Exploring Amazon Simple Queue Service (SQS) for Censorship Circumvention

Michael Pu
University of Waterloo
Waterloo, Ontario, Canada
michael.pu@uwaterloo.ca

Andrew Wang
University of Waterloo
Waterloo, Ontario, Canada
andrew.wang2@uwaterloo.ca

Anthony Chang
University of Waterloo
Waterloo, Ontario, Canada
anthony.chang@uwaterloo.ca

Kieran Quan
University of Waterloo
Waterloo, Ontario, Canada
kieran.quan@uwaterloo.ca

Yi Wei Zhou
University of Waterloo
Waterloo, Ontario, Canada
yw3zhou@uwaterloo.ca

ABSTRACT

The Snowflake censorship circumvention system uses blocking-resistant *rendezvous methods* to connect clients to proxies. This paper describes our experience implementing a new rendezvous method that uses Amazon SQS (Simple Queue Service) and discusses its suitability as a general *signalling channel*. We provide an overview of the implementation followed by some of the design decisions and implementation details that we encountered. The SQS rendezvous method has been deployed in the latest version of Snowflake and the Tor Browser. It has served over 14808 client connections from over 20 countries, including Iran, the United States, China, and Russia. Additionally, we present a country-wise breakdown of users utilizing the existing Snowflake rendezvous methods, as determined by our newly implemented metrics.

KEYWORDS

censorship circumvention, cloud computing, signalling channels

1 INTRODUCTION

There is an increasing effort to develop long-term censorship circumvention techniques that "stand the test of time". That is, even if an adversary gains knowledge of the system's inner workings or makes advances in their surveillance capabilities, it should remain robust and resistant to blocking [7]. This idea aligns with a fundamental idea in cryptography, Kerckhoffs' Principle, which states that "the system must not require secrecy and can be stolen by the enemy without causing trouble" [16].

One such system that attempts to address the objective of "standing the test of time" is **Snowflake** [2]. Snowflake is a censorship circumvention system that uses a large pool of low-cost, temporary proxies that relay traffic between censored clients and a centralized bridge via WebRTC, a common peer-to-peer connection protocol. The sheer volume of proxies that are constantly changing coupled with tunneling through the popular WebRTC protocol [8, 13] makes it highly resistant to blocking attempts by adversaries.

Free and Open Communications on the Internet 2024 (2), 22–26

The process in which clients discover proxies through a central server, known as the *broker*, is called *rendezvousing*. This process is vulnerable to blocking. The different protocols that clients can use to communicate with the broker over blocking-resistant channels are known as *rendezvous methods*, which are described in more detail in 2.1. There are currently two rendezvous methods for Snowflake: *domain fronting* and *AMP cache* [2]. In this paper, we describe our experience with implementing and deploying a third rendezvous method using *Amazon SQS* to increase Snowflake's resilience.

2 BACKGROUND

2.1 Signalling Channels

The process of *rendezvousing* in Snowflake is an instance of a more general problem in censorship circumvention known as *signalling*, *bootstrapping*, or *registration*. We use *signalling channels* to communicate circumvention connection information with clients that lack access to the uncensored internet in the first place [25].

The requirements for these signalling channels are generally less stringent than those for the channels through which actual circumvention traffic flows, as they can tolerate higher latency, lower bandwidth, and sporadic usage [25]. For example, in the case of Snowflake, any method capable of transmitting messages of about 1,500 bytes bidirectionally between the client and broker could work as a rendezvous method [2].

Although the requirements for signalling channels are relatively loose, the channels must be highly resistant to blocking. Vines et al. describes two properties of blocking-resistant signalling channels: *indirectness* (the two endpoints connect to one or more intermediate hosts) and *publicly addressability* (information needed to initiate the connection is known to the adversary) [25]. A common strategy to achieve such resistance is to leverage an existing service that a censor deems too costly to block. By using the idea of "collateral damage", we want to make it harmful to the censor to block the channel by forcing them to also block content with economic or social value [9]. This is the philosophy behind using Amazon SQS as a signalling channel.

2.2 Amazon SQS

Amazon SQS (Simple Queue Service) is a distributed messaging queue for programmatic sending and receiving of messages between web services [21]. It was first proposed to be used as a rendezvous

method for Snowflake in 2018[1], but had not been worked on until now. Its intended use is as a messaging queue for production systems and provides "extremely high message durability" [17].

Since clients communicate with SQS servers at a fixed endpoint (`https://sqs.us-east-1.amazonaws.com`) over HTTPS, it is infeasible for an adversary to differentiate SQS traffic intended for censorship circumvention from genuine Amazon SQS traffic used by other web services. This means that to block this signalling channel, an adversary would have to completely block usage of Amazon SQS.

2.3 Related Works

In addition to the two rendezvous methods available in Snowflake [2, 9], there are a multitude of other signalling channels that have been proposed. Conjure uses DNS requests [3] and refraction networking [10], SWEET [11] uses email messages, MoneyMorph [15] is a provably secure scheme using cryptocurrencies, CoverCast [14] uses live video streams, Camoufler [24] uses instant messaging platforms, and Raceboat [25] experimented with AWS S3, Flickr, and Tumblr.

3 IMPLEMENTATION

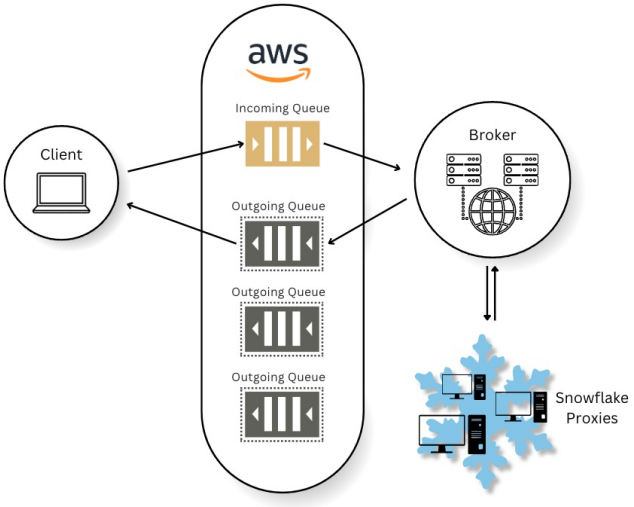

Figure 1: Overview of SQS Rendezvous in Snowflake

The following is an overview of the flow of events between a client and a broker using the SQS rendezvous method illustrated in Figure 1:

(1) When the broker starts up, it listens on the *incoming queue*, which is used for one-way client to broker communication. The URL of this queue is distributed to clients through some out-of-band protocol (e.g. online forums, email). Note that the knowledge of this queue URL by an adversary does not compromise the rendezvous method, as explained in 2.1.

[1]https://gitlab.torproject.org/tpo/anti-censorship/pluggable-transports/snowflake/-/issues/26151

(2) When the client wishes to rendezvous, it sends a message to the *incoming queue* with a unique client ID that it generates along with a Session Description Protocol (SDP) *offer*, which includes information needed to establish a WebRTC connection.

(3) The broker receives the message from the client and it creates a temporary, single-use *outgoing queue*, which is used for one-way broker to client communication. This *outgoing queue* is uniquely identified by the client ID generated by the client. The broker also forwards the SDP offer in the message to an available Snowflake proxy.

(4) When the broker receives an SDP *answer* from the Snowflake proxy indicating that the proxy can accept the connection, the broker will forward the message to the *outgoing queue*.

(5) The client, which has been continuously polling the *outgoing queue*, will receive the message with the SDP answer, completing the rendezvous process.

3.1 Bi-directional Communication

We describe the different ways of implementing bi-direction communication between clients and the broker that were considered.

3.1.1 Using a Single Incoming, a Single Outgoing Queue. A single SQS queue supports having multiple producers (message senders) and consumers (message receivers) where a particular service can be both a producer and a consumer simultaneously [21]. A straightforward approach would be to have two queues: one for incoming messages to the broker and another for outgoing messages from the broker.

However, since there is no way to direct a message to a particular consumer or to retrieve a specific message from the queue, this means that all consumers will be receiving messages intended for all other consumers as well when receiving messages [23]. This raises concerns with efficiency (clients will have to continuously pop from the queue and add messages back until it find a message directed for them) and privacy (clients would be able to see messages directed towards other clients that are also rendezvousing at the same time). The problem with privacy could be solved by using public key encryption [6], but the problem with efficiency still stands.

3.1.2 Using a Single Incoming, Multiple Outgoing Queues. To mitigate these concerns, we decided to create a temporary *outgoing queue* for each client that is identified by a randomly generated 64-bit ID. All clients would send messages to the broker through a single *incoming queue* shared between all clients. To preserve privacy, clients would only have permission to send messages into the queue and only the broker would have permission to read messages from the queue.

Since all messages in each *outgoing queue* would be directed towards a single client, clients can be sure that any message they receive through that queue is intended from them.

After rendezvous is complete, the *outgoing queues* are no longer needed and need to be cleaned up. The broker periodically checks for and deletes *outgoing queues* that were last modified more than a specific number of minutes ago (since rendezvous should not take more than several minutes). This ensures that queues corresponding to completed rendezvous as well as failed rendezvous attempts are both cleaned up.

3.1.3 Using Amazon SQS Temporary Queues. An implementation using the `Temporary Queue Client` provided by Amazon SQS [1] was also considered. This feature allows the client to implicitly map many temporary queues (that are visible to the application) onto a single SQS queue. It is recommended in the AWS Developer Guide to be used for request-response systems, which fits this use case [19]. However, under the hood, all message are still being sent over a single queue which is abstracted as multiple virtual queues that are actually visible to the client. This meant that the concerns with efficiency and privacy described in 3.1.1 still stand and these problems have just been abstracted away from the clients. As a result, we decided to proceed with the implementation described in 3.1.2.

3.2 SQS Details

We discuss some of the intricacies of SQS that we configured or encountered as part of using it as a rendezvous method.

3.2.1 Short vs. Long Polling. By default, SQS queues use short polling to receive messages [18], which means that only a subset of the SQS servers are checked for messages and a response is sent back immediately. On the other hand, in long polling, all of the SQS servers are queried and a response is sent back only when at least one available message is collected, a specified timeout is reached, or a specified maximum number of messages is collected. Long polling is usually preferable over short polling in cases where an immediate response is not necessary [17], which is true for our use case.

We use the maximum allowed value of 20 seconds for the long poll timeout since it minimizes the number of empty responses received (when the queue is empty) and also reduces the number of false empty responses received (when not all of the SQS servers are queried). This helps reduce cost since SQS usage is billed according to the number of requests made [17], as well as the load on the broker spent processing empty responses.

3.2.2 Visibility Timeout and Deleting Messages. When a message is received by a consumer from an SQS queue, the message remains in the queue until it is explicitly deleted [20]. To prevent other consumers from processing the message again, there is a "visibility timeout" on each queue that prevents the message from being received by all consumers once it has been received. Since we did not expect processing messages on the client or broker side to take more than 30 seconds, we decided to simply use the default 30 second "visibility timeout" value.

When receiving messages from a queue, we first process the message before deleting it from the queue. This is to ensure that if the message isn't completely processed (e.g. due to a connection issue), it remains in the queue to be retried at a later time.

3.2.3 Deleting Outgoing Queues. When *outgoing queues* that are no longer needed are deleted as described in 3.1.2, it may take up to 60 seconds for the entire deletion process to complete [22]. This meant that when retrieving a list of queues to delete, that list may include queues that have already been recently deleted, and attempting to delete them again would result in an error.

If we encounter an error when deleting a queue, we simply ignore it. If the error occurred because the queue was already recently deleted, then eventually its deletion process will complete and it will no longer be returned in the list of queues. If the error occurred for some other reason, then we would attempt to delete the queue again the next time the cleanup operation is run again.

3.3 Metrics

Snowflake had existing metrics that track the total number of client polls per rendezvous method. As part of our changes, we added the SQS rendezvous method to these metrics as well as binned counts of the number of clients polling each rendezvous method aggregated by country. These are both now being collected by CollecTor and published publicly[2]. This allowed us to identify what rendezvous methods were being used in each country, which is very useful to know since specific rendezvous methods may work better in certain countries than others.

3.3.1 Country Identification for Clients. To identify which country a client is polling from, we can use the IP address of the client to geolocate it against a database of IP address ranges for each country. However, identifying the IP address of a client in an indirect signalling channel is a challenge. This is because a request from a client will likely make multiple hops in the network before reaching the broker.

For the **domain fronting** and **AMP cache** rendezvous methods [2], we can use the IP addresses in the header of the HTTP request. We use the `realclientip-go`[3] library to retrieve the leftmost address that is not internal or private from the `Forwarded` header [4] first. If this header field is not populated, we then try to use the `X-Forwarded-For` header [5], followed by the IP source address as a last resort.

For the **SQS** rendezvous method, we use the IP addresses from the list of ICE (Interactive Connectivity Establishment) candidates in the SDP offer parsed using the `pion`[4] library. ICE candidates with a higher priority candidate type are preferred since they are geographically closer to the client [12]. Ties are then broken using the ICE candidate's priority property. The IP address of the selected ICE candidate is then used as a "best guess" to geolocate the client.

4 DEPLOYMENT

4.1 Core Functionality

The core functionality[5] [6] of the SQS rendezvous method was merged in February 2024, deployed in Snowflake `v2.9.0`[7], and was released in Tor Browser Stable `13.0.10`[8] and Tor Browser Alpha `13.5a5`[9].

We then announced the new feature on the net4people BBS forum[10] and the Tor forum[11] for testers to begin trying the new rendezvous method. As of 2024-06-17, the usage has remained within

[2] https://metrics.torproject.org/collector.html#snowflake-stats
[3] https://pkg.go.dev/github.com/realclientip/realclientip-go@v1.0.0
[4] https://pkg.go.dev/github.com/pion/ice/v2
[5] https://gitlab.torproject.org/tpo/anti-censorship/pluggable-transports/snowflake/-/merge_requests/214
[6] https://gitlab.torproject.org/tpo/anti-censorship/pluggable-transports/snowflake/-/merge_requests/243
[7] https://gitlab.torproject.org/tpo/anti-censorship/pluggable-transports/snowflake/-/commit/38352b22ade217bd1372772b9cb69f8eff93e919
[8] https://blog.torproject.org/new-release-tor-browser-13010/
[9] https://blog.torproject.org/new-alpha-release-tor-browser-135a5/
[10] https://github.com/net4people/bbs/issues/335
[11] https://forum.torproject.org/t/new-sqs-rendezvous-method-for-snowflake/11713

the limits of the AWS Free Tier (1 million requests per month) and has not yet incurred any costs.

4.2 Bug Fixes and Improved Metrics

After the initial deployment, we received feedback from users and identified some bugs. One problem that was encountered was that the SQS rendezvous method would take much longer than usual (about 3 minutes) to complete the rendezvous in some cases.

The root cause of this issue was because we were reusing the same 64-bit client ID across rendezvous attempts. This meant that if the first attempt to rendezvous failed, subsequent attempts would have to wait for the deletion process for the *outgoing queue* corresponding to the first rendezvous attempt to complete first, as described in 3.2.3. This introduced unnecessary delay and was fixed by using a new 64-bit client ID on each rendezvous attempt[12].

Another problem we encountered was related to the AWS credentials that we distributed publicly for clients to access the Amazon SQS API on the net4people BBS forum[13]. These were likely discovered by AWS Support through GitHub's automated secret scanning[14]. Although the credentials were provisioned with very limited permissions and were intentionally distributed publicly, we were still asked by AWS support to delete them[15]. As a workaround, we now encode the credentials in base64 before distributing them publicly[16].

To better understand which rendezvous methods were being used most frequently in which countries, we implemented improvements to the metrics collected by Snowflake by tracking binned counts (rounded up to the nearest multiple of 8 to preserve privacy) of the uses of each rendezvous method by country[17] as described in 3.3.1. These changes were merged in March 2024, deployed in Snowflake `v2.9.2`[18], and was released in Tor Browser Stable `13.0.12`[19] and Tor Browser Alpha `13.5a6`[20]. Figures 2, 3, 4, and 5 are summaries of these improved metrics that have been collected since 2024-03-21.

5 FUTURE WORK

This paper describes our experience with the initial deployment of SQS as a rendezvous method for Snowflake. In the future, we hope to roll this out as a default bridge provided to users alongside domain fronting and AMP cache. Additionally, we hope to further explore the censorship resistance of SQS, such as against DDOS attacks, TLS fingerprinting, and packet size analysis.

[12]https://gitlab.torproject.org/tpo/anti-censorship/pluggable-transports/snowflake/-/merge_requests/263
[13]https://github.com/net4people/bbs/issues/335#issue-2157478835
[14]https://docs.github.com/en/code-security/secret-scanning/secret-scanning-patterns#supported-secrets
[15]https://gitlab.torproject.org/tpo/anti-censorship/pluggable-transports/snowflake/-/issues/40337
[16]https://gitlab.torproject.org/tpo/anti-censorship/pluggable-transports/snowflake/-/merge_requests/264
[17]https://gitlab.torproject.org/tpo/anti-censorship/pluggable-transports/snowflake/-/merge_requests/258
[18]https://gitlab.torproject.org/tpo/anti-censorship/pluggable-transports/snowflake/-/commit/05a95802c195b1d8a68bb6fe4fa98f12763af519
[19]https://blog.torproject.org/new-release-tor-browser-13012/
[20]https://blog.torproject.org/new-alpha-release-tor-browser-135a6/

ACKNOWLEDGMENTS

The authors would like to thank Cecylia Bocovich for advising this project. We would also like to thank David Fifield, meskio, Nathan Freitas, and Xiaokang Wang for supporting this project.

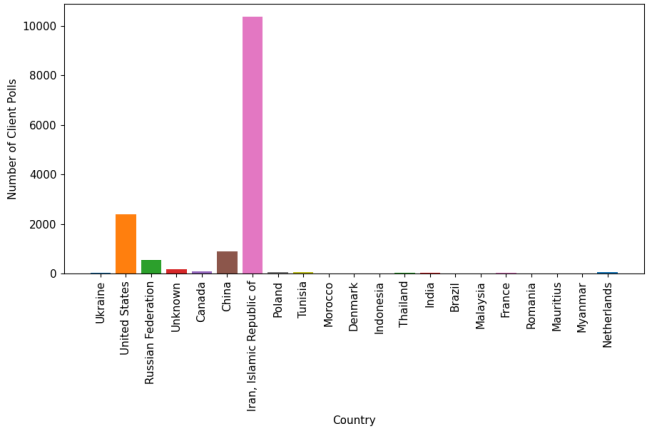

Figure 2: Total number of client polls for the SQS rendezvous method per country since 2024-03-21 until 2024-06-22

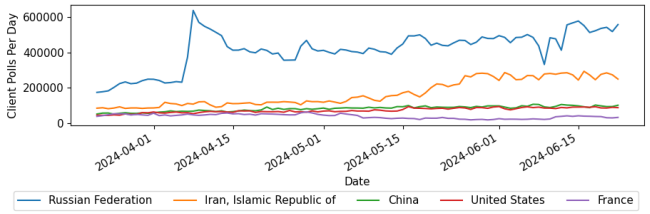

Figure 3: Number of client polls per day for the domain fronting rendezvous method for the top 5 countries

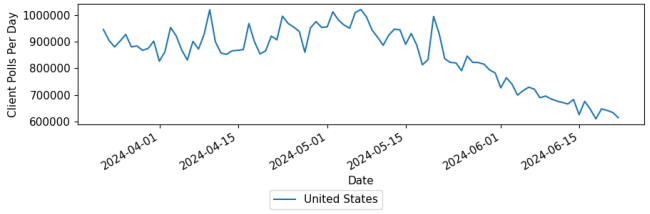

Figure 4: Number of client polls per day for the AMP cache rendezvous method for the top 5 countries

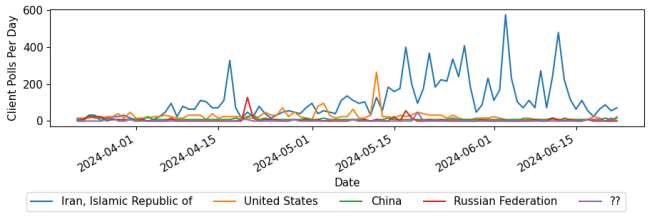

Figure 5: Number of client polls per day for the SQS rendezvous method for the top 5 countries

REFERENCES

[1] awslabs. 2023. *Amazon SQS Java Temporary Queue Client.* Retrieved April 6, 2024 from https://github.com/awslabs/amazon-sqs-java-temporary-queues-client

[2] Cecylia Bocovich, Arlo Breault, David Fifield, Serene, and Xiaokang Wang. 2024. Snowflake, a censorship circumvention system using temporary WebRTC proxies. (2024).

[3] Mingye Chen. 2022. *DNS registration.* Retrieved April 13, 2024 from https://github.com/refraction-networking/conjure/wiki/DNS-registration

[4] Mozilla Contributors. 2024. *Forwarded.* Retrieved April 6, 2024 from https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Forwarded

[5] Mozilla Contributors. 2024. *X-Forwarded-For.* Retrieved April 6, 2024 from https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-Forwarded-For

[6] W. Diffie and M. Hellman. 1976. New directions in cryptography. *IEEE Transactions on Information Theory* 22, 6 (1976), 644–654. https://doi.org/10.1109/TIT.1976.1055638

[7] David Fifield. 2024. Against the "arms race". (2024). https://www.bamsoftware.com/talks/arms-race-foci-2024/ Free and Open Communications on the Internet 2024.

[8] David Fifield and Mia Gil Epner. 2016. Fingerprintability of WebRTC. arXiv:1605.08805 [cs.CR]

[9] David Fifield, Chang Lan, Rod Hynes, Percy Wegmann, and Vern Paxson. 2015. Blocking-resistant communication through domain fronting. *Proceedings on Privacy Enhancing Technologies* (2015).

[10] Sergey Frolov, Jack Wampler, Sze Chuen Tan, J. Alex Halderman, Nikita Borisov, and Eric Wustrow. 2019. Conjure: Summoning Proxies from Unused Address Space. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 2215–2229. https://doi.org/10.1145/3319535.3363218

[11] Amir Houmansadr, Wenxuan Zhou, Matthew Caesar, and Nikita Borisov. 2017. SWEET: Serving the Web by Exploiting Email Tunnels. *IEEE/ACM Transactions on Networking* 25, 3 (2017), 1517–1527. https://doi.org/10.1109/TNET.2016.2640238

[12] Ari Keränen, Christer Holmberg, and Jonathan Rosenberg. 2018. Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal. RFC 8445. https://doi.org/10.17487/RFC8445

[13] Kyle MacMillan, Jordan Holland, and Prateek Mittal. 2020. Evaluating Snowflake as an Indistinguishable Censorship Circumvention Tool. arXiv:2008.03254 [cs.CR]

[14] Richard McPherson, Amir Houmansadr, and Vitaly Shmatikov. 2016. CovertCast: Using Live Streaming to Evade Internet Censorship. *Proceedings on Privacy Enhancing Technologies* 2016 (2016), 212 – 225. https://api.semanticscholar.org/CorpusID:46488544

[15] Mohsen Minaei, Pedro Moreno-Sanchez, and Aniket Kate. 2020. MoneyMorph: Censorship Resistant Rendezvous using Permissionless Cryptocurrencies. *Proceedings on Privacy Enhancing Technologies* 2020 (07 2020), 404–424. https://doi.org/10.2478/popets-2020-0058

[16] Fabien A. P. Petitcolas. 2011. *Kerckhoffs' Principle.* Springer US, Boston, MA, 675–675. https://doi.org/10.1007/978-1-4419-5906-5_487

[17] Amazon Web Services. 2024. *Amazon SQS FAQs.* Retrieved April 6, 2024 from https://aws.amazon.com/sqs/faqs

[18] Amazon Web Services. 2024. *Amazon SQS short and long polling.* Retrieved April 6, 2024 from https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-short-and-long-polling.html

[19] Amazon Web Services. 2024. *Amazon SQS temporary queues.* Retrieved April 6, 2024 from https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-temporary-queues.html

[20] Amazon Web Services. 2024. *Amazon SQS visibility timeout.* Retrieved April 6, 2024 from https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-visibility-timeout.html

[21] Amazon Web Services. 2024. *Basic Amazon SQS architecture.* Retrieved April 6, 2024 from https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-basic-architecture.html

[22] Amazon Web Services. 2024. *DeleteQueue.* Retrieved April 6, 2024 from https://docs.aws.amazon.com/AWSSimpleQueueService/latest/APIReference/API_DeleteQueue.html

[23] Amazon Web Services. 2024. *Receive and delete a message (console).* Retrieved April 6, 2024 from https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/step-receive-delete-message.html

[24] Piyush Kumar Sharma, Devashish Gosain, and Sambuddho Chakravarty. 2021. Camoufler: Accessing The Censored Web By Utilizing Instant Messaging Channels. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security* (Virtual Event, Hong Kong) *(ASIA CCS '21)*. Association for Computing Machinery, New York, NY, USA, 147–161. https://doi.org/10.1145/3433210.3453080

[25] Paul Vines, Samuel McKay, Jesse Jenter, and Suresh Krishnaswamy. 2024. Communication Breakdown: Modularizing Application Tunneling for Signaling Around Censorship. *Privacy Enhancing Technologies* 2024, 1 (2024). https://www.petsymposium.org/popets/2024/popets-2024-0027.pdf