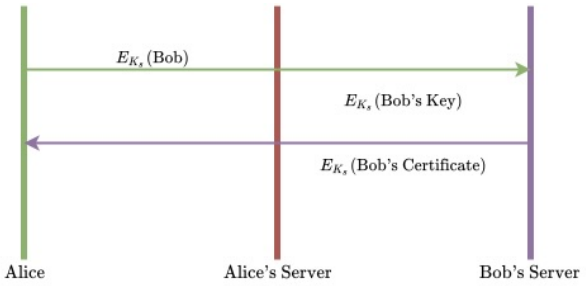

Figure 2: **Step 2**

This is similar to Step 1, except that the symmetric key now established (K_c) is between the sender *Alice* and the receiver *Bob*. The request goes through both servers *alice-server.com* and *bob-server.com*. But *alice-server.com* must not know the identity of *Bob* to meet our protocol goals. To enable this, *Bob*'s identity is encrypted with the key K_s that was established in Step 1. The request contains the following fields

- **ID Key:** *Alice*'s public key (acting as an identifier).
- **Public Key:** *Alice*'s public key.
- **Sender Domain:** *alice-server.com*
- **Receiver Domain:** *bob-server.com*
- **Receiver:** $E_{K_s}(Bob)$

When the request reaches *bob-server.com*, it performs the following actions.

- Checks the request type is for client key exchange.
- Uses the identifier key (*Alice*'s public key which is sent as part of the request) to lookup its database and determine which symmetric key K_s to use to decrypt the requested client's email address.
- Decrypts the receiver: $D_{K_s}(E_{K_s}(Bob)) = Bob$
- Selects a public key PK_{Bob} corresponding to that client (i.e. *Bob*) to send back to the sender.
- Sends back PK_{Bob} along with *Bob*'s certificate to *alice-server.com*.

The response contains the following fields

- **ID Key**: *Alice*'s public key.
- **Key:** PK_{Bob}
- **Certificate:** *Bob*'s certificate.

alice-server.com simply relays the response back to *Alice*. Note that since *bob-server.com* does not know the origin of the request, it simply sends the response back to the server it came from, i.e. *alice-server.com* with *Alice*'s public key sent back as an identifier. *alice-server.com* uses this identifier to know that the response is for *Alice*. To accommodate the scenario when *Bob* may be offline, we assume that every client shares a set of public keys in advance with its own mail server, from which it picks one to serve out when key exchange is requested with that client (for details see subsection 3.5). Clients also store their certificates with their mail servers so that they may also be sent out with the public key to enable a sender to verify the client's identity before sending an email.

3.3 Step 3: Message Reaches Receiving Server

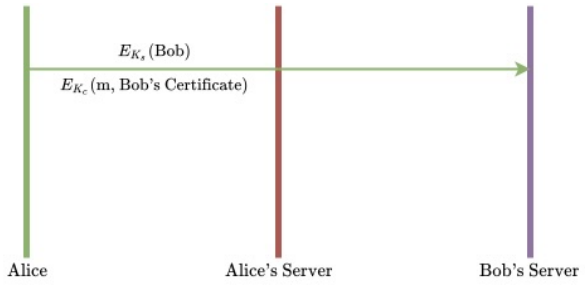

Figure 3: **Step 3**

Step 3 and 4 happen during the forwarding of the actual message to the receiver. The sender *Alice* encrypts its message *m* as well as its own identity with K_c, and the receiver's identity *Bob* with K_s. It again sends its public key as an identifier so that *bob-server.com* knows which key K_s to use to decrypt the receiver's identity. The request generated by *Alice* has the following fields

- **ID Key:** *Alice*'s public key
- **Sender Domain:** *alice-server.com*
- **Sender:** $E_{K_c}(Alice)$

- **Receiver Domain:** *bob-server.com*
- **Receiver:** $E_{K_s}(Bob)$
- **Message:** $E_{K_c}(m)$

alice-server.com simply forwards this request to the receiver domain *bob-server.com*. Noet that *alice-server.com* does not know the message or the receiver, as they are both encrypted and *alice-server.com* does not know K_s or K_c. Once the request reaches *bob-server.com* it performs the following steps

- Uses the id key to figure out the corresponding K_s to be used
- Decrypts the receiver: $D_{K_s}(E_{K_s}(Bob)) = Bob$
- Forwards the request to *Bob*

3.4 Step 4: Message Reaches Receiver

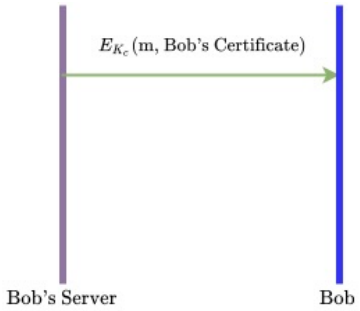

Figure 4: **Step 4**

When the message request reaches the receiver, it has the same structure as described in Step 3. The receiver has already been decoded by *bob-server.com* in order to figure out which user to forward the request to. *Bob* now uses the id key to figure out which K_c to use, then uses it to decrypt the message and the sender's identity

- Message: $D_{K_c}(E_{K_C}(m)) = m$
- Sender: $D_{K_c}(E_{K_C}(Alice)) = Alice$

In this way the message and sender identity reach the receiver *Bob* without being known to the receiver's mail server *bob-server.com*.

3.5 Step 0: Client Stores Keys On Server

As noted before, we want our email protocol to be able to handle sending emails without the requirement of the receiver being online. But Step 2 of the protocol involves a Diffie-Hellman Key exchange between the sender and receiver clients.

Entity	Sent	Recd
Alice	1114	529
alice-server.com	1652	1644
bob-server.com	1063	1122
Bob	0	534

Table 1: Per email data transfer in baseline implementation (bytes)

To enable the protocol to work while the receiver is offline, we want the receiver to generate their Diffie-Hellman public-private key pair in advance, and communicate the private key to its mail server. Then the receiver's side of the Diffie-Hellman key exchange can be carried out through the receiver's mail server without involving the client directly.

To further improve security, we propose a scheme where every client generates a set of public-private key pairs, where the size of the set is configurable by the client. After generating the set, the client shares the public keys in the set with its mail server, each paired with an identifying number. The client also stores the private keys against the corresponding identifying numbers in its own database.

A mail server, on receiving a client key request, selects one of the stored public keys of the corresponding client (either in round robin fashion or by random selection) and keeps a record of the identifying number against the id key of the request. When forwarding an email to the client (when the client is online) the server adds a field containing the corresponding identifying number to the request, so that the receiver can use the correct private key to decrypt the message and sender.

4 Experiments and Results

We implemented email communication using multi-threading over sockets and ran the experiments on the same machine for our two email implementations. The first (baseline) implementation encrypts the message but does not attempt to provide anonymity, i.e. it leaves the sender and receiver fields unencrypted. The second implements the protocol described in section 3, ensuring both confidentiality and anonymity.

We simulated the exchanging of 200 messages of random length between 1 and 200 bytes among 13 users, each belonging to one of 5 mail servers. We exchanged the same set of messages with the same senders and receivers across both implementations to ensure comparability. We measured performance along two kinds of values - processing time (i.e. the time taken to form a request or response in the protocol chain including encryption and decryption) and data size. The data from the simulations is presented in the table below

Since our protocol implementation adds overheads in terms of additional encryption and fields, along with a greater num-

Entity	Sent	% chg	Recd	% chg
Alice	1557	39.77	1037	96.03
alice-server.com	2609	57.93	2594	57.79
bob-server.com	1651	55.32	1572	40.10
Bob	0	N/A	614	14.98

Table 2: Per email data transfer in protocol implementation (bytes)

Entity type	Baseline	Protocol	% chg
Client	0.068	0.139	104.44
Server	0.230	0.640	178.26

Table 3: Average processing time (s)

ber of communications per email, we expect the performance to degrade from the first to the second implementation both in terms of data transfer requirements and average processing time per email. It is the extent of this degradation that we are interested in.

From Table 1 and Table 2, it is clear that for every email, received communication almost doubles for the sender while sending requirement increases about 40%, communication both ways increases about 40-60% for the servers, and the receiver receives about 15% more bytes in the end. While substantial, with modern data rates and the frequency with which email communication occurs, this is a reasonable price to pay for anonymity.

From Table 3, we can see that on average the processing time required at each client roughly doubles, and the processing time at the servers increases over 3 times. A client can be expected to easily bear the additional processing since it can be expected to only send a few emails per minute or even hour. However, the additional processing needed at the servers is a cause for some concern, since they may be expected to handle thousands of incoming and outgoing emails per second. Our protocol imposes a steep additional processing need per email on these servers, and optimizations to reduce this overhead will be immensely useful in making the protocol more practicable.

5 Related Work

5.1 OnionMail

OnionMail [2] is an email implementation over Tor. It essentially anonymizes the user's address while sending emails. For receiving emails, a user advertises certain Tor servers as introduction points (out-of-band), where the user's public keys along with her/his rendezvous points are distributed. Since this information is distributed, no individual server can retrieve the key, or know who the key belongs to. Senders can retrieve keys from the introduction points and send encrypted emails to the corresponding rendezvous points. Received emails are again distributed over a few rendezvous points so that no individual server can even attempt to read the email. Finally, the user can retrieve his/her emails from the rendezvous points. Note that after retrieval, the user is required to delete her/his emails from the rendezvous points.

Our protocol is, in some sense, a highly specific use-case oriented version of Tor and there are several parallels between our protocol and OnionMail. Like sessions are established sequentially between users and intermediate onion routers in Tor, our protocol establishes sessions between Alice and Bob's domain, and Alice and Bob. Our protocol also does a form of onion routing as the two domain servers only know their previous and next hop, i.e., their respective clients. However, unlike onion routing in Tor, we don't encrypt the payload recursively. We only encrypt it once using the session key between Alice and Bob. We feel that this provides adequate privacy, and aren't sure why Tor encrypts the payload recursively.

A major difference in our protocol and OnionMail, is the use of a commercial server setup. As highlighted in Section 1, domain servers can provide many features to clients. One such important feature is abuse protection, something that OnionMail can't reliably provide yet. OnionMail allows users to create custom blacklists of email addresses. However, it can be very painstaking for individual users to create such blacklists. This is much easier in a server setup as this task is delegated to the servers. Since, domains authenticate users for using their service and actively seek malicious users, other domains can trust these policies and confidently relay these emails to their users. Conversely, if some user comes from a relatively unrecognized domain, the server can alert the user about possible spam. Furthermore, the server setup yields slight performance benefits to the client as well. This is specifically useful during the offline key exchange. In our protocol, Alice only needs to contact Bob's server for his keys. However, in OnionMail, a sender would have to query several introduction points.

Ultimately, our protocol compromises some anonymity (the server knows when the client sends emails, and how many emails the client sends/receives) to add extra features over OnionMail.

5.2 iCloud Private Relay

iCloud Private Relay [1] implements the same ideas as Tor to provide anonymity while routing. It allows a user to access the web such that web hosts and other CDNs don't see the user's activities. The Apple server plays the role of Alice's server, while CDNs that host a website, play the role of Bob's server. Apple doesn't know what website the user wants to access, while the CDNs don't know who has accessed the website. Much like our protocol, the iCloud private relay also leverages the server setup to provide additional functionalities

to the users. Unlike Tor, which can't provide location-specific content for the user, the Apple server computes geohashes using the user's IP address to extend this functionality. This is of course in addition to the multitude of features and the good UI users get while using Apple products. In a way, it's a much more seamless way for an average user to get anonymity, something we hope our protocol can provide too.

6 Discussion and Open Problems

We have presented a protocol that allows two users, with different mail servers, to communicate anonymously over email. Further, our experiments show that the protocol has minimal bandwidth overheads and computations costs. Thus, we have essentially given a very initial blueprint for a commercially-viable email hosting service that can provide security features like confidentiality and privacy, while also providing value-added features like abuse protection, spam-filtering, mail storage, etc. However, much more research is needed before such a system can actually be deployed. We provide some directions that are worth exploring to take this idea forward.

Deployability: Even though, our protocol has minimal overheads, it needs to be compatible with the current standards of email protocols to be deployable. In particular, email transfer protocols like SMTP need special attention as it involves several parties. From our research, SMTP seems to be a protocol where Alice's server doesn't care who Bob is, it only cares about Bob's domain, and similarly, Bob's server doesn't care who Alice is, it only cares about Alice's domain. Further, the intermediate mail transfer servers also only care about the involved domains. Given this limited evidence, it seems that our protocol, after some modifications to the servers should be deployable. However, there may exist several servers that may not agree to participate in the protocol, or may not have the computing ability for the required encryption/decryption operations. If a user wants to send an email to such a server, all security features would be absent. Thus, further research is needed to assess the deployability of our protocol.

Traffic Analysis: We haven't analyzed our protocol for traffic analysis attacks. Given how such attacks can reveal the anonymity of the users, it is certainly a limitation of our work and the effect of such attacks should be analyzed carefully.

Alice and Bob communicate using the same server: We have made a very strong assumption regarding Alice's and Bob's mail servers. Our protocol, as stated in Section 2, fails in the case when Alice and Bob communicate using the same server. While our protocol can certainly help in the different domain case, a lot of users still communicate with users having the same domains. There has been a lot of research into this area, and several classes of techniques have been proposed in literature. Most of these techniques also provide security against traffic analysis attacks. Among the most prominent are private information retrieval [3], mixnets [6], and adding cover traffic [7]. However, they aren't realizable yet due to performance (private information retrieval), latency (mixnets) and/or bandwidth issues (cover traffic).

Certificate Authority Assumption: Our certificate authority assumption is certainly unrealistic. Even if it might be operated by 'good' people, there is always the danger that it can be compromised, which can again violate our goals. One can possibly use Tor like networks to verify certificates, but that adds a lot of overheads. Further, if a certificate authority allows anonymous users to verify certificates, it can fall prey to DoS attacks. More research is needed in this area.

References

[1] icloud private relay. https://www.apple.com/privacy.

[2] Onionmail. https://onionmail.org.

[3] Carlos Aguilar-Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. Xpir: Private information retrieval for everyone. *Proceedings on Privacy Enhancing Technologies*, 2(2016):155–174, 2016.

[4] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. Technical report, Naval Research Lab Washington DC, 2004.

[5] Trinabh Gupta, Henrique Fingler, Lorenzo Alvisi, and Michael Walfish. Pretzel: Email encryption and provider-supplied functions are compatible. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 169–182, 2017.

[6] David Lazar, Yossi Gilad, and Nickolai Zeldovich. Karaoke: Distributed private messaging immune to passive traffic analysis. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 711–725, 2018.

[7] Jelle Van Den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 137–152, 2015.