Ten Years Gone: Revisiting Cloud Storage Transports to Reduce Censored User Burdens

Paul Vines
Two Six Technologies
Arlington, Virginia, USA
paul.vines@twosixtech.com

ABSTRACT

We present Skyhook, a cloud storage-based censorship circumvention channel providing highly-available short-lived bidirectional communications to assist users to bootstrap higher-performance secret-based circumvention connections (e.g. bridge and proxy requests). In designing Skyhook, we revisit a prior circumvention system (CloudTransport) and redesign its approach to optimize for signaling use cases. We implement Skyhook as a decomposed channel using the Raceboat framework and demonstrate its flexibility for combining with other circumvention channels.

KEYWORDS

censorship, privacy, cloud storage, network, security

ACM Reference Format:
Paul Vines. 2024. Ten Years Gone: Revisiting Cloud Storage Transports to Reduce Censored User Burdens . In *Proceedings of Free and Open Communications on the Internet 2024.* ACM, New York, NY, USA, 8 pages.

1 INTRODUCTION

Internet censorship has been increasing in prevalence and sophistication over the past decade and more. In response, many different censorship circumvention techniques, or **channels**, have been developed to enable users to freely reach content[8, 11]. The canonical use-case for these circumvention channels has been enabling browsing censored websites, but more generic goals likes enabling connection into the Tor [10] network are also used. These goals require channels to have low-latency, high-bandwidth, and high-availability. These goals have driven development of several channels successfully deployed around the world today [1, 5]. However, these goals have also become constraints on innovation that preclude the development and adoption of some channel concepts despite their potential utility under different circumstances.

In this paper we revisit the CloudTransport [3] channel published by Brubaker et al. in 2014; we modify and extend its core concept in light of alternative use scenarios for circumvention channels. CloudTransport used uncensored cloud storage services to exchange network data and built a channel to provide (relatively) low-latency and high-bandwidth connections aligned with the canonical use case of uncensored web browsing or Tor network access. We argue that low-latency long-lived connections were not the optimal use case for cloud storage-based channels and present

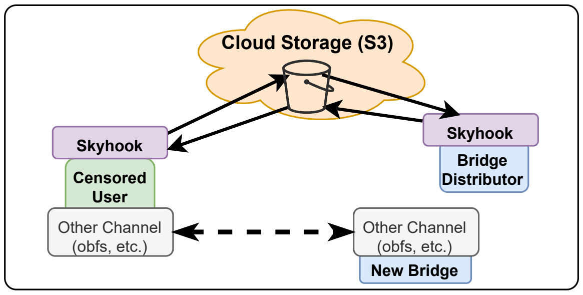

Figure 1: Canonical use-case for Skyhook as a signaling channel to accomplish tasks like bridge distribution. A censored user and a distribution server relay requests and responses through a cloud storage service to bootstrap a higher performance connection that requires secrecy to remain unblocked.

Skyhook as an alternate design and implementation suited for use as a *signaling channel* (see below). **By changing its goals, Skyhook can make different tradeoffs and mitigate several significant drawbacks of CloudTransport's original design: namely, operational costs and requiring paid end-user accounts.**

We follow Vines et al's [12] definition of *signaling channels* as channels that remain available *even when the user has no shared secrets.* A primary use-case for these channels is bootstrapping higher-performance channels that rely on shared secrets, e.g. Tor bridge address distribution [14]. In these contexts, Skyhook only needs to provide transmission of a few messages in each direction and can suffer higher latency without driving away users because it is only used during an initial connection-setup step. We also make use of the Raceboat [12] decomposed application tunneling framework to construct Skyhook in a modular fashion that enables more advanced versions for enhanced threat models.

As a signaling channel, Skyhook represents an alternative to existing channels with similar characteristics such as: meek [7] (domain fronting); raven [13] (email); and PushRSS [15] (push notifications). Skyhook is not necessarily better or worse than these channels, but represents *another* pathway for communication that requires *a different* service to be censored to prevent its use. We argue that a greater diversity of channels is valuable if users can easily move amongst them, because it requires a censor to block many different services or channel techniques before users actually experience increased censorship. This could even dissuade censors from attempting to block *any* channels, knowing there are easy alternatives.

Our contributions are:

- Design of the Skyhook signaling channel conceptually inspired by CloudTransport.
- Formalizing use of unilateral permissioning to enable a bidirectional connection.
- Implementation of Skyhook as modular Raceboat *Transport* and *User Model* components to support flexible combination with other channels and standalone use.
- Evaluation of Skyhook in terms of network security, signaling performance, and cost.

2 DESIGN

We revisit the original CloudTransport approach in light of redefining goals to that of a signaling channel.

2.1 Design Requirements

The requirements of a signaling channel have three major differences from typical circumvention channels:

(1) Required connection lifespan is limited to two messages, ‘client hello‘ and ‘server response‘
(2) Required bandwidth and latency are poorer: roughly 1KB per message and ~1 minute latency.
(3) Must require *no* information known by the client and server but not the censor (i.e. **no shared secrets**) to remain available and secure.

Otherwise, the requirements are much the same as any other circumvention channel: avoid being detectable or blockable by the censor via passive analysis or active attacks at the network or application layers; avoid denial of service (DOS) attacks on infrastructure; **minimize burden on clients and minimize operational costs on servers**. We call out the latter two as particular shortcomings of the original CloudTransport design.

2.2 CloudTransport In Brief

Figure 2 shows the operations of the CloudTransport system. At its core (Figure 2A), CloudTransport transmits data by writing bytes to an object file and then uploading that file to a cloud storage service (e.g. AWS S3) at a specific file name. The receiver side polls to check for the file to exist, fetches it to process the data, and deletes the file in the cloud storage. The deletion signals to the sender that they can write the file again to transmit new data. This process occurs on two files, one for client-to-server connectivity and one for server-to-client connectivity.

2.2.1 *Connection Handshake.*

CloudTransport establishes connections via a two-way handshake (Figure 2B) during which the client selects a random UUID from which the bidirectional data file names are derived (e.g. "client-UUID" and "bridge-UUID") and sends this *hello* request to the server as data written to a default "init" filename. Similarly, the server responds to a default "resp" filename to indicate its side of the connection is ready. For efficiency, multiple connections can be established from a single handshake (i.e. multiple UUIDs can be sent in a single request). There is additionally a "cumuliform" mode which batches and multiplexes all data across a single pair of files, rather than using a different pair for each application TCP connection; this reduces the number of storage service operations necessary to facilitate highly-parallelized network applications like web browsers.

2.2.2 *Accounts.*

CloudTransport's use of a storage service is based on the *client* user providing a paid account for the service and then sharing access credentials (i.e. read/write file permissions only) to the server during the bootstrapping process. The client and server then use the client's account for the read/write operations described above. Interestingly, this creates a type of *proof of work* for the client by requiring they register a storage account *and imposes the financial costs for CloudTransport use on the client.* **The account design mitigates denial of service and sybil attacks, but also imposes a high burden on the client (*i.e. the censored user*) to go through account setup, access credential creation, and provide payment information.**

2.2.3 *Bootstrapping.*

CloudTransport bootstraps client-server relationships (Figure 2C) by having servers create publicly-readable and publicly-writable cloud storage directories, *dead drops*, that prospective clients submit encrypted bootstrap tickets to. The address of the *dead drop* and the public key of the server are assumed to be publicly known; the prospective client constructs a ticket by bundling *the client* account access credentials, *the client* storage directory address, and *the client* public key, and then encrypting the bundle with the server's public key. CloudTransport's dead drop bootstrapping approach is similar to Skyhook's connection design, but we introduce several critical changes to provide signaling channel usage which could not be achieved by the existing CloudTransport design.

2.3 Removing Client Storage Accounts

CloudTransport's design uses cloud storage accounts owned by the client (censored user). This design may be sensible for a circumvention channel to support full network applications since it mitigates some DOS attacks. However, it is a significant burden on users that we can alter the design of Skyhook to avoid.

A trivial change would be to simply flip the account usage design to have servers own the accounts but publicly share access credentials to read/write the cloud storage. This creates a significant DOS attack surface since the censor can use these publicly shared credentials to spam the account.

Instead, we move to an *unauthenticated* access model: each client is an anonymous public user with no credentials at all. The server sets up a publicly writable *object* in the storage service and publicly shares that address. Clients begin a connection by writing a *hello* to the object containing randomly chosen *fetch UUID* and *post UUID* to enable a subsequent connection without authentication as described in the next subsection. Additionally, this *hello* can contain arbitrary application data (e.g. a bridge request, or the start of a key exchange). Figure 3 shows this process using the link-address-oriented Raceboat APIs.

Note that unlike CloudTransport's *dead drops*, this is a *single object* not an entire directory, and it only has public *write* permissions which eliminates potential attacks based on observation of the *hello* requests. We discuss advanced DOS protection approaches further in section 5.

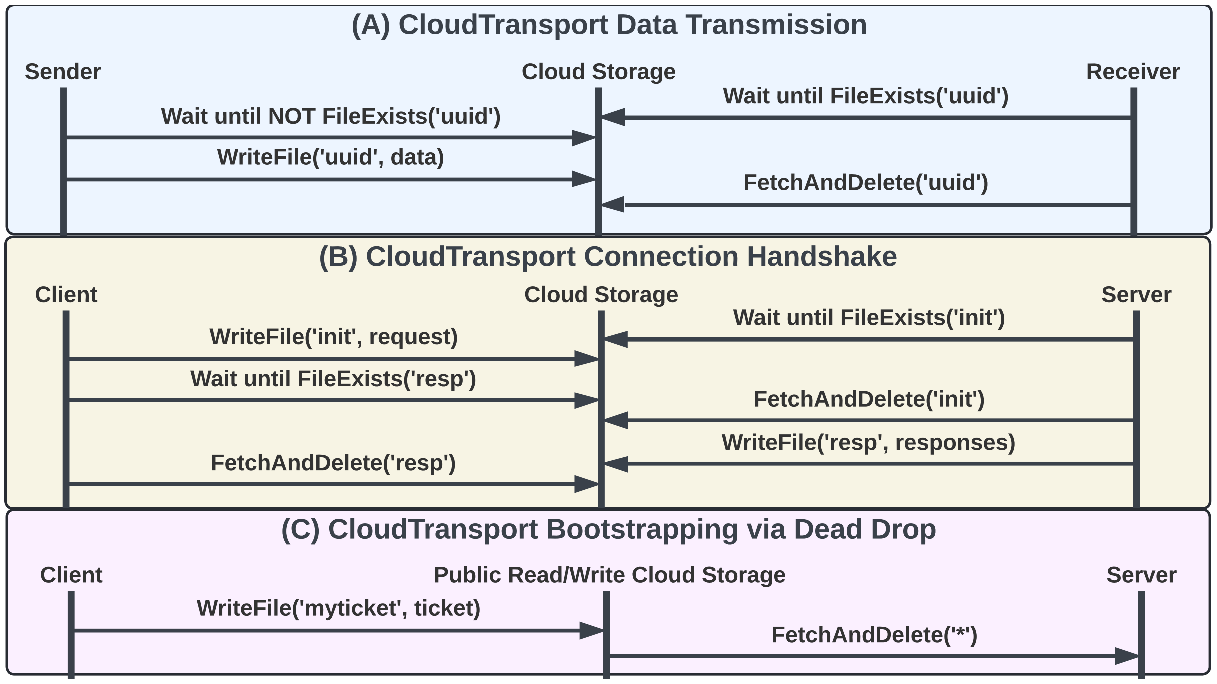
(A) CloudTransport Data Transmission
Sender
Cloud Storage
Receiver
Wait until NOT FileExists('uuid')
Wait until FileExists('uuid')
WriteFile('uuid', data)
FetchAndDelete('uuid')
(B) CloudTransport Connection Handshake
Client
Cloud Storage
Server
Wait until FileExists('init')
WriteFile('init', request)
Wait until FileExists('resp')
FetchAndDelete('init')
WriteFile('resp', responses)
FetchAndDelete('resp')
(C) CloudTransport Bootstrapping via Dead Drop
Client
Public Read/Write Cloud Storage
Server
WriteFile('myticket', ticket)
FetchAndDelete('*')

Figure 2: Summary connection diagrams of the three main "modes" of CloudTransport operation: (A) transmission on an existing connection; (B) establishing a connection via handshake; (C) bootstrapping a client-server relationship via dead drop. Skyhook borrows from but alters and extends each of these.

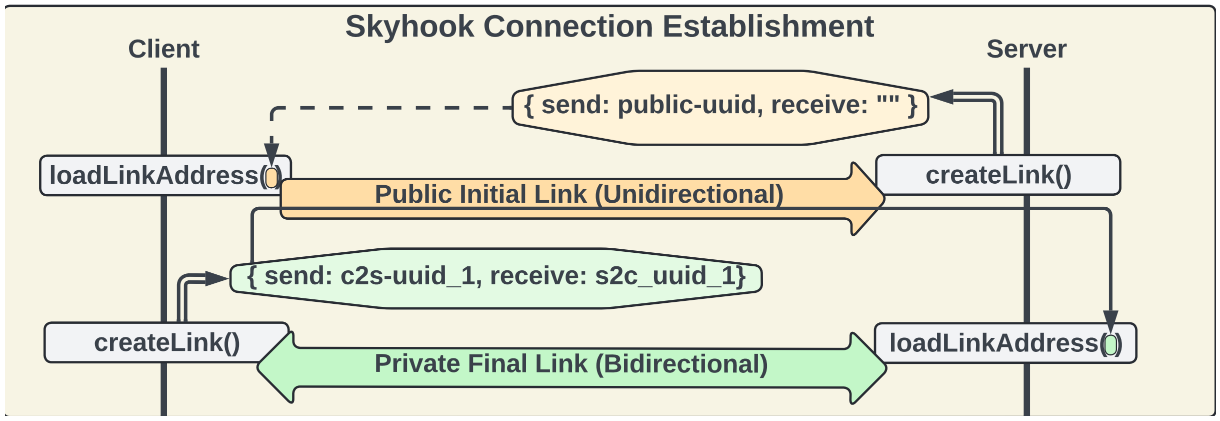
Skyhook Connection Establishment
Client
Server
{ send: public-uuid, receive: "" }
loadLinkAddress()
createLink()
Public Initial Link (Unidirectional)
{ send: c2s-uuid_1, receive: s2c_uuid_1}
createLink()
loadLinkAddress()
Private Final Link (Bidirectional)

Figure 3: Connection diagrams showing how a private bidirectional Skyhook connection is bootstrapped from an initial public Skyhook link. Note that application data (if present) is sent as part of the initial client message, so most signaling uses can be accomplished in a single round-trip.

2.4 Unilateral Permissioning For Bidirectional Connections

Providing only public *write* permissions for the initial *hello* object clearly enables a private *hello* to be sent. However, we must then support subsequent *bidirectional* communications without resorting to an authenticated access model. To do this, we revisit the random UUIDs set at the start of the connection. CloudTransport just uses these to disambiguate connection files. Skyhook uses object-level permissions to set individual UUIDs as publicly readable *or* publicly writable. We use the space of possible UUIDs, and the ability to restrict listing the objects in the directory, to enable a *de facto* access restriction based on the secrecy of the UUIDs: only the client and server know the UUIDs sent in the *hello* request, and thus only they know which objects are available for public reading and writing.

Only the account holder can set the object-level permissions, so the server manages setting both public read permissions for

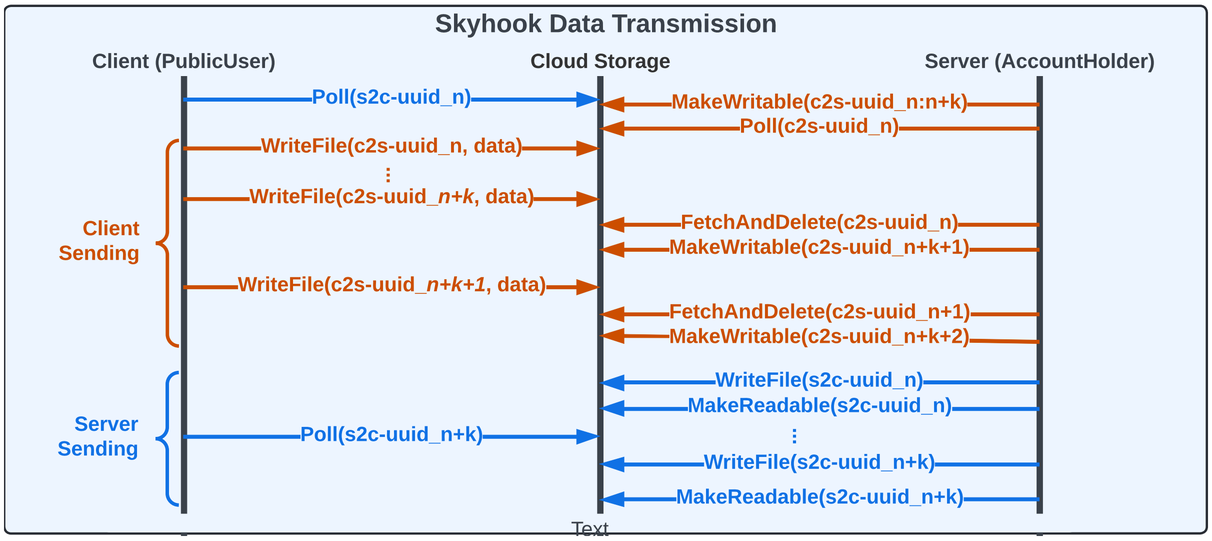
Skyhook Data Transmission
Client (PublicUser)
Cloud Storage
Server (AccountHolder)
Poll(s2c-uuid_n)
MakeWritable(c2s-uuid_n:n+k)
Poll(c2s-uuid_n)
WriteFile(c2s-uuid_n, data)
WriteFile(c2s-uuid_n+k, data)
Client Sending
FetchAndDelete(c2s-uuid_n)
MakeWritable(c2s-uuid_n+k+1)
WriteFile(c2s-uuid_n+k+1, data)
FetchAndDelete(c2s-uuid_n+1)
MakeWritable(c2s-uuid_n+k+2)
WriteFile(s2c-uuid_n)
MakeReadable(s2c-uuid_n)
Server Sending
Poll(s2c-uuid_n+k)
WriteFile(s2c-uuid_n+k)
MakeReadable(s2c-uuid_n+k)
Text

Figure 4: Connection diagrams for an established Skyhook connection transmitting data bidirectionally with multiple send "bursts" in each direction to illustrate the UUID chaining concept. First a series of client-to-server, then a series of server-to-client.

server-to-client objects and public write permissions for client-to-server objects. This unilateral permissioning scheme requires that the server is always aware of which files should and should not have public permissions.

2.4.1 UUID Chains. CloudTransport used a single constant UUID for each direction of its connection and used the deletion of the file upon fetching by the receiver as a signal to the sender that another message could be sent. Once again, this is sensible in a low-latency synchronous use case but not the only design available. Instead, Skyhook uses a *chain* of UUIDs in each direction of connectivity, where the next UUID is derived from a hash of the prior UUID.

The server pregenerates a "buffer" of UUIDs in each direction based on the *hello* request and sets their public permissions. This enables either side of the connection to send up to the buffer-length in separate messages before the other side needs to actively receive them. Each time the server sends or receives a message, it extends the relevant buffer by an additional UUID (by setting a public read or write permission on it). Figure 4 shows this operation and the asymmetry of operations between the public and account holder sides of the connection. From the client, we use an *implicit acknowledgment*: if the client sends a message, the server assumes it has already fetched any prior posted UUID on the server-to-client chain, and can unpermission and delete prior messages. For specific application loads, this scheme may be suboptimal and a variety of other options exist if application tailoring is desired. For the signaling use case, there are so few messages in each direction that the chains never actually need to be extended in-practice.

3 IMPLEMENTATION

We implemented Skyhook as a set of C++ components within the decomposed application tunneling framework provided by Raceboat and refer the reader to that publication [12] for context on subsequent sections. Figure 5 shows this component breakdown for the client and server implementations. We chose to use a compiled language to minimize the size and dependency footprint of Skyhook, particularly for the client users; the client plugins total just 528KB.

3.1 Transport Components

The core of Skyhook logic and its cloud storage interactions are handled by the *transport* components. There are two variants: the *PublicUser* transport which is used by the client and makes no use of account credentials; and the *AccountHolder* transport used by the server that manages object permissions corresponding to connections. Both components execute *FETCH* and *POST* actions from the User Model timeline (see below). Both also specify no constraints on the type of content they require for *POST*, since a cloud storage object can contain any data (including raw bytes) and Skyhook's baseline threat model assumes TLS prevents the censor from inspecting the actual content in-transit. However, Skyhook can be paired with arbitrary Encoding components to mitigate threat models that involve content inspection by the censor without modification to any of the Skyhook code, we discuss this further in Section 5.

3.1.1 PublicUser Transport. The PublicUser Transport executes *FETCH* and *POST* actions by making HTTPS requests to a specified AWS S3 object UUID. The URL is intentionally built as $https://s3.<region>.amazonaws.com/<directory>/<uuid>$ such that, without breaking TLS, a network observer can only observe a request to the regional S3 subdomain. Thus, blocking access to this URL would require blocking all HTTPS requests to an entire S3 region. This is also why we use a CURL request, rather than an API library like AWS-CLI, because the latter makes requests to an account-specific subdomain that could be blocklisted if known by the censor.

The PublicUser Transport supports two methods of "link establishment." Firstly, it can *load* a *link-address* that contains the starting fetch and post UUIDs and essentially "jumps into" a connection immediately. Alternatively, it can *create* a link, including generating a *link-address* that specifies the fetch and post UUIDs. In the create case, this link-address must then be transmitted to an AccountHolder Transport (i.e. a server) to load it before the specified object permissions will actually be enacted. Until then, the PublicUser cannot actually *use* the link it has created; in particular it cannot try to post any messages ahead of transmitting the link-address because the AccountHolder has not yet made the post UUIDs available for public writing.

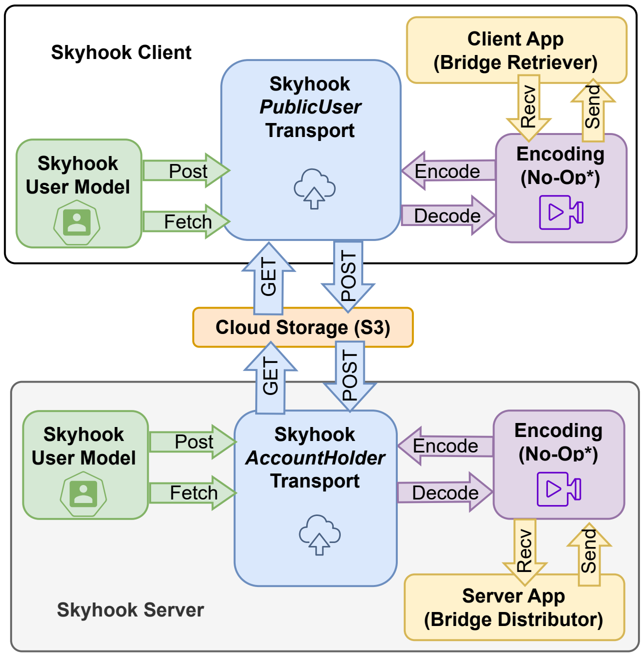

Figure 5: Architecture of the Skyhook client and server implemented in the Raceboat decomposed channel framework. A generic user model component is used by both, but distinct *PublicUser* and *AccountHolder* transport variants. Skyhook uses a No-Op encoding under the baseline adversary model, but *can* use any encoding since cloud storage supports arbitrary content types.

3.1.2 AccountHolder Transport. The AccountHolder is effectively a superset of the PublicUser Transport. It executes *FETCH* and *POST* actions in the same way, but with additional steps associated with maintaining the object permissions. Each time the AccountHolder Transport *FETCH*es a new message, it removes the public write permissions from the UUID, deletes the file, and extends the buffer of publicly writable UUIDs by one. It *also* removes the read permission and deletes each file it *wrote to* before this fetched file was written to, under the assumption that the PublicUser will read files before writing new ones. Each time the AccountHolder Transport *POST*s a new message, it extends the buffer of publicly readable UUIDs to include the new file.

3.2 User Model Components

The User Model components of Skyhook generate sequences of *FETCH* and *POST* actions for each link that is established. The timing of these sequences is designed to avoid excessive delays while also limiting the amount of unnecessary requests (e.g. requests for objects that have not been written yet). We settle on a 1-second polling rate and an on-demand sending model. The former means a *FETCH* action is scheduled for each link every second; the latter means that *POST* actions are only scheduled if there is data to send, and they are scheduled for as-soon-as-possible execution.

The User Models are simple generators of *FETCH* and *POST* actions to avoid excessive costs while providing sufficient performance for signaling channel use cases; they are not designed for providing security against user behavior analysis attacks. We argue that the limited duration and infrequent execution of signaling channel uses makes these types of attacks in a common censorship circumvention scenario unlikely. However, if a specific use/threat model for Skyhook *does* include such attacks (i.e. if Skyhook were planned to be used for longer-lived continuous connections like the original CloudTransport, see Section 5) then the User Models can be extended as appropriate to disguise Skyhook use at some performance cost. For example, the GAN-based approach used by Wails et al. [13] could be used to produce realistic timelines of actions based on real world traffic traces. We stress that these changes are formally abstracted from the Transport code via the decomposed framework, such that no Transport code modifications would be required.

4 EVALUATION

Skyhook is a fully functioning channel implementation for the Raceboat framework (technically two: the PublicUser and AccountHolder variants) and can be used in any way within that framework. However, our specific motivation for its use is as a signaling channel to support operations like retrieving a bridge address or bootstrapping another channel from only publicly available information. We demonstrate Skyhook's utility and flexibility in two configurations:

(1) Skyhook-only bridge request & response scenario
(2) Skyhook + another signaling channel bridge request & response

4.1 Skyhook in Isolation

In this scenario, we run Raceboat with Skyhook as the only channel; there is a single initial Skyhook link created by the server and its link-address is publicly shared out-of-band with the client (and assumed known by the censor). This initial *public* link is used for the *hello* request by the user which *both* contains a request for a bridge at the "application" layer (i.e. above Raceboat) *and* an in-band link-address for a new Skyhook link that provides a new, *private*, link for the server to respond on.

Table 1 shows the latency for this bridge request operation across varying AWS S3 regions with the client and server run from private devices nearest the U.S. East region.

4.2 Skyhook in Combination

To demonstrate the flexibility of using Skyhook, we conduct the same bridge request & response operation with two other Raceboat-supporting channels: EmailBase64 and FlickrJEL. The former uses automation to send and/or receive emails and simply base64-encodes the request data; the latter uses JEL [4] to steganographically encode the request into a JPEG image and then transmits it via a public Flickr post. Table 2 show results for swapping these with Skyhook for *both* the initial client-to-server (public) link and the responding server-to-client (private) link.

We do not show these results to demonstrate Skyhook's superiority to other signaling channels, but to demonstrate its flexibility to mix-and-match with other channels within the Raceboat framework. Depending on the threat model and user environment, some channels may be functionally worse (or nonfunctional) as public request links, or private response links, or as either but never both.

5 EXTENDED SECURITY MODEL

Our initial design requirements (Section 2.1) used a broad and generic threat model to represent the censor: we assumed that TLS remained intact, HTTPS queries to cloud storage services were innocuous, Skyhook would be used for short-lived signaling connections, and no extensive analysis of user behaviors was being conducted. Essentially, we claim Skyhook can hide within the noise of a busy network environment without providing a concise detection or blocking rule for the censor, short of blocking access to the cloud storage service altogether.

However, such a generic threat model is not always valid. **In this section, we highlight several attack surfaces along with directions Skyhook can be extended to mitigate them.** Figure 6 lays out a hypothetical scenario of adversary advances and Skyhook responses described in the following subsections. We posit that the modular construction of Skyhook lends itself to this sort of agility against real world censor tactics.

5.1 Content Inspection Attacks

Our standard threat model assumes TLS encryption is intact so we do not bother encoding the content of Skyhook data as anything other than binary blobs since it is unobservable to the censor. However, for threat models that cannot assume intact transport security (e.g. TLS is subject to man-in-the-middle attacks or is blocked outright) Skyhook can be combined with an arbitrary other Encoding component [12] that provides steganographic encoding of message data. Since a wide variety of content might plausibly be stored in the cloud, a wide variety of Encodings might be reasonable: e.g. Raceboat already has support for one steganographic image encoding [4]; Format Transforming Encryption (FTE) [6] could also provide a variety of encoding options like static HTML content or natural language text [2].

Due to implementation of Skyhook in the decomposed Raceboat framework, addition of an Encoding requires no modification of the Transport code itself. Modifying a single manifest file and providing the plugin that contains the Encoding component at runtime is all that is necessary.

Table 1: Latency of a Lox-based Bridge Request + Invitation Redemption (346B request + 1.3KB response) using Skyhook across different AWS S3 regions, client and server run in eastern US.

S3 Region	Total Time
US (Virginia)	7.926s
US (California)	11.763s
Sweden	14.618s
India	19.361s
Japan	18.611s

Table 2: Latency using Skyhook (in US, Virginia) and another signaling channel, with one used as the request channel and the other as the response channel. Same 346B + 1.3KB Lox-based Bridge Request + Invitation Redemption as Table 1

Req. Channel	Resp. Channel	Total Time
Skyhook	EmailBase64	24.54s
Skyhook	FlickrJEL	90.10s
EmailBase64	Skyhook	14.51s
FlickrJEL	Skyhook	65.13s

5.2 User Behavior Pattern Attacks

Skyhook can theoretically be detected based on behavior patterns: both analysis of individual Skyhook connections and aggregate analysis of all connections a user makes over time *could* provide a pattern that is separable from benign user behaviors using e.g. machine learning classifiers. Wails et al. [13] illustrated this type of attack on uses of email-based circumvention channels that create anomalous email sending patterns.

Connection pattern attacks are inherently about detecting the "shape" of the censored application traffic influencing the "shape" of the circumvention connection: this is a significant attack surface for a long-lived Skyhook (or CloudTransport) connection that is tunneling a synchronous application. However, we posit that it is much less feasible for short-lived signaling connections. E.g. in a bridge request scenario, there is a single HTTPS POST and HTTPS GET request, which we argue would correspond to many legitimate uses of cloud storage services. In contrast, using Skyhook to tunnel a livestream upload could create a distinctive pattern of traffic (i.e. constant rate of small POSTs) that might be separable from all "normal" cloud storage usage.

In summary, we consider these types of pattern attacks to be outside the scope of our core threat model, potentially mitigated by our intended use case, and highly dependent on knowledge of a *specific* more advanced censor. Without a concrete basis for this censor, it is not effective to devise defenses. *However*, due to Skyhook's decomposed implementation, pattern defenses can be developed and deployed without altering (or even rebuilding) the core plugin code. E.g. given some data for the censored environment, a data-driven approach like Raven's [13] GAN-based user model generation could be applied to create a user model that restricts Skyhook's connection traffic to be within the local norm.

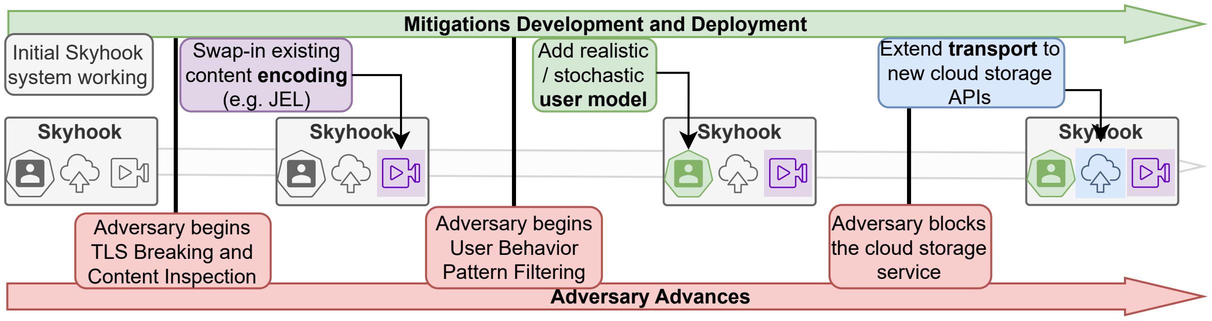
Mitigations Development and Deployment
Initial Skyhook system working
Swap-in existing content **encoding** (e.g. JEL)
Add realistic / stochastic **user model**
Extend **transport** to new cloud storage APIs
Skyhook
Skyhook
Skyhook
Skyhook
Adversary begins TLS Breaking and Content Inspection
Adversary begins User Behavior Pattern Filtering
Adversary blocks the cloud storage service
Adversary Advances

Figure 6: Hypothetical timeline showing deployment of Skyhook which provokes adversary advances to block it. These advances are then mitigated by targeted extensions to Skyhook. The modular nature of Skyhook leaves the majority of its code untouched by any given mitigation.

5.3 Cloud Storage Access Fingerprinting Attacks

Skyhook uses HTTPS cURL requests to fetch and post content to cloud storage. We specifically use this "raw" type of request to enable directing the request to a generic regional subdomain rather than an account-associated subdomain; the account-associated subdomain is assumed known by the censor and so could be blocked without disrupting all access to the storage provider.

However, there may be network environments where this is the wrong trade-off: if the censor can determine that this style of request is suspicious and safe to block, then Skyhook should be modified/extended to use an alternative access method. Once multiple modes are supported, this could be easily set on a per-link basis to enable a universal Skyhook channel that can operate in either case (including heterogenous mixes, if applicable).

5.4 Denial of Service (DOS) Attacks

There are several DOS attack surfaces in the Skyhook implementation. Object-level permissioning on pseudorandom object UUIDs prevents DOS attacks by third-parties *on private links*: only the client and server of the connection know the chain of UUIDs that can be read and written publicly.

However, in the case of a *public* link where the initial UUID is assumed known, or if a client turns out to be malicious, a DOS attack surface exists. The censor could write repeatedly and/or large objects to a publicly writable UUID. In assessing DOS threats we must distinguish two types of resources that can be exhausted: actual processing and the *cost* of operating the cloud storage service.

5.4.1 Compute DOS. Processing-based DOS attacks are less of a threat to Skyhook than some other circumvention channels because a major cloud storage provider will actually bear the brunt of the attack: **e.g. the censor will be trying to launch a DOS attack on AWS S3, not a privately hosted webserver.** Large numbers of *hello* requests could also slow the processing of legitimate requests by the Skyhook server, but the amount of work conducted per-request is small and the Skyhook server can be run *within* the hosting environment of the cloud storage provider to give it an inherent advantage in processing requests versus the external censor in making the requests. A concerted attack by the censor on the cloud storage service approaches the same effect as blocking the storage service in the first place. We assume that imposes unacceptable collateral damage on the censor's economic and social interests, but ultimately is a choice the censor can make (just as cutting off Internet access entirely is).

5.4.2 Monetary DOS. Exhausting Skyhook server infrastructure *monetarily* is a different case. We analyze potential monetary DOS attacks based on AWS S3 pricing [9], assuming other cloud providers follow similar billing approaches. First, data transfer *into* storage is free, so there is no cost for the censor uploading large amounts of data initially. Objects *in* storage are deleted after processing, and additional rules for automated deletion after time are also supported, this should minimize costs for storage to sustainable levels that would raise the bar for the monetary DOS attack to become a computational/network DOS attack on the cloud provider.

There are no publicly-known read links in the Skyhook use-cases formulated in this paper, so any publicly readable UUID is known only to a client that has successfully started a connection with a server. Assuming a malicious client connection *is* established, the client can maliciously fetch readable objects to impose costs for transferring data *out* of storage. These are free up to the first 100 GB/month, and $0.05-0.09 GB/month after. In the context of our 1.3KB bridge response, this equates to more than 15 *million* signaling connections per-dollar. Furthermore, since these read operations depend on establishing a connection, there is an opportunity for the server to execute DOS-protection logic to prevent an undetermined number of readable files. Naturally, cost mitigations like slowing the rate of connection acceptance or adding some "proof of work" on the client can itself provide the original denial of service the censor aimed for.

5.4.3 API Gateways Defenses. If DOS attacks cannot simply be *sustained* by the cost and compute infrastructure, there is another design mitigation available: shifting resource access to a managed API gateway enables inclusion of WAF-like rules such as tracking connection source IP addresses and blocking abusive hosts. Thus, a API gateway-guarded Skyhook link could detect and block censor attempts to exhaust resources with reads and writes. The trade off, however, is the gateway requires directing requests to a unique subdomain, which could become the target of censorship. The ephemeral nature of the domain *does* enable a potential flux-based defense which raises the complexity of censorship by requiring continuous update of blocking rules. However, since the address *must* be public, there is no *fundamental* way to prevent censor blocking of the subdomain.

6 CONCLUSION

In this paper we have revisited the concept of censorship circumvention using cloud storage services proposed in CloudTransport. We find shortcomings in its original use case and the design oriented towards it: supporting generic and long-lived networked application support. We instead reorient the cloud storage concept towards a more suitable and increasingly important use case: providing *signaling channels* for short-lived connections with no pre-shared secrets in order to bootstrap higher-performance circumvention connections. We devise a new channel, *Skyhook*, within the decomposed application tunneling framework provided by Raceboat: we redesign several important aspects of CloudTransport to support greater resilience and usability, and demonstrate the suitability of Skyhook for use as a signaling channel to bootstrap other censorship circumvention connections (e.g. Tor bridges) without secret information.

ACKNOWLEDGMENTS

This material is based upon work supported by the AFRL-RI and DARPA under Contract No. FA8750-19-C-0501. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the AFRL-RI and/or DARPA. “A” (Approved for Public Release, Distribution Unlimited).

REFERENCES

[1] Yawning Angel. 2023. obfs4: The obfourscator. https://gitlab.com/yawning/obfs4 Access on 5/30/2023.

[2] Luke A Bauer, James K Howes IV, Sam A Markelon, Vincent Bindschaedler, and Thomas Shrimpton. 2021. Covert Message Passing over Public Internet Platforms Using Model-Based Format-Transforming Encryption. *arXiv preprint arXiv:2110.07009* (2021).

[3] Chad Brubaker, Amir Houmansadr, and Vitaly Shmatikov. 2014. CloudTransport: Using Cloud Storage for Censorship-Resistant Networking. In *Privacy Enhancing Technologies Symposium*. Springer. https://petsymposium.org/2014/papers/paper_68.pdf

[4] Chris Connolly. 2015. libjel – JPEG Embedding Library. https://github.com/SRI-CSL/jel Access on 5/30/2023.

[5] Tor Documentation. 2023. Snowflake: pluggable transport that proxies traffic through temporary proxies using webrtc. (2023). https://trac.torproject.org/projects/tor/wiki/doc/Snowflake

[6] Kevin P. Dyer, Scott E. Coull, Thomas Ristenpart, and Thomas Shrimpton. 2013. Protocol Misidentification Made Easy with Format-Transforming Encryption. In *Computer and Communications Security*. ACM. https://eprint.iacr.org/2012/494.pdf

[7] David Fifield, Chang Lan, Rod Hynes, Percy Wegmann, and Vern Paxson. 2015. Blocking-resistant communication through domain fronting. *Privacy Enhancing Technologies* 2015, 2 (2015). https://www.icir.org/vern/papers/meek-PETS-2015.pdf

[8] Sheharbano Khattak, Tariq Elahi, Laurent Simon, Colleen M. Swanson, Steven J. Murdoch, and Ian Goldberg. 2016. SoK: Making Sense of Censorship Resistance Systems. *Privacy Enhancing Technologies* 2016, 4 (2016), 37–61. https://murdoch.is/papers/popets16makingsense.pdf

[9] s3-pricing 2024. Amazon S3 pricing. https://aws.amazon.com/s3/pricing/ Accessed on 4/15/2024.

[10] tor 2023. Tor. https://torproject.org/ Accessed on 5/30/2023.

[11] Michael Carl Tschantz, Sadia Afroz, Anonymous, and Vern Paxson. 2016. SoK: Towards Grounding Censorship Circumvention in Empiricism. In *Symposium on Security & Privacy*. IEEE. https://www.eecs.berkeley.edu/~sa499/papers/oakland2016.pdf

[12] 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

[13] Ryan Wails, Andrew Stange, Eliana Troper, Aylin Caliskan, Roger Dingledine, Rob Jansen, and Micah Sherr. 2022. Learning to Behave: Improving Covert Channel Security with Behavior-Based Designs. *Proceedings on Privacy Enhancing Technologies* 3 (2022), 179–199.

[14] Qiyan Wang, Zi Lin, Nikita Borisov, and Nicholas J. Hopper. 2013. rBridge: User Reputation based Tor Bridge Distribution with Privacy Preservation. In *Network and Distributed System Security*. The Internet Society. https://www-users.cs.umn.edu/~hopper/rbridge_ndss13.pdf

[15] Diwen Xue and Roya Ensafi. 2023. The Use of Push Notification in Censorship Circumvention. *Free and Open Communications on the Internet* (2023).