CrowdMixer: Differentially Private Anonymous Communication

Alex Yang and Sebastian Angel

Introduction

In the era of mass surveillance, there is a growing need for anonymous communications systems. Even if the content of anonymous messages is encrypted, adversaries can learn information through metadata such as who they are communicating with, the size of the messages, and the time of communication. The leakage of metadata itself compromises user privacy, posing a problem to individuals who require complete anonymity such as government whistleblowers and activists under repressive regimes.

Previous anonymous communications systems such as Vuvuzela[1] and Karaoke[2] are able to hide metadata, tolerate compromised servers, achieve scalability, and deliver messages with good latency. However, they still face two limitations: First, oftentimes the number of users of an ACN is very small, which makes connecting to the system inherently suspicious and less anonymous. Second, these systems only allow for ephemeral messages that are not retrievable over time. CrowdMixer aims to expand upon existing protocols to address these two challenges. The privacy guarantee can be quantified using the concept of differential privacy, which utilizes the parameters ε and δ. A low ε and δ indicates a higher standard of privacy for a user.

Project Overview

The overall goal of CrowdMixer is to build a system that meets the same standards as previous systems with the addition of 2 capabilities: increasing the number of users connected to the system and allowing messages to be stored for a set period of time. To achieve these goals, the CrowdMixer utilizes a heavily modified version of the Vuvuzela protocol with improved adaptive composition theorems[3] to generate the noise necessary for the storage of messages over time.

CrowdMixer also increases the number of users connected to the system through JavaScript code in a Chrome extension, creating involuntary “passive” users that are indistinguishable from real “active” users of the system. This thereby increases the anonymity set and makes the act of connecting to the system less suspicious. The JavaScript code constantly runs on a user’s chrome background processes, but doesn’t take up much computing power due to how infrequently it runs. As long as a user has a chrome instance open, which is highly probable, they will appear to the adversary to be the same as active users - indistinguishable. The prototype includes a demonstration background script that can be hidden in any chrome extension. While there are potential ethical concerns surrounding this setup, these concerns have been covered in depth by papers such as “CoverUp”.

Another improvement over Vuvuzela is the distributed server architecture. Instead of having one chain of servers, it uses two chains of entry servers. This has two major advantages. First, it allows for each server chain to generate half the noise, which significantly reduces the latency behind the system for the first two hops and also distributes the computational cost.

Second, it allows for users to generate cover traffic (by hashing to the same mailbox) when they are not communicating with anyone. If only one server chain was present and the entry server was compromised, an adversary would be easily able to tell if passive or active users were generating double exchange messages given their IP addresses and addresses of the mailboxes they are trying to hash to. Two servers with different public encryption keys means that the adversary cannot tell when cover traffic is being generated by active or passive users since the same message mailbox addresses look different to each server. Finally, given our case of using 2 concurrent requests for each user, this kind of model is the most fitting.

This system aims to meet standards of ε, δ differential privacy, with parameters for ε not exceeding ln(2) and parameters for δ not exceeding 1 x 10^-4 over k rounds of communication, which are derived from Vuvuzela as being sufficient in the realm of differential privacy. It is important to note that privacy guarantees are independent of the number of connected users, increasing the system's scalability.

General Concepts/Big Ideas

There are three major concepts that we would like to focus on in this project.

1A - Generating cover traffic

In order to effectively generate cover traffic with fake users, the fake traffic must be indistinguishable from the real traffic. We either have to bring communications traffic closer to the appearance of web-browsing traffic, or web-browsing traffic closer to the appearance of communications traffic. Initially, the idea was to do the same thing as in coverup whereby a JavaScript iframe could generate a connection between the passive user and the server. However, there are several huge limitations to this. First, most users don't stay on a webpage for very long at all. Thus, you can't at all rely on this providing consistent cover traffic. Second, real users who do communications traffic end up staying on a lot longer than fake users who are just browsing the web. As a result, it becomes very easy to use statistical disclosure attacks to figure out who and when the real users are talking to.

As a result, we settled on a compromise in the middle. We would instead use a background script in a chrome extension that periodically connects to a server to generate the fake cover traffic. There are several advantages to this. First, cover traffic is being generated as long as a chrome browser instance is open. Given how much time people spend on the internet these days, this will lead to consistent generation of cover traffic. Second, this background process can be hidden in a chrome extension without generating much user overhead, ensuring that a passive user's experience is not affected. Finally, this can offload some of the computations of cryptography and generating the fake messages onto the passive users, reducing the load once again on the servers.

2A - Non-ephemeral messages

The problem with many ACNs today is that they don't allow for messages to be preserved. Both users must be online at the same time in order for them to receive messages. To remedy this, we use a "sliding window" concept. Messages sent can be stored for up to 6 hours. When an active user wants to retrieve the sent messages, they can send 2 concurrent requests to past mailboxes since they have the secret key and number of rounds that they missed. Eventually, they will be able to "catch up" to the current round due to the 2 concurrent requests per round. Alternatively, the 2 concurrent requests could be used to generate cover traffic by hashing to the same mailbox in the current round, or even used to hold 2 concurrent conversations with different users in the current round.

The disadvantage to this setup is of course increased sensitivity, and thus increased noise. However, the noise increase is not drastic, and we believe that the costs of this on the latency are worth the benefits and flexibility garnered from this setup. It transforms the system from one that relies exclusively on real time messaging (and thus must be low latency) to one that can operate like an email style system with more of a high latency, but with the option to retrieve past messages. Future work could potentially increase this 6 hour timeframe that we have chosen to store messages.

3A- Improved Server Topography

The final concept we would like to focus on is the improved server topography of having 2 chains of servers. This allows us to generate half of the noise per server chain before sending it to the final mailbox server, which greatly reduces the latency in the first two hops. Additionally, it also allows for the generation of cover traffic by idle users when they are not engaged in conversations, and overall decreases the amount of computation performed per server.

However, there is a large bottleneck of messages from the mailbox servers back to the two mixnet servers. There is not much that can be logically done to optimize this besides not coding in JS, and instead using a production language such as C. Surprisingly, the JavaScript's single threading processing is not the main issue, but rather the inability to send data to servers quickly.

Composition Theorems and Math Overview

By far, the most challenging part of this was figuring out the math behind concurrent requests. Most of the formulas that we used can be derived from ones proven in Vuvuzela. However, the sensitivity calculations are entirely different.

1B. Sensitivity

In order to calculate maximum sensitivity, we model the mailbox server as taking in two types of requests - single messages (m1) and double messages (m2) . Double messages are when the server receives two addresses to the same mailbox and makes a swap. Single messages are when the mailbox either gets a single message and stores it for later or when a mailbox gets a single message and retrieves a previously stored request. To fully model the sensitivity, we will use the example of users A and B in different scenarios.

Scenario 1:

- A sends to mailbox a, b
- B sends to mailbox a, b
    - Crowdmixer swaps the messages, resulting in a sensitivity of 2 for m2

Scenario 2:

- A sends to mailbox a,b
- B sends to mailbox b,b
    - Crowdmixer swaps a,a - b,b resulting in sensitivity of 2

Scenario 3:

- A sends to a, c
- B sends to b, b
- C sends to a, c
    - As a result, we have a sensitivity of m1 = 0 and m2 = 3, but the sensitivity difference caused by A is once again 2

Scenario 4:

- A sends to a, c
- B sends to b, b
- There exists pre-existing dead drops e, f
    - While e, f may have higher sensitivity, A does not affect this. As a result, the sensitivity will be m1 = 2 for A specifically

Scenario 5:

- A sends to a, c
- B sends to b, b
- There exists a pre-existing a, c
    - For this calculation, e, f are single access deaddrops with a sensitivity of 2 x number of rounds stored -> maximum sensitivity = m1= 6 x 2 + 2(current round) + 0 (sensitivity of m2) = 12

Scenario 6

- A sends a, c
- B sends to b, b
- There exists a pre-existing a, d
    - For this calculation, a is single access deaddrops with a sensitivity of 1 x number of rounds stored -> maximum sensitivity = m1= 6 x 1 + 2(current round) + 0 (sensitivity of m2) = 8
    -

Scenario 7

- A sends to a, a
- B sends to b, b
- Pre-existing c, d
    - For this calculation, a is single access deaddrops with a sensitivity of 1 x number of rounds stored -> maximum sensitivity = m1= 6 x 1 + 2(current round) + 0 (sensitivity of m2) = 8

Scenario 8

- A sends to a, b
- B sends to a, b
- Pre-existing c, d
    - Once again, c , d don't really affect A that much, unless A sent it
    - In that case, c, d will translate into 2 x number of storage rounds, which as a maximum of 2 x 6 = 12 + 2 (m1 of current round) = 14

That was pretty confusing, so let me boil this down. Basically, no matter what, it is impossible for a single user A to increase the M2 accesses of one round by more than 2 since he/she has only 2 concurrent requests. Since M2 accesses don't persist over rounds, that means we can model it exactly the same as Vuvuzela of sensitivity 2/epsilon per round. However, assuming that A deposits a single request in some previous round, that noise does carry over. That means that for every round of storage, we multiply the number of rounds of storage x the single access requests stored. If A stored 2 single access 4 rounds ago, we get 2 x 4 = 8. Since the maximum storage is 12, we can get 2 x 6 = 12 / epsilon as the maximum sensitivity from storing single requests.
Finally, A can also have 2 single requests retrieving a previously stored request in the current round, resulting in a 2/ epsilon.

Adding all of these sensitivities together, we get 2 / epsilon + 2/ epsilon + 12 / epsilon = 16 / epsilon.

2B - Applying the sensitivity

In order to apply our parameter of 16 over epsilon, we once again model this in a similar way to Vuvuzela. Once we have calculated the u, b parameters for the laplace distribution based on the overall sensitivity of 16/ epsilon, we can then sample Laplace(u/2, b/2) for the double access message noise, and Laplace(u, b) for the single access message noise. There are 7 types of single access noise - noise that is retrieved in the next 1-6 rounds, and noise that is never retrieved. The total laplace for single access messages is then divided between these 7 types of noise via a Poisson distribution. More on this will be explained later.

3B - The Poisson Distribution

One of the difficult unresolved issues of the project is how to model the time distances between retrieval and depositing of messages, if the messages are to be retrieved at all. According to Shuva Gupta (Penn stats professor), the best way to model this would be to pick a distribution AFTER looking at real life data and trying to do some model fitting. While that would be nice, it is firmly out of the scope of this project, so as a result we can only guesstimate.

My best guess on this subject is to divide the single access messages into two overall categories: messages that are retrieved with Pr(A) and messages that are never retrieved with Pr(1-A). This would be modeled once again off of real life data. Next, we can assume that of the messages that are retrieved, each of the 6 time intervals is equally probable, so the expectancy of 1 + 2 + 3 + 4 + 5 + 6 / 6 = 3.5. This means the average time is 3.5 rounds, so we plug this as a parameter for a poisson distribution - Poisson(3.5), from which we draw for each individual retrieved messages how many rounds it will be retrieved in. If the Poisson distribution draws a number greater than 6, we discard the result and try again, so it is actually a truncated Poisson distribution.

The primary advantage of using the Poisson is that it produces discrete numbers, unlike an exponential distribution which could also model the result but would need to be rounded. Additionally, Poisson distributions are well established for determining time between 2 random events, such as what was performed in the Loopix paper. If we are able to get a dataset to look at in the future via some kind of similar model or a small-scale deployment, model fitting would certainly be the best option to determine what distribution works best to mirror real life traffic.

4B - The K-fold Adaptive Composition Theorem

Instead of using the Theorem from Dwork's textbook, we chose to use an improved K-fold theorem to model the privacy degradation over time from a 2015 paper. It looks like this:

$$\frac{(e^{\varepsilon}-1)\varepsilon k}{e^{\varepsilon}+1}+\varepsilon\sqrt{2k\,\log\left(e+\frac{\sqrt{k\varepsilon^{2}}}{\tilde{\delta}}\right)}$$

K represents the number rounds, epsilon is the epsilon for a single round, and the delta-looking parameter underneath is arbitrary, c = (delta prime) - (original delta) x K. In our case, we chose this arbitrary parameter to be c = 0.0003 for optimal balance between delta prime and epsilon prime.

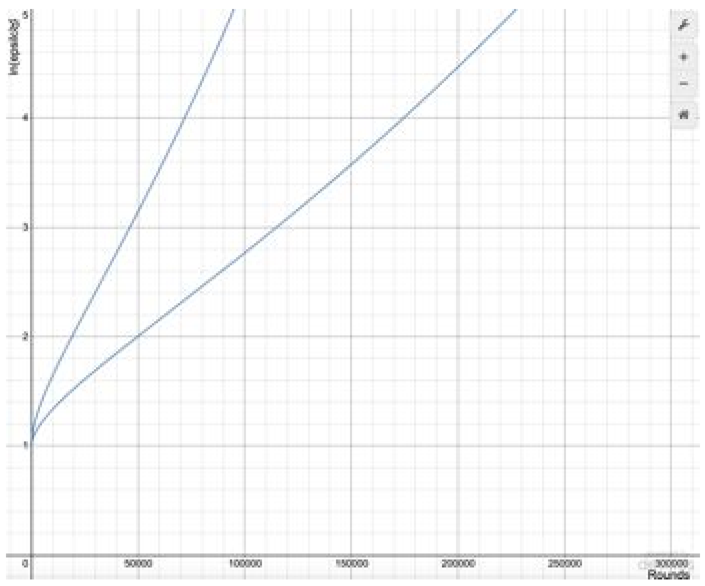
We also have some equations derived from the original Vuvuzela paper regarding their integration of a truncated Laplace mechanism.

$$\mathrm{b}=\frac{t}{\mu}$$

Delta = $e^{\left(\frac{2-u}{b}\right)} + \frac{1}{2}e^{\left(\frac{14-u}{b}\right)}$

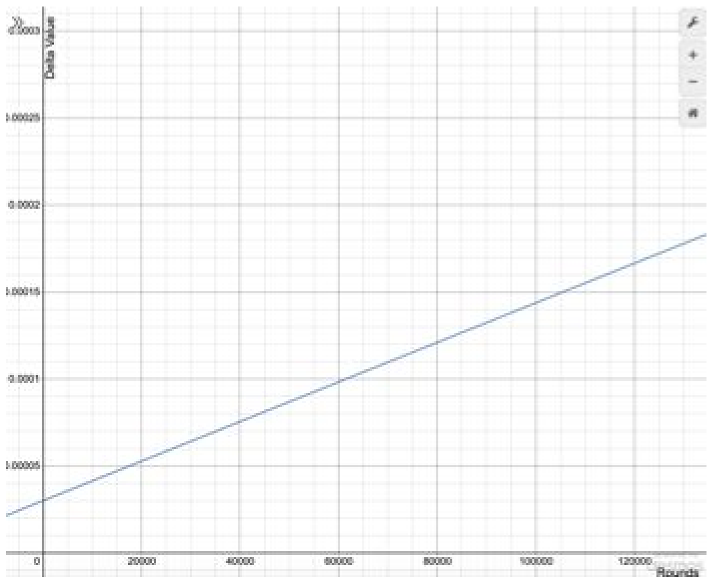
When we combine these various equations together, we found that setting u = 315,000 and b = 15,000 allowed for k = 50,000 rounds before privacy degrades to epsilon = ln(2) and delta degrades to 1x10^-4, the same targets that Vuvuzela aimed for.

Here is the graph of ln(ε) over k rounds, μ = 315,000 :



The left line shows the degradation under the original k-fold composition, indicating that this improved k-fold composition is indeed an improvement. Yay!

This is the graph of δ over k rounds, μ = 315,000



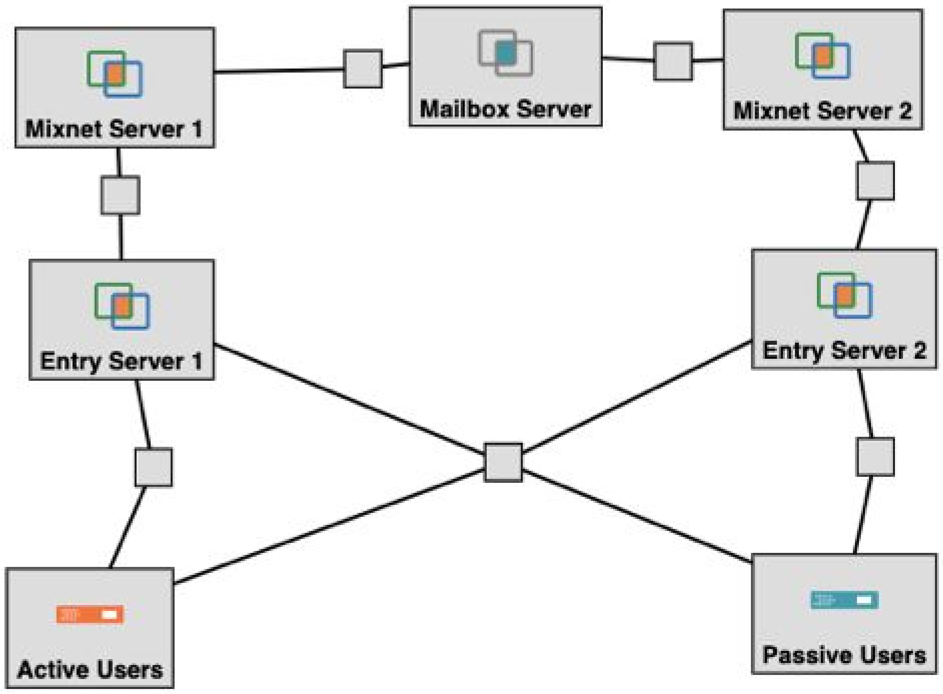
The most important overall takeaway is that the amount of noise u = 315,000 isn't really that much higher than the original Vuvuzela's 150,000 - 300,000 noise messages per server. It is definitely worth the increased latency for the concurrent requests and message storage.

5B - Combining the Passive and Active Messages
We can assume that there exists R passive users sending passive messages every round. This itself can be treated somewhat like a mixnet, since the passive users randomly choose when to send their requests using Math.random(). Because of this already existing mixnet, that means we can scale down the noise generated proportionally to the passive users. Instead of Laplace(u, b) , we would instead sample from Laplace(u - R, b), further decreasing server load.
6B - Choosing K
Whereas Vuvuzela chose K to be 100,000, our K is set at 50,000 with good reason. Vuvuzela operates on rounds every minute. We operate on rounds every hour. That means that Vuvuzela runs for 100,000 minutes while we run for 50,000 hours. So yeah we don't really have to worry about that as much, thankfully.

Prototype Code

Here, we will briefly summarize what prototype code exists. Using Javascript, we built a dummy chrome extension that sends out dummy requests to the mixnet on an hourly basis and then retrieves those dummy requests. Additionally, we built the full mixnet system with 5 servers and a database server running MongoDB. Finally, we also have a single JS script to simulate active users sending and later retrieving a message.

The prototype is lacking in several ways: it doesn't do the onion encryption due to the overhead load on the servers and the fact that it would take a long time, the dummy messages don’t look similar enough to the real messages, and there is no real active user UI that could be deployed. However, the basic functionality to model the protocol is there, allowing us to use the prototype to accurately run latency tests.



Protocol in Depth

0C - Onion Encryption

Due to the overhead cost, this is something that we didn’t do. However, the general scheme of the onion encryption is to wrap up 3 symmetric keys using RSA public key encryption and also append the keys to the message so that the server can use the symmetric keys to decrypt the specific message. Symmetric key encryption is not possible because the servers do not know who the user is and thus cannot apply a symmetric key.

In order to prevent messages from expanding too much, we chose to use a less secure form of encryption, format preserving encryption. This encryption uses a symmetric key and ensures that the message stays the same length. Not using FPE would lead to rapidly ballooning message size at the benefit of increased security.

1C - Passive Users

When passive users have a chrome instance open, the chrome extension has an alarm that rings every hour. Once that alarm rings, the passive user formulates a random time in the first 20 minutes of the hour to deposit 2 fake messages to the entry servers. The fake messages are onion encrypted using the public keys for all 5 servers and symmetric keys that are generated for each layer of the onion. The IDs of the messages are stored using Chrome’s local storage. Then, passive users randomly choose a time in the last 20 minutes of the hour to retrieve that message with the previously stored IDs. This exactly mirrors behavior for active users.

2C - Active Users

Active users must exchange keys with other active users out of band, or perhaps bootstrap the system with something like Alpenhorn. Anyways, the key exchange is not covered by our version of the project ¯_(ツ)_/¯. However, when a round starts, the active user can do three things with each of their concurrent messages. First, they can attempt to hash to past mailboxes to check them and see if they had missed messages. Second, they can hash to the current mailboxes with other online users to hold concurrent conversations. Finally, they can choose to do nothing, in which case the application would hash to the same mailbox and generate double access cover traffic.

In all of these cases, messages are onion encrypted using the public keys for all 5 servers and symmetric keys that are generated for each layer of the onion. The application chooses a random time in the first 20 minutes of the hour to send a deposit request, containing a request ID, a request message, and request address. Then, in the last 20 minutes of the hour the application chooses a random time to send the request ID to retrieve the user's messages. It is important to note that the message length limit is 850 characters as of now, and anything shorter than that is padded so that all messages are of equal length to avoid leaking further information. Even messages that do not get a valid response (messages that are stored) return a random noise string of 850 characters in length.

3C - Mixnet

In the first 20 minutes of each hour, the mixnet is waiting in the receive phase. If it receives a message during this time, it takes the address and message and pushes it into an array of objects to be sent to the next server. The request ID is stored in a separate array to allow for users to actually get their messages later on. It is also worth noting that the 2 entry servers and 2 mixnet servers are all connected to the MongoDB database server.

Once the receive phase has ended, the servers draw from the Laplace distribution with our chosen parameters to determine how much noise is needed. It also draws from the Poisson distribution to determine the time between individual message deposits and retrievals, although this behavior is not modeled in our prototype. From these distributions, the server calculates how many noise messages need to be retrieved in round 1, 2, 3…. or never, and stores these in separate MongoDB collections. Then, it generates fake noise messages, adds it to the array of real messages, and shuffles all of the messages using a Yates-Fisher shuffle algorithm. After that, the messages are sent to the second server in the chain. The second server does the exact same thing: reshuffles and adds more noise. Finally, when it gets to the mailbox server, the messages are processed with hashmaps. Messages that go to the same mailbox are swapped in place, messages that map to mailboxes that are stored swap with the stored mail, and all other messages are stored for the next 6 rounds. Afterwards, the messages are returned to both mixnet servers, where they are unshuffled and the noise is removed. This process is repeated until all messages are returned to the entry servers. Our trials indicate that this message processing varies in time, between around 10-15 minutes usually. That means that a 20 minute processing window should be sufficient in the vast majority of cases.

In order to avoid leaking timing information, the entry servers must wait until the 40 minute mark until they allow clients to retrieve messages again. Clients then submit their unique IDs, which the server maps to the order in which the messages were originally deposited, in order to retrieve their returned message. At the 60 minute mark, the entire cycle repeats.

Vulnerability Analysis

Due to the fact that we now use 2 server chains, we also increase the attack surface and thus increase vulnerability in a sense. If an adversary takes down either server chain, the system could theoretically still function but without the functionality of retrieving past messages. However, this is still a significant improvement over Vuvuzela since taking down a single server in the chain would result in stopping the entire system.

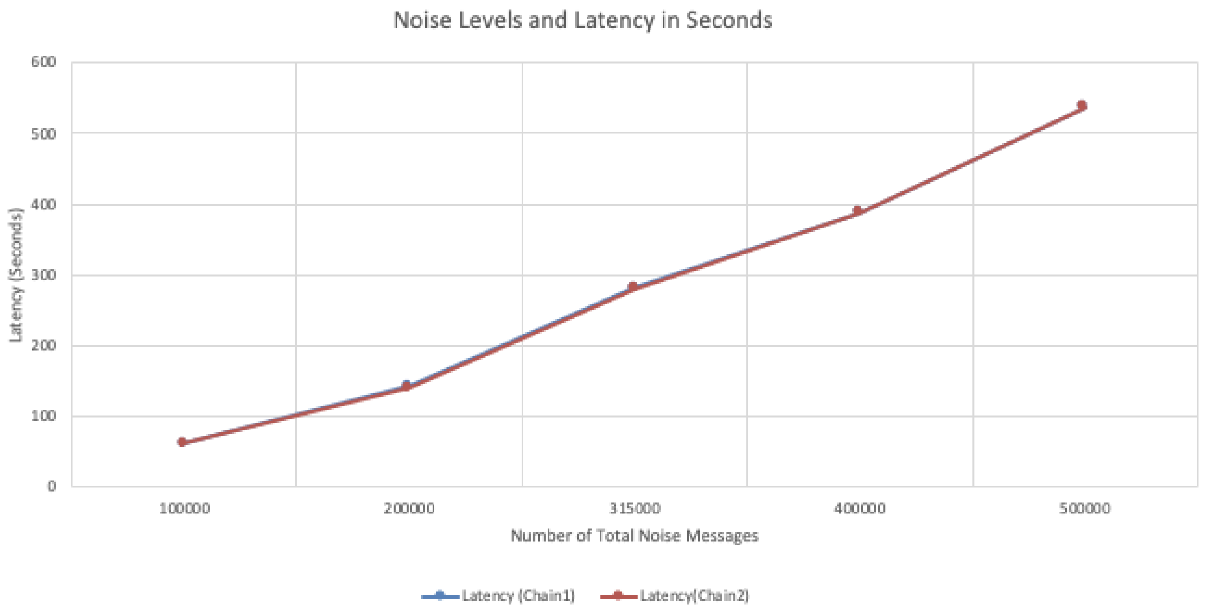
Our server also operates on the anytrust model - if one server is trustworthy, the differential privacy guarantees would still apply. However, if an adversary were to hijack one of the server chains and not add any noise, then we would only be generating half of the necessary noise, destroying our privacy guarantee. However, it should be very apparent to us if that attack were to happen.

Finally, since our system relies on HTTPs post requests, there is also the chance that a vulnerability could arise from previous successful exploitations of HTTPs. At the same time, implementing onion encryption of the messages should entirely mitigate this problem.

Results and Discussion

In our chosen configuration, we aimed to preserve ε , δ differential privacy standards of $\varepsilon < \ln(2)$ and $\delta < 1$ x 10-4 with $k = 50{,}000$ rounds, which equates to approximately 5.7 years of continuous use of the system. We also assume that there will be $c = 100{,}000$ passive users, so we are able to scale down μ to $\mu - 100{,}000$. The improved k-fold composition theorem[3] brings significant improvement to the noise generation, increasing the k rounds with $\varepsilon < \ln(2)$ from 19,300 to 50,000, which is 260% improvement over the previous composition theorem[4].

Furthermore, due to the more distributed server architecture, each side of the server chain only needs to generate 50% of the total noise message, allowing for significant latency decreases in the entry servers and mix servers. However, this comes at the disadvantage of increasing the total attack surface, since all servers are necessary for the functioning of the system. Additionally, we find that the optimal noise level is $\mu = 315{,}000$ and $b = 15{,}000$. Under these parameters, we ran 10 trial rounds on the CrowdMixer prototype for an average latency of 286.68 seconds for message processing, well within the 20 minute window that we are aiming for. We can improve the number of rounds k as well as the differential privacy bounds ε , δ, but at the cost of increased latency. As expected, the latency increases linearly with the number of noise messages, shown in the graph below:



Conclusions

From our work, we can draw several conclusions:

1. Bootstrapping existing anonymous communications systems to increase the size of the anonymity set is simple and does not add much overhead to passive user experience
2. The chrome extension is also useful for making passive users responsible for noise generation, decreasing the load on the servers proportional to number of passive users
3. Adding capabilities for non-ephemeral communications increases the sensitivity linearly, causing a linear increase in the necessary noise
4. A more distributed server topography drastically improves latency and evenly the computational costs
5.

There are several improvements that could explored in future research:

1. Making the server network more distributed as to decrease vulnerability to DDoS attacks
2. Finding a way to decrease the Laplace sensitivity, thus decreasing the necessary noise
3. Adopting this protocol for real-time low latency communications
4. Expanding the number of concurrent user connections

All in all, a very fun project to work on.

Citations

1. J. van den Hooff, D. Lazar, M. Zaharia, and N. Zeldovich. Vuvuzela: Scalable private messaging resistant to traffic analysis. In Proceedings of the ACM Symposium on Operating Systems Principles (SOSP), Oct. 2015.
2. Lazar, D., Gilad, Y., & Zeldovich, N. (2018). Karaoke: Distributed Private Messaging Immune to Passive Traffic Analysis. *OSDI*. Retrieved from https://www.usenix.org/conference/osdi18/presentation/lazar
3. Kairouz, P., Oh, S., & Viswanath, P. (2015). The Composition Theorem for Differential Privacy. *MLR* . Retrieved from http://proceedings.mlr.press/v37/kairouz15.pdf
4. Roth, A., & Dwork, C. *The Algorithmic Foundations of Differential Privacy* (Vol. 9).