Communication Breakdown: Modularizing Application Tunneling for Signaling Around Censorship

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

Samuel McKay
Two Six Technologies
Arlington, Virginia, USA
sam.mckay@twosixtech.com

Jesse Jenter
Two Six Technologies
Arlington, Virginia, USA
jesse.jenter@twosixtech.com

Suresh Krishnaswamy
Two Six Technologies
Arlington, Virginia, USA
suresh.krishnaswamy@twosixtech.com

ABSTRACT

We present Raceboat, a novel framework for developing and managing censorship circumvention channels. The Raceboat framework simplifies using signaling channels for low-bandwidth and/or latency-tolerant tasks like bridge distribution and authentication. We further develop a novel decomposition of application tunneling circumvention channels that is well suited to signaling channel usage. This decomposition enables modular components that are reusable across varied channels. We demonstrate the flexibility and extensibility of Raceboat for signaling uses by mixing-and-matching seven different channels.

KEYWORDS

censorship, networks, privacy, application tunneling

1 INTRODUCTION

Censorship of user access to the internet is growing increasingly sophisticated and common across the globe. In response, many approaches to circumventing this network censorship have been developed. To combat network-level censorship (blocking at the level of individual network connections) a number of censorship circumvention channels have been developed over the past years to facilitate users reaching applications from behind their censors' firewalls. The canonical use-case is enabling browsing censored websites, but enabling instant messaging, video streaming, and other arbitrary internet-connected applications is also prevalent. To serve this use case, most of these circumvention channels emphasize low latency and high bandwidth, and so often rely on a direct IP connection to a circumvention channel proxy server or, in Tor parlance, a *bridge* (we will use bridge throughout this paper to avoid ambiguity, although we do not assume the server necessarily connects the user to the Tor network). This creates a second-order problem: censors can enumerate bridges based on a variety of techniques and then block any connections to those IPs. Creating new bridges or changing their IPs is easy, but the problem lies in how to communicate these new bridge-addresses to users that have no connectivity to the uncensored internet (because their bridges were blocked).

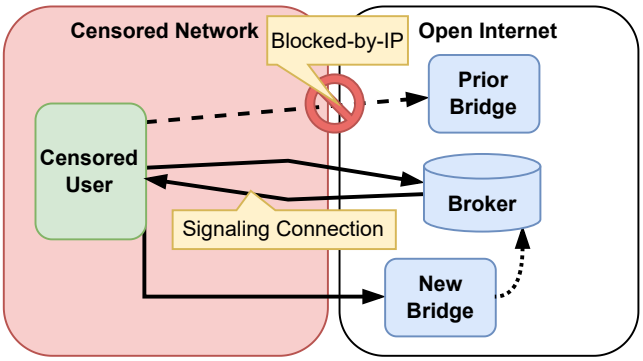

Figure 1: Censors can block connections to bridges based on IP address; the addresses of new bridges are distributed to users via a *broker*. However, the user then needs a connection to this broker that requires no shared secrets.

This is known as the bridge distribution problem [34] or, more generally, a *rendezvous problem* (illustrated in Fig. 1): users need a circumvention channel that remains available even when the user and the adversary have all the same information (i.e. no secrecy is required to prevent blocking). We refer to channels that can serve this purpose as ***signaling channels***. A number of channels have been developed for this purpose; the most prominently deployed is Domain Fronting [14] but prototype systems like Raven and SWEET that use email [37], CloudTransport [4] that uses cloud-storage, and even MoneyMorph that uses cryptocurrencies [24], have been demonstrated.

A necessary aspect of signaling channels is to remain available when the censor is employing IP-blocking; therefore these channels rely on the client-side of the channel (the user's device) making IP connections to a third-party server that services some non-circumvention use-case the censor is reticent to block. We refer to such channels as *indirect* since they avoid direct IP connections between channel endpoints. Indirectness implies the channel must make a legitimate connection on the protocol served by the intermediary server and the messages are transmitted as content

Proceedings on Privacy Enhancing Technologies 2024(1), 465–477

https://doi.org/10.56553/popets-2024-0027

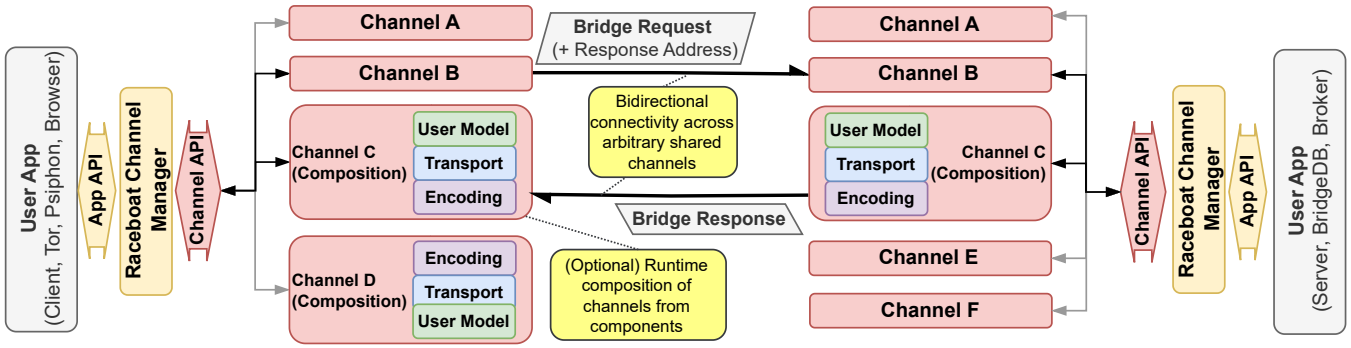

Figure 2: Example of a User App using Raceboat to perform a bridge request: the User App uses Raceboat's App API to specify a request-response mode, the request message, and the channels to use. Raceboat creates bidirectional connectivity with multiple potentially unidirectional channels managed via its Channel APIs. The bridge request traverses one of these channels, along with an address to receive the response on an entirely separate channel. Each channel can be any type of channel that implements the Channel API, or application tunneling channels composed at-runtime from reusable modules (see Section 4).

within that protocol. Therefore, all signaling channels of this sort fit within the *application protocol tunneling* [19] approach: e.g. Domain Fronting tunnels within the TLS protocol; Raven within email; CloudTransport within cloud-stored documents; and MoneyMorph within cryptocurrency transactions. In each case the application protocol is unaltered but the content conveyed by the protocol is an encoded version of the signaling channel's messages, indiscernible to the censor either because of application-layer encryption preventing inspection or through the use of steganography.

Development of signaling channels has often been *ad hoc*; e.g. TorBrowser's Moat functionality uses meek [14] (domain fronting) to fetch bridge addresses and is packaged as a Pluggable Transport [25]. However Snowflake, another Pluggable Transport that uses short-lived proxies, also needs rendezvous functionality and thus re-packages its own implementation of domain fronting within its codebase. The alternative signaling channels listed above are each implemented as standalone prototypes and, at best, adopt the Pluggable Transports (PT) specification as their application interface. Even the use of the PT interface is a hindrance because it is designed for continuous socket-like connections (and, in its widely adopted original interface, is explicitly supposed to expose a SOCKS-proxy interface). In contrast, signaling channel tasks are often much lighter-weight - e.g. sending a single client request and receiving a single response. Additionally, there is no reason why the same channel needs to be used for both directions of communication (indeed some signaling use cases may only need a unidirectional push of information). However, existing solutions always seek to function as standalone channels providing bidirectional connectivity even when this does not suit the channel.

In the case of application protocol tunneling signaling channels, development is also hampered by an *ad hoc* one-off approach. Looking across many such systems we observe that there are marked similarities in internal functionality that hint at opportunities for abstraction, modularization, and re-use.

Fig. 2 illustrates the end-to-end usage of the Raceboat system and highlights our contributions: we formalize the properties and functionalities of signaling channels and then design and implement the *Raceboat framework*[1] to provide a more flexible interface for signaling channels that supports a variety of use cases, including seamless mixing-and-matching of unidirectional channels. Additionally, we formalize a *decomposition of application protocol tunneling channels* into components and design and implement the *decomposed application tunneling framework* for dynamically assembling these components to synthesize functional channels. These contributions can both drastically increase re-use of research products and decrease developer effort when extending the existing capabilities of channels.

Our contributions in this paper are as follows:

- Formalization of signaling channel functionalities
- Implementation of a generic multi-channel interface for multiple signaling use cases
- Formal decomposition of application tunneling channels into modular components
- Implementation of a framework for runtime construction and use of application tunneling channels based on modular components
- Implementations of exemplar components providing varying signaling functionality over email, AWS S3, and redis services

2 BACKGROUND

In this section we provide background on the censorship circumvention layers Raceboat innovates in: the bridge distribution problem; the generalized type of circumvention channels needed; and application tunneling.

2.1 Bridge Distribution and Signaling

Most circumvention channels are developed with an assumption that some secret information can be exchanged out-of-band and assumed unknown by the censor. These "shared secrets" can vary in number and nature but, for almost any channel that uses a direct network connection between endpoints, it includes the IP address of the bridge. If the censor knows this IP then they can apply an

[1] https://github.com/tst-race/raceboat-pets2024

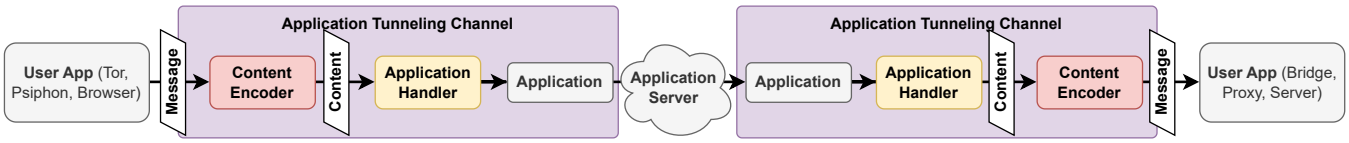

Figure 3: Application content tunneling at a conceptual level: all such channels take user app messages, encode them in a valid format for the application (not *necessarily* covert), then inject the content into the application in some manner for a second application instance to receive, extract, and decode the content.

IP-based block of all connections to it. This is indeed the way most circumvention communications are blocked - by enumerating and blocking bridges by their IP address [12].

Bridge servers can rotate to new IP addresses, but this creates the problem that they must share this *new* secret to their users. This forms the lowest-layer of the bridge distribution problem: how to get users a *bridge address* when the reason they need the bridge is because their internet access is censored. Further concerns exist around how to decide to allocate bridge addresses [8, 30], but these are out of the scope of Raceboat's contribution.

Existing methods for bootstrapping bridge information include a mix of "out-of-band" methods and domain fronting. Examples of the former include Tor-run email and telegram accounts which users can message specific "bridge request" messages to, receive responses back, and copy these responses into their Tor bridge settings [11]. These require many specific user interactions: getting the information about where to send a message and the exact content of that message; manually sending the message; and manually copying data back from the response to their circumvention application. Domain fronting approaches, specifically *meek* [14] for the TorBrowser, offer more automation: they automatically run the meek pluggable transport to a static domain-fronted URL and then employ a CAPTCHA service to try to prevent censors from obtaining all the bridge addresses. In this case the request and response are automatically handled by the user clicking a button. Overall, bridge distribution is provided by a very small number of channels and relies entirely on domain fronting for the most usable one.

Other circumvention apps and services similarly rely on domain fronting for these types of tasks. We generalize the *bridge distribution problem* to the problem of *signaling*: trying to communicate "control plane" types of information with circumvention app clients when they may have no user-level connection to the uncensored internet. Examples of other kinds of signaling could be communicating information about bridges being blocked, or trying to perform authentication of some kind. In general, we assume these communications are smaller, more latency tolerant, and less frequent, than the user-driven circumvention channel uses like messaging, browsing the web, or streaming content.

2.2 Signaling Channels

We target this *signaling* use case to be fulfilled by *signaling channels*. We consider these to be a subset of circumvention channels that have greater leeway for high latency, low bandwidth, and/or sparse use. In turn, we require greater degrees of blocking resilience. Specifically we require two related properties: first, we require *indirect* channels, meaning endpoints connect to one or more neutral intermediate hosts and not directly to one another at the IP-layer; second, we require *public addressability*, meaning there is no information that must be kept secret from the censor to *initiate* the connection. *Indirectness* is actually a requirement produced by *public addressability* if the censor is assumed to be able and willing to block the IP of any signaling server it learns about. *Public addressability* is necessary because it allows for indiscriminate distribution of the signaling server address without compromising its reachability.

Various systems meeting this signaling channel definition exist: meek and other domain fronting channels achieve this *despite* not satisfying indirectness; refraction routing approaches use routing-level indirection [16]; various channels use email [20, 32, 37]; some use cloud storage services [4]; some use video streaming platforms [23]. A common factor to signaling channel unblockability requirements is a reliance on some existing service that the censor has either not thought to block or finds too costly to block. E.g. hiding with an international communication application used by large segments of the population, or hiding within hosting or routing infrastructure shared by many important internet services. This leaves the functioning of any given channel ultimately up to the censor deciding blocking the channel is worthwhile after all (or convincing the utilized service to take action on the censor's behalf). Individual signaling channels, then, are inherently tenuous and it is therefore valuable to have more of them to reduce the impact to circumvention (and thus also reduce the value to the censor) of blocking any individual channel.

2.3 Application Tunneling

Application tunneling refers to a circumvention channel that functions by running a legitimate application and tunneling covert messages through its existing network connections in some manner. This can and has been done in many different and application-specific ways [2, 3, 15, 17, 18, 21, 23, 26–28, 31, 32, 37]. However, to satisfy our signaling channel properties we are particularly interested in *indirect* application tunnels: the covert data is actually tunneled not just from the application client to a server, but *through* that server and on to another instance of the application client, while assuming no modification to the intermediary server(s).

This distinction is important because it requires the *content* to be fully valid application content that is properly processed by a server. E.g. this excludes approaches like balboa [26] or rook [31] that inject and extract covert data *below* the application layer, as well as those that rely on directly peer-to-peer applications like FreeWave [18]. We observe that these content-based channels largely perform four operations:

- Control running an application
- Encode and decode (often steganographically) covert messages into/out-of valid application content
- Inject encoded content into the application input interface
- Address content to enable the other side to receive it

We make use of this abstraction in Section 4 to decompose application tunneling channels into modularized interoperable components to decrease development time and multiply the impact of novel research developments.

2.4 Domain Fronting and Refraction Routing

Domain fronting and refraction routing both function by "tunneling" a connection at or below the IP-layer. These are both valid methods for obtaining the unblockable public addressability requirements stated above. However, they also both rely on at least *tacit* cooperation from the relevant infrastructure providers (hosting and internet service providers, respectively). Therefore, we seek to improve the development of additional signaling channels.

3 RACEBOAT CHANNEL MANAGER

This section describes the design and implementation of the channel management layer of Raceboat, which handles using multiple channels simultaneously.

3.1 Definitions and Concepts

First, we define a few higher-level concepts used to abstract signaling channels and design Raceboat (but applicable to communications more broadly):

3.1.1 Channel. We use the term channel to refer to an abstract implementation of communications such that two or more instances of the channel can instantiate a link between them and communicate (potentially only unidirectionally).

3.1.2 Creators, Loaders, Addresses, and Directionality. We define two new terms, loader and creator, to capture the roles in establishing a link. These terms avoid the ambiguity of the overloaded terms client and server, and establish semantics that information is only ever needed to be transferred from one side (the creator side) to the other (the loader side). This required information is called a link-address and is generated by the creator and consumed by the loader. This is analogous to a traditional socket connection: the creator is the server-side that binds the socket, the link-address is the <IP, port, protocol> tuple, and the loader is the client-side which connects to the socket. The channel is *publicly addressable* if this link-address can be known by the censor without harm; in this traditional socket analogy, the channel is not publicly-addressed because if the censor knew the link-address they could block access via IP/port-blocking.

The second concept is the directionality of links relative to the creator and loader. That is, is the link bidirectional, or unidirectional from creator to loader or from loader to creator. These seemingly trivial semantics actually have significant implications on how channels can be used in practice when out-of-band communication is limited and public vs. secret information is critical to viability.

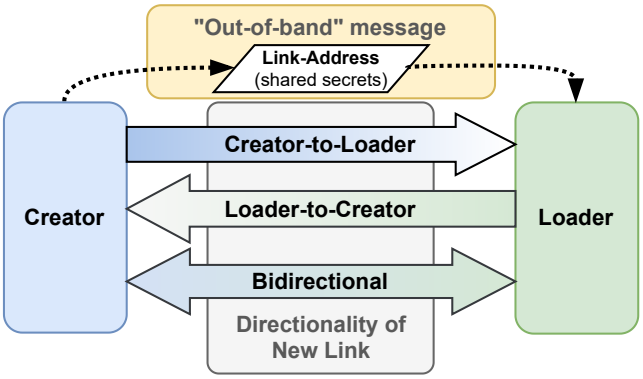

Figure 4: The semantics of link creation: a creator called to instantiate a new link creates new Link-Address that is shared via some pre-existing method to the loader; the loader uses just this Link-Address as a source of shared secrets for completing the link. The directionality of the new link is based on the directionality of the channel.

3.2 Bridge Request/Response Use Case

We examine how these semantics impact signaling channels in the common use-case of bridge distribution: a client within the censor's sphere of control, with a copy of publicly available software and no shared secret information, needs to successfully send a request to a broker and get a response containing a bridge-address back. The lack of shared secrets immediately implies several conditions:

The client cannot use a direct IP connection to the broker because the censor would know this IP and block connections to it. Using an indirect connection implies an application tunneling approach because there must be (one or more) 3rd-party servers facilitating the client-to-broker connection. The request and response need not traverse the same channel. The broker-to-client connection can use a shared secret, so long as the client creates it.

Current approaches take this scenario and a novel application tunnel and develop a bespoke set of messages over that tunnel in both directions to provide a publicly-addressable link. On occasion, some systems [33] have explicitly designed asymmetric hybrid connections that employ different uplink/downlink channels but these are always rigidly built to support a specific pair of channels, effectively just creating a doubly-complex channel.

In contrast, Raceboat abstracts over arbitrary signaling channels implementing the creator/loader semantics above. We are able to state exactly the channel primitives sufficient to handle the use-case - which are not restricted to a publicly-addressable bidirectional channel (that is *sufficient*, but not *necessary*). Instead, we require a loader-to-creator publicly-addressed channel for the client-to-broker request and either a publicly-addressed creator-to-loader or just a loader-to-creator channel for the response. In the latter case, we can piggyback on the client request to also pass a link-address for the broker to respond on (semantically a "reply-to address"). Note that this still collapses to a single publicly-addressed bidirectional channel if that use of such a channel is desired, and Raceboat makes no requirement that multiple channels be used.

This may initially seem like unnecessary additional theory, and all that is really useful is a common interface for using one-of a suite of signaling channels. However, there are often practical considerations of usability and scalability that can make some channels infeasible to use in a bidirectional manner.

3.3 Communication Modes

Raceboat supports more than just the single-request single-response "protocol" above. There are actually four modes that drive use of channels:

(1) Unidirectional Push
(2) Request-Response
(3) Socket
(4) Bootstrapping Socket

(1) Unidirectional Push is just a one-sided version of (2) Request-Response, described in more detail above. This again may seem a trivial use-case, but explicitly building support for it means Raceboat avoids performing any unnecessary (and potentially more detectable or resource-wasting) handshakes as e.g. a SOCKS-wrapped implementation of a channel would. Similarly the request-response mode explicitly avoids more than a single round-trip of messages. The (3) Socket mode is provided for completeness in cases where a continuous connection is desired. Finally, the (4) Bootstrapping Socket is a specialized case in which Raceboat bootstraps a socket-like connection with shared secrets from a publicly-addressed initial channel, illustrated in Figure 5. This is equivalent to embedding both the bridge request and subsequent bridge connection into a single application-level step and could be used when larger amounts of data needs to be conveyed in either direction.

3.4 Implementation

We implemented Raceboat with a simple CLI-based application layer to perform any of its four communication modes (see above) and take a configuration bundle of channel names and special parameters (e.g. account credentials required for account-based services, etc.) as either CLI-arguments or a manifest file. Raceboat can also be directly included as a C++ library. Internally, Raceboat implements a plugin-based architecture where each plugin provides one or more channels (or components of decomposed channels, see Section 4.2 below). These plugins are dynamically loaded at runtime to support flexible deployment scenarios and minimize difficulty updating.

Note, for the purposes of mobile use, plugins are not run as separate processes but are all run within the Raceboat process (or a parent process, if Raceboat itself is included as a library).

Plugins implement a straightforward asynchronous API covering activation/deactivation of a channel, creating/loading/destroying links and connections, sending/receiving packages, and callbacks to update the status of the channel, links, or packages. This API is slightly more complex than the Pluggable Transports API, but also allows support for more nuanced use of the channels and better handling of error conditions.

3.4.1 Cross-Language Bindings. While the Raceboat framework is implemented in C++, we have built language bindings for Python, Java, Rust, and Go to seamlessly support running plugins in those languages.

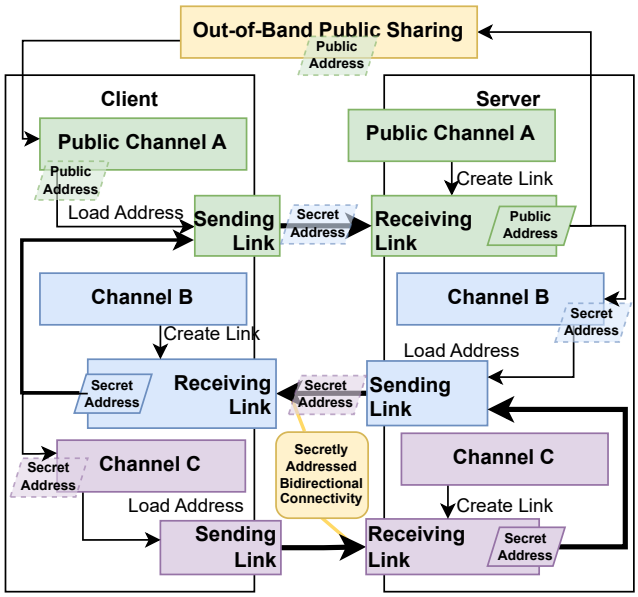

Figure 5: Bootstrapping-socket protocol that constructs a secretly-addressed bidirectional connection from an initial publicly-addressed unidirectional link. Illustrated channels are all loader-to-creator but bidirectional and creator-to-loader channels can also be used.

The cross-language implementation varies depending on the language: Python and Go are handled via SWIG auto-generated bindings [29]. Java and Rust bindings are both explicitly built as bidirectional translation layers. This is more time consuming to develop, but also provides greater transparency over SWIG's auto-generated translation layers.

3.4.2 Communication Modes. Raceboat provides several distinct communication modes to support different use cases and implements these as separate protocol state machines for easier extensibility. Each state machine manages the asynchronous use of the channels involved in the connection. We will now walk through the state machine for the complex bootstrapping-socket case (see Fig 5): first it activates each channel involved in the protocol (up to three in this case); then links are created or loaded as appropriate for their role in the protocol (sending and/or receiving) and directionality (see Fig 4). For links that are created, their link-addresses are extracted and concatenated with User App messages: this enables bootstrapping a new link in-band of an existing one. Finally, User App messages are batched into a minimum number of channel sends based on a reported maximum-transmission-unit (MTU) for the channel being used.

The protocol transparently multiplexes Raceboat control messages (namely, link-addresses of new links) with User App messages to minimize channel usage. Since some links cannot exist until after the connection starts (e.g. in-band bootstrapped links) the protocol state machines encompass the entire connection, not just an

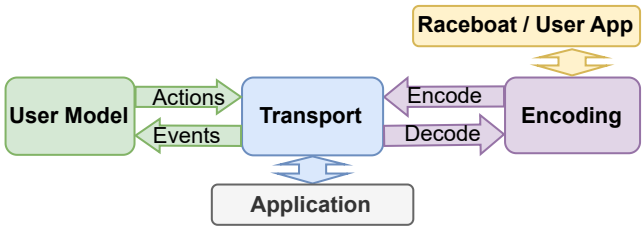

Figure 6: High-level relation of components. Raceboat sends/recvs messages through the encoding. The user model emits actions. The transport executes actions using data from the encoding and passes received data back to be decoded.

internal initialization logic phase on each node. For less complex communication modes, the state machines are effectively subsets of the bootstrapped-socket case, in which fewer channels and links are used and fewer (or no) control messages are necessary.

4 DECOMPOSING APPLICATION TUNNELING

We observe that application tunneling channels have a common set of functionalities required for their use. This orthogonality should mean that an advancement in one functionality can be applied to improve, or more often extend, existing channels by "swapping in" the new version for the old. However, the current bespoke one-off nature of channel implementations mean this development looks like source-code level reuse at best, and often design-level reuse that involves reimplementing all the shared functionality.

4.1 Components

There are likely many ways to modularize application tunneling channels - our design divides channels into three *components* and aims to enable swapability of components without impacting the ability to express novel application tunneling techniques or domains. Additionally, we sought to separate the concerns of the component developers to allow experts in one area to contribute novel components while using existing versions of other components. We observe that application tunneling protocols broadly function as: managing use of an application; encoding/decoding messages into/out-of application content; interfacing with the application to send and receive content. We will use a running example of an application tunneling channel based on sending emails containing steganographic image attachments throughout the following descriptions of our design.

4.1.1 User Models. The user model component controls when and what application actions are executed. Conceptually this is targeted at user-level behaviors - e.g. when an email is sent, how large it is, whether it has an attachment or not. Many existing application tunneling channels do not include any functionality of this sort. However, as recent research shows [32], this can leave a significant gap in security because it causes the cover application usage to be shaped by the user (whether a human or a circumvention application). Breaking this linkage provides behavioral independence [32] and enables evaluating the security of a channel independent of its eventual communications usage.

User models represent application usage as a timeline of actions. These actions correspond to implementations provided by the transport component and can contain parameters about the actions. E.g. our email user model specifies a timeline of send-email actions, the size of the body text, and whether there should be an attachment or not. This is fairly simple, but more complex behaviors are expressible in the user model framework: in particular, the set of action types is only restricted by what the transport component implements.

In some use cases, true behavioral independence is unnecessary and introduces significant performance loss. E.g. an email channel operating in a permissive network environment may only need to avoid exceeding some sending limit rather than make its messages adhere to a rich model of real user behavior. For these cases we provide an optional "on-demand" API that informs the user model of new data to send and allows it to provide a set of "supplemental" actions to facilitate sending. Even in this case the user model may still add nuance to its response, e.g. providing delayed sending actions or sometimes responding with supplemental actions and sometimes sticking to its original timeline.

In addition to actions, there are events that are produced by the transport and consumed by the user model. This feedback path enables reactive user models that could, e.g., update the action timeline to include a new send action for a fraction of new emails received (to mimic a user sending more replies when they receive more emails). Again the space of events is only constrained by the implementation choices of the user model and transport. This means an arbitrarily complex user model can be expressed, such as the very-reactive OUStralopithicus model [22].

Despite requiring the user model and transport agree on the types of actions and events there is still value in separating them: it enables independent development, distribution, and use of new intercompatible versions: e.g. once a single email transport and user model are developed, new user models can be built and deployed with no interaction with the email transport code or developer knowledge. Imposing a formal separation also encourages modular code that is more easily built upon: e.g. adapting an approach taken in one user model is easier when its code is not deeply intertwined with other functionalities.

4.1.2 Transports. The transport component handles all interactions with external entities; e.g. the email transport includes a library which functions as a client for 3rd-party email servers and handles actually sending and receiving emails. As described above, the transport implements translating user model actions into interactions with the cover application and detecting and sending events back to the user model.

Transports can define their actions and events arbitrarily, but to maximize utility we encourage transports of similar types to share actions and events as much as possible. e.g. a Twitter transport and a Mastodon transport will need independent implementations to deal with different application APIs, but ideally they can provide identical action and event types to enable user model interoperability. We could have pushed the decomposed design further and explicitly built multiple types of transports with different interfaces to enforce this pattern. However, that could restrict the expression of new transports.

The transport is also responsible for handling "addressing." Conceptually, this is *how a message sent by one transport instance is able to be (efficiently) received by a counterpart instance* and it requires implementing the semantics of link-addresses described above (Section 3.1.2). E.g. the email transport straightforwardly implements this by simply sending emails to specific email addresses. However, something like publicly posting to a series of pseudorandomly rotating hashtags (and the receiver polling the same) can make this logic more complex. Theoretically the addressing functionality and the application interaction could be decomposed into separate components. However, addressing is often deeply intertwined to application-specific implementations, even among similar applications: e.g. Twitter supports searching for an intersection of multiple hashtags while Mastodon only supports searching for single hashtags, so a hashtag-based addressing scheme would need be different as well.

Execution of actions is how messages are ultimately sent and received. However, since all data is being sent through an application tunnel, the actual messages must be encoded into valid application content before being transmitted. Thus, in addition to consuming *actions* from the user model, the transport also consumes *content* from the encoding (see below). The transport provides a specification about what content (if any) is suitable to include in the execution of an action and the encoding provides content that satisfies this specification. E.g. the email transport receiving a send-email action for a message with an attachment would, in turn, request a body of natural language text and an image from the encodings to support execution of the action.

4.1.3 Encodings. The encoding component is the most straightforward: it encodes messages into application content according to a specification or, in reverse, decodes messages from application content. As with the user model and transport, the encoding and transport can operate over any agreed set of content. However, the intention is to support generic types of content used across many applications: i.e. common image and video formats, natural language text, and base64 strings. The transport can specify more complex parameters for content, such as the size of an image, or the length of a body of text. Within this context, arbitrarily complex encodings can also be constructed, e.g. an image encoding that takes a genre parameter for what type of image content to return.

In addition to encoding and decoding messages, the encoding is also responsible for providing requested content even when there is no message to send. E.g. if the transport is instructed to send an email with an attachment but there is no message to send, the encoding is still required to provide an image to attach, and said image simply will not have any data encoded in it.

4.1.4 Expressibility of Existing Circumvention Channels. The benefits of the decomposed approach are clear: re-usability and elimination of redundant development. However, a valid concern would be whether adopting the decomposed framework restricts the potential circumvention channels that can be built. To assess this, we surveyed 8 recent circumvention channels and assessed if-and-how they would be suitable to decompose. We also note that in many cases an existing channel does not already contain representative functionalities from all three component types *but would be stronger if it did.*

Camoufler [27]. Camoufler communicates over digital instant messenger platforms - *transporting* messages through the text and attachment features of WhatsApp, Signal, Telegram, Slack, and Skype. In a decomposition, each of these applications would be its own transport but they could share a generic "instant messenger user model." The camoufler system as-published does not perform any encoding of data, but the authors explicitly note a need for steganography if the application does not provide end-to-end encryption (E2EE) and the censor can access or influence the service. Hence, decomposing could immediately resolve this by enabling text and multimedia encodings to be seamlessly added in for applicable cases. Further, we believe a more realistic user model could be necessary in cases where the censor can profile user behavior and the user app is performing significant data transfer.

Collage [5]. Collage communicates over image-posting websites - *encoding* messages into images and then posting those images on particular image hosting services and microblogs. Collage also uses the concept of *tasks* to specify both when and where an image should be posted or searched for in order to both retain realistic application usage and enable receiving to find sender content. The image generation of the collage system naturally decomposed to an image-based encoding. The task-based posting and polling system is more complex, and would best decompose into both a user model and transport component, or potentially multiple such components. The user model timeline fits with the "when and where" task scheme, and the transport would simply need to implement taking the encoded content and posting it to a particular service. We also note a decomposition of Collage particularly shows the expressibility of our decomposition: e.g. Collage stipulates a connection between a task (post an image with "#flowers") and the content for it (the image posted should contain flowers). This connection is provided for by the user model and transport's ability to specify arbitrary parameters, like an image genre, to the encoding for each content generation request.

CovertCast [23]. CovertCast communicates over video streaming services - *encoding* messages into valid video content formats and publishing them via livestream services like YouTube. This system neatly decomposes into an encoding piece that generates video data and a transport component that interfaces with YouTube. The "encoding" used is focused on bandwidth not covertness and could easily be detected if inspected, thus there would be a benefit to having alternative video encodings to use in scenarios with a stronger adversary. Inversely, the encoding could benefit from additional transports for other livestreaming services where YouTube may be blocked. Finally, a user model could improve covertness, particularly to prevent detectable behavior patterns in streaming clients.

FreeWave [18]. FreeWave communicates over audio streaming services - *encoding* messages into audio data via frequency manipulation and streaming it between endpoints via Skype. This system decomposes into an encoding component to produce the audio data and a transport component to control Skype and inject/extract audio content. Decomposing to an abstracted and reusable encoding component could make it easier to handle fixed and variable length audio codecs (a noted challenge for the authors). Decomposing

would also have enabled easier adaptation to other audio streaming applications (e.g. see Protozoa WebRTC transport below). Finally, a user model would again improve covertness against censors that can employ behavioral profiling.

Format Transforming Encryption (FTE) [9]. FTE was originally employed in a protocol-mimicry channel. However, we identify that the underlying mechanism of *encoding* messages into arbitrary regex-defined (or otherwise rankable) languages is a valid steganographic technique for many types of content (particularly machine-generated content). A number of specialized FTE instances could be built as encodings for use with various transports (e.g. see Camoufler transports above).

Protozoa and Stegozoa [3, 15]. Protozoa and Stegozoa both communicate over WebRTC video connections. Protozoa *encodes* messages in a high-bandwidth manner but is easily detected if the WebRTC video content can be inspected; Stegozoa *encodes* messages using a steganographic encoding that reduces bandwidth but provides security against content inspection. These would be decomposed as a shared WebRTC-based transport component and two separate encoding components. It is clear the authors took this modular approach in developing both systems, so we argue the advantage of Raceboat's decomposition is in pushing for a more flexible modularization that would enable these encoding components to be effortlessly used for various other transports with video application content (e.g. see CovertCast YouTube transport above). Similarly, a WebRTC transport could be used with other encodings that might provide situational advantages in bandwidth or security. Finally, a user model would again improve covertness against censors that can employ behavioral profiling.

Raven [32]. Raven communicates over email services - it encrypts messages using GPG and then sends them to the recipient. The main innovation in Raven is introduction of a strict timeline of email (send-time, size) tuples to produce realistic behavior that is independent of (and thus unable to be violated by) the user app demands to send and receive messages. This timeline is based on sampling a sophisticated GAN model of real human email usage. Raven would be decomposed into an email client transport, a GPG-based encoder (or simply a GPG-formatter), and a user model that generates the timeline of sending actions. Decomposing would be beneficial by enabling raven to swap in more sophisticated encodings (i.e. if the censor can inspect email content then GPG emails will likely be blocked or a source of suspicion).

4.2 Implementation

As with "unified" channels described above (see Section 3.4) Decomposed channels are also implemented as plugins, where each plugin provides one or more components (see Fig. 7). Components are dynamically assembled into a composition at runtime. This enables extremely lightweight mixing-and-matching of components, e.g. the user or calling program can specify a new channel simply by modifying a JSON file to switch which components are used. Furthermore, the same component can be used simultaneously by multiple compositions - we explicitly designed component instantiation to enable resource-intensive pieces (like a generative AI model) to only be loaded once in these cases.

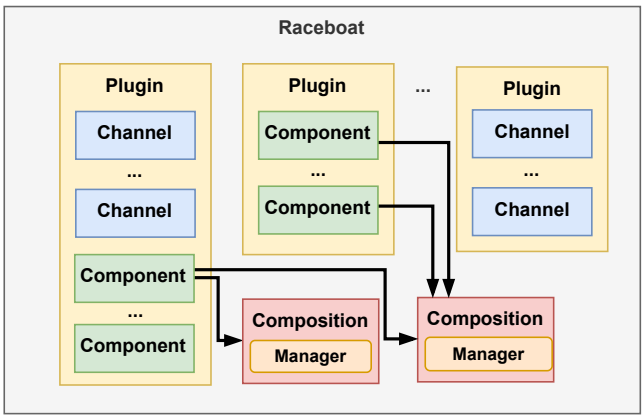

Figure 7: Relationship of Raceboat, plugins, channels, components, compositions and managers. Each component can be used in multiple compositions, including sharing singleton resources.

4.2.1 Composition Manager and Execution Flow. Internally, each composition is run by a *manager* which coordinates: fetching and scheduling a timeline of actions from the user model; requesting content from the encoding(s) for actions based on parameters the transport provides; passing encoded content to the transport; and finally pushing the scheduled actions to the transport for execution.

The call-paths for sending and receiving in a composition are shown in Fig. 8. These paths abstract-out the role of the manager in making each of these API calls, handling callbacks, and passing data between components. The manager has two inputs that drive its behavior: the timeline of actions provided by the user model ((1), *getTimeline*) and the queue of user app messages to-be-sent ((3), *sendPackage*). The action timeline dictates when the manager makes calls to *doAction* in the transport. Actions are intended to drive all transport behavior, so they include not only actions like "post an image" (to send messages) but also "view the newest post with this hashtag" (to *receive* messages) and "post an innocuous comment" (to be a more convincing user).

In cases where content is required for an action (e.g. posting an image), the enqueued user app messages (if any) are involved. First, the manager transforms an action into a set of content parameters via the transport ((2), *getActionParams*)). These parameters can include details specified by the user model (carried over from the action) and by the transport, which most importantly is a MIME-type field to state what type of content is required. An action can also require multiple pieces of content of different types (e.g. an image and text). A composition can contain multiple encodings (e.g. a text encoding and an image encoding) and the manager selects from among the encodings for a given piece of content based on the MIME-type field of the parameters. Once selected, the manager passes the parameters and enqueued user app message bytes to the encoding ((4), *encodeBytes*). Fragmentation and batching of user app messages is handled by the manager based on the covert data capacity for a piece of content reported by the encoding. If a message is too large then it is fragmented over multiple pieces of content, potentially spanning multiple actions. Conversely, messages that

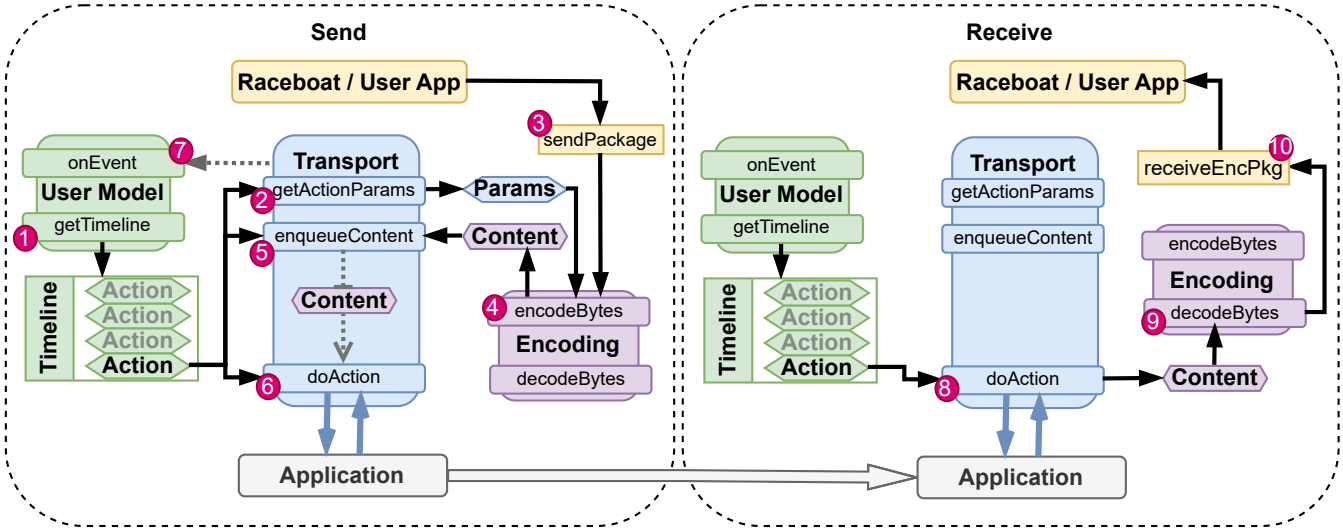

Figure 8: Decomposed channel component interactions to send and receive messages. Arrows between components are calls and callbacks facilitated by the composition manager (not explicitly show). Both the sender and receiver executions will be co-occurring on an execution single instance, but are split apart for clarity. Numbers correspond to API calls highlighted in Section 4.2.1.

are smaller than the capacity are batched together into a single piece of content. If there are no messages to encode, the encoding is expected to produce a valid piece of content, and it will simply lack any covert data. When the encoding finishes it passes the encoded content back to the manager, which passes it to the transport for use with a particular action ((5), *enqueueContent*).

When the timestamp of an action comes, the manager calls the transport to execute it ((6), *doAction*). The action has already been checked for content requirements, and if any content is required then it has already been enqueued in the transport (see Section 4.2.2 below for complexities). Thus, the transport should execute the action immediately, preserving a high-fidelity to the user model timeline. The interaction between transport and "application" is entirely internal to the transport: this could be requests against a remote server API (e.g. using an email client library) or manipulation of a locally running application (e.g. injection of data into a browser). At any time, the transport can also issue an *event* reflecting some application-level change (e.g. a service outage or a new email received) that is passed to the manager and then on to the user model ((7), *onEvent*); this enables arbitrarily complex, reactive, user models for cases where that type of covertness is necessary.

Receiving data follows much the same path, except typically there is no content required *from* the encoding. Rather, the transport executes an action ((8), *doAction*) that involves *receiving* content. These pieces of received content are passed to the manager which again passes them to an encoding based on MIME-type for decoding ((9), *decodeBytes*). The content is decoded into user app message data and a header (for handling fragmentation) and passed back to the manager to be reassembled and pushed to the user app as one or more new received messages ((10), *receiveEncPkg*).

4.2.2 Scheduling. A critical challenge to composition usage is handling potentially conflicting timings between when user models dictate actions should occur and how long encodings take to generate content for those actions. Failing to properly engineer these steps could disrupt the security and/or performance of a composition in several ways.

If encodings are too slow then the transport could either be forced to execute actions without necessary content or wait and violate the user model timeline of action occurrences. A potential heuristic to minimize this problem would be eagerly encoding content as soon as new message data exists to encode. However, this heuristic has two flaws: 1) new messages arrive for encoding within an existing sequence of actions, we ideally want to send the message on the soonest possible action, but we need a way to know if the soonest action is *too soon* for the encoding to complete in time; 2) if we eagerly encode as soon as the *first* new message arrives then we miss the opportunity to batch subsequent messages into the same content for the soonest action. For some compositions and use cases this sort of batching can be critical for usable performance, and we do not want to force the user app to understand these details when it decides whether and how to concatenate or fragment its messages.

Relatedly, we give the option for the user model to *update* its timeline in case, for security reasons, the user model needs to adjust its near-future actions based on events received from the transport. When an update occurs, actions must potentially be removed, which cascades into the need to re-encode messages originally encoded for those actions into new content for new actions.

We overcome these challenges by using an estimated max-encoding-time for each encoding component. The manager then uses this as a basis for estimating how soon *before an action will execute* that encodings must be called to generate content for the action.

Table 1: Components Implemented

User Model	Transport	Encoding
FileScript	RedisClient	NOOP
OnDemand	EmailClient	Base64
	S3Bucket	JEL Image

This way the transport is never delayed waiting for content, but neither are messages that could have been batched instead put into separate actions and bandwidth wasted.

4.2.3 Intercomponent Data Structures. There are naturally several points at which components need to produce data to be consumed by other components. The structure of these data are often obvious when working through a single example, but quickly become varied when the components involve change in their character. e.g. an email transport can essentially take arbitrary content for attachments, but an imageboard transport must take images (and possibly images of specific size and type if server-side conversions are to be avoided).

4.3 Component and Channel Implementations

In collaboration with other performer teams on the DARPA RACE program we have developed an initial set of components (see Table 1) designed to act as a combination of real world circumvention channel components and as exemplars for aiding independent development. We believe some of these components have novel value on their own, but leave this information to independent publications for brevity.

We also have a suite of non-decomposed channels that have been wrapped to support the Raceboat channel APIs. These include wrappers around several higher-bandwidth direct channels (Obfs [35], Snowflake [7], and Balboa [26]), as well as seven lower-bandwidth indirect channels suitable for signaling use (using a variety of social media or email services and varying image-based steganographic methods).

5 EVALUATION

We consider the design and development of the Raceboat framework to be our main contribution, not the performance or security of any individual channel or composition that it can run. Therefore, we focus our evaluation on demonstrating the flexible capabilities described in the above design. We use a set of channels, some constructed as compositions from components and some built as monolithic channels, to demonstrate Raceboat's capabilities, and briefly describe them below for context and to demonstrate the range of channels Raceboat can enable to interoperate.

Obfs. We use a version of Obfs wrapped in a Raceboat-compatible wrapper written in Go. It imports the standard Obfs Go module which is a *direct* channel and so likely not usable for many signaling use cases, but for higher-bandwidth communications.

Snowflake. We use a version of Snowflake wrapped in a Raceboat-compatible wrapper written in Go. It imports the standard Snowflake module which, like Obfs, is *direct* and useful for non-signaling use cases.

S3Bucket. This is a transport based on reading and writing to publicly-permissioned AWS S3 objects. This is inspired by CloudTransport, but uses permissions to enable public *put/get* without *list* to enable secret addressing as well as public addressing. Network traffic appears as generic HTTPS traffic to an S3 region, not associated with the specific bucket. A null encoding is used for this threat model, but arbitrary content encodings can be used since S3 will accept any type of object data.

Email+Base64. This channel is a composition of an email transport and user-model with a simple base64 encoder. The composition sends and receives by encoding data into the body of the email and sending emails to particular recipients. It is essentially equivalent to Raven or SWEET but written within the decomposed framework.

Email+JEL. This channel is a composition of the email transport and user-model (above) combined with jpeg steganographic encoder based on the JEL technique [6]. It encodes data into jpegs that are sent as attachments instead of the body of emails.

Flickr+JEL. This channel uses the above JEL encoder but rather than transporting them via email it uploads them to Flickr with pseudorandomly chosen hashtags. Receivers then poll these hashtags to check for new covert message. Upload and polling rates are controlled to avoid Flickr API limits.

Tumblr+JEL. This channel uses the JEL encoder and a similar pseudorandom hashtag scheme but uploads and polls on the Tumblr service rather than Flickr.

5.1 Bridge Distribution Evaluation

One of the primary use cases identified for Raceboat is facilitating bridge distribution. Recent research on the Lox system [30] shows the development of more elaborate bridge distribution protocols may be necessary for preventing adversary exhaustion of bridges via enumeration. Therefore we decided to demonstrate the flexibility and performance of Raceboat by empirically evaluating the latency for completing the equivalent of a Lox bridge fetch and invitation acceptance protocol. This consists of a single 346B request and a 1.3KB response that bundles together a Lox invitation redemption and bridge request from client-to-server and a Lox response and bridge address from server-to-client. We evaluate this sequence across a combination of channels and compositions used as upstream and downstream links to demonstrate the ability to run arbitrary combinations of channels. We mark channels with an asterisk(*) that are likely inappropriate for this use case, usually due to using direct IP connections that would be enumerated and blocked by a censor, but include them to demonstrate compatibility in the Raceboat framework.

All connections were run on a laptop with 6-core 2.6GHz CPU and 32GB of RAM. Where channels required accounts with application services (e.g. email channels requiring email addresses) we created temporary accounts for testing.

Table 2: Latency of a Lox-based Bridge Request + Invitation Redemption (346B request + 1.3KB response) using different combinations of upstream and downstream channels. * indicates direct channels likely unsuitable for signaling; N/A indicates technical incompatibility of underlying channel software.

	Downstream						
Upstream	obfs*	snowflake*	S3Bucket+NOOP	Email+Base64	Email+JEL	Flickr+JEL	Tumblr+JEL
obfs*	1.47s	0.96s	2.54	11.28s	14.11s	89.4s	69.34s
snowflake*	0.75s	0.47s	2.52	11.34s	15.17s	66.25s	66.14
S3Bucket+NOOP	3.15	2.99	6.58	24.54	26.73	90.10	71.42
Email+Base64	11.93s	6.7s	14.51	16.54s	21.48s	89.51s	52.13
Email+JEL	9.81s	10.46s	18.56	19.3s	20.50s	89.51s	55.13
Flickr+JEL	62.59s	55.61s	65.13	47.91s	54.90s	90.58s	N/A
Tumblr+JEL	32.91	27.59	32.88	47.24	49.73	N/A	80.94

5.2 Results

The results shown in Table 2 demonstrate that Raceboat is capable of flexibly using a variety of very different channels to achieve a core signaling use-case. Purely in terms of *performance* some channels clearly make stronger cases for use than others. However, the point of Raceboat is that different channels will be suitable for different threat scenarios: snowflake and obfs are (expectedly) the fastest but are also not actually usable in most signaling use cases. Further, the very existence of multiple seamlessly-swappable channels is a compelling strength because it forces a censor to split attention and resources blocking many different channels before benefits can be gained.

6 RELATED WORKS

6.1 Pluggable Transports

Pluggable Transports(PT) [25] is the current de facto standard interface for exposing circumvention channels to user apps. There are multiple active versions of the PT specification, including the original subprocess-oriented version and the newer V2.0 suited for inclusion as a library. There is direct overlap in the effect of the PT and Raceboat interfaces: both provide a simple and uniform way to use a circumvention channel with socket-like semantics. However, PTs are designed to be implemented on the basis of individual and independent channels - even if some PT libraries implement multiple channels, *a given connection can only use one at a time.* For example, the obfs4proxy [1] software provides both obf4s and meek [14], but there is neither an interface nor internal logic to use both for a given connection. Moreover, the single-channel orientation of PT means there are no semantics around which to build a fully expressive multichannel library like Raceboat: i.e. there is no existing formalism around a channel providing anything other than bidirectional connectivity, and no uniform way to generate new links on channels based on a desired set of properties. We address the topic of compatibility with the PT interface in Section 7.2.

6.2 Turbo Tunnel

Raceboat incorporates some concepts already expressed by the Turbo Tunnel [13] design, namely automatically applying fragmentation and ordering to user app messages. However, Turbo Tunnel presents itself as a *design pattern* to be applied to individual channels. This inherently assumes continuing the "one channel at a time" paradigm in which the user app is expected to use a single channel for a given bidirectional communication task. Some of Turbo Tunnel's suggestions, like using sessions to seamlessly shift among short-lived proxies, are reminiscent of Raceboat's desired channel agility; the difference is that Turbo Tunnel applies these goals at the level of each channel independently, while Raceboat aims to enhance communication by *flexibly shifting between* channels.

Implementing Turbo Tunnel's approach inside a channel does not impact Raceboat's use of that channel, so we consider Turbo Tunnel to be orthogonal to our work, operating at the channel layer. We believe Turbo Tunnel's suggestions mostly apply to the *transport* component of our decomposed channel design. Furthermore some of its approaches, like packetizing and providing ordering, are actually automatically implemented by the Raceboat *manager* and so do not need to be implemented by each individual transport.

6.3 Marionette

Marionette [10] is spiritually similar to Raceboat's channel decomposition. However, the two are fundamentally different: Marionette explicitly rejects application tunneling in favor of a more sophisticated version of protocol mimicry. The main shared features are an emphasis on building a modular framework with developer use and plugin re-use as major goals. Marionette does not engage with the topic of flexibly shifting between channels or specifying multiple simultaneously, rather trying to mimic on the fly. If the advanced protocol mimicry was still wanted by a user, it could be wrapped into a usable Raceboat channel.

7 DISCUSSION

7.1 Other Signaling Channel Use Cases

We have referred to bridge requests as the exemplary case for signaling channel usage throughout this paper. However, we believe there are a number of other immediate use cases for them.

Recent bridge distribution schemes [8, 30] introduce nontrivial amounts of latency-tolerant "control traffic" between censored clients and uncensored infrastructure. Reporting bridge blockages or other censorship telemetry from the client is a similar case. In these cases signaling channels are beneficial if there is either a

significant chance the client knows no reachable bridges or if there is a cost to using a bridge (increasing likelihood of discovery) that is only justified for servicing actual user requests.

Signaling channels can also have roles outside of large-scale circumvention infrastructures. For example, we have built-in a *bootstrapping* mode in which a publicly addressed channel is used for a single upstream message, which then bootstraps secretly addressed links in each direction (or a single bidirectional one). This combines the conceptual elements of bridge distribution and bridge use; very similar to Snowflake's use of a domain-fronted broker mediating connection to ephemeral proxies, but *abstracted to any suitable combination of channels Raceboat supports.*

Finally, signaling channels are potentially suited for any other latency-tolerant uses. Asynchronous messaging akin to social media can be accomplished, as prior systems have shown [20]. Higher latencies also do not always imply low bandwidth: e.g. steganographic videos uploaded to 3rd-party streaming sites can involve lengthy encoding and upload delays but provide megabytes of data transfer at a time without using a direct connection to a bridge. Use cases for such channels could include distributing static censored content *to* users, pushing content *from* users, or even distributing software updates for circumvention tools.

7.2 Pluggable Transport Compatibility

There are two sides to the Pluggable Transports interface: the user app and the channel (or "transport" in PT parlance). Raceboat already demonstrates the ability to wrap the channel interface to allow a PT-compatible channel to be used by Raceboat (see use of Obfs4 and Snowflake).

The user app interface is different in that the PT interface specifies a subset of Raceboat functionality. A PT connection is equivalent to Raceboat in continuous connection mode with a single channel for send and receive and static link addresses. Therefore, Raceboat *can* be wrapped in a PT-compliant wrapper for use by existing PT-based user apps. However, that eschews most of the benefits of the Raceboat framework: the continuous connection mode is a poor fit for short-lived signaling uses; the restriction to bidirectional channels removes flexibility and security gains from including unidirectional channels.

7.3 Security of Composed Channels

We have focused on Raceboat as an intermediary capability that includes dynamically creating *compositions* from modular components. We have detailed the interfaces of those components and which areas of security each is responsible for, but have not addressed the actual security of a composition. Ultimately we consider this out-of-scope of this work, aside from ensuring our framework does not *weaken* the security provided by each component. E.g., Raceboat never generates its own *actions* for a transport to execute: only the user model component can do so. Therefore, Raceboat cannot violate behavioral independence unless the user model does so. Similarly, Raceboat never provides content for transport actions, only the encoding does.

We do *not* guarantee the security of any given component, and therefore the security of any given composition. Rather, the varying landscape of threats and frequent tradeoffs between security and performance would make this overly restrictive. E.g. we include a "NOOP" encoding component that simply returns the bytestream given to it; this is useful in cases where a transport is assumed secure from adversarial inspection but is a gross security flaw otherwise. We recommend newly developed components are designed and evaluated as they are now: with specific threat models and evaluations. Insofar as the authors plan to maintain a library of components and compositions, the relevant threat models for each will be clearly and prominently documented to avoid misunderstandings that could endanger users.

7.4 Value of User Models

Related to threat models, a reasonable question is whether the user model component is necessary or, if not, brings any value along with its added complexity. Current reports on censors do not show the use of application behavior profiling to block circumvention channels [36]. Despite this, we argue including user models is valuable for several reasons: first, behavioral profiling is not **yet** in use, but a censor could begin using it in the near future (particularly on a smaller or more focused scale than whole-of-nation). We argue it is better to have proactively designed user models *now* than suddenly require a major reworking of components, almost certainly breaking backwards compatibility, to incorporate them later.

Additionally, in the case of transports that interact with application *servers*, e.g. email or social media transports, user model-style rate limitations may be necessary. Most such services have denial-of-service, spam, and/or bot detection protections; Raceboat transports do not aim to act maliciously with respect to the application server, but if interaction is driven based on the user app then the transport could still fall afoul of these protections. In these cases, even if a *realistic* user model is not necessary, a *throttling* user model may be necessary to avoid automatic blocking by the server.

8 CONCLUSION

We have presented Raceboat, a framework for flexibly developing and using censorship circumventing signaling channels. We have provided a formalized definition and implementation for circumvention channel link establishment that enables a flexible mix-and-match approach to upstream and downstream communications, including the concepts of public vs. secret addresses and indirect vs direct channels. We have designed an expressive modular decomposition of application tunneling channels into orthogonal components to enable faster development and greater research reuse. We have implemented both of these innovations in a framework and shown its capability to use both new and existing circumvention channels, based on adaptation of Pluggable Transports compatibility, to conduct useful signaling channel tasks, e.g. supporting use of a cutting edge bridge distribution approach.

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] Dustin Bachrach, Christopher Nunu, Dan S. Wallach, and Matthew Wright. 2011. *#h00t: Censorship Resistant Microblogging*. Technical Report. Rice University and University of Texas at Arlington. https://arxiv.org/pdf/1109.6874v1.pdf

[3] Diogo Barradas and Nuno Santos. 2020. Towards a Scalable Censorship-Resistant Overlay Network based on WebRTC Covert Channels. In *Distributed Infrastructure for Common Good*. ACM. https://www.gsd.inesc-id.pt/~nsantos/papers/barradas_dicg20.pdf

[4] 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

[5] Sam Burnett, Nick Feamster, and Santosh Vempala. 2010. Chipping Away at Censorship Firewalls with User-Generated Content. In *USENIX Security Symposium*. USENIX. https://www.usenix.org/event/sec10/tech/full_papers/Burnett.pdf

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

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

[8] Frederick Douglas, Rorshach, Weiyang Pan, and Matthew Caesar. 2016. Salmon: Robust Proxy Distribution for Censorship Circumvention. *Privacy Enhancing Technologies* 2016, 4 (2016), 4–20. https://censorbib.nymity.ch/pdf/Douglas2016a.pdf

[9] 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

[10] Kevin P. Dyer, Scott E. Coull, and Thomas Shrimpton. 2015. Marionette: A Programmable Network-Traffic Obfuscation System. In *USENIX Security Symposium*. USENIX. https://www.usenix.org/system/files/conference/usenixsecurity15/sec15-paper-dyer.pdf

[11] email-bridges 2023. Get Bridges for Tor. https://bridges.torproject.org/ Accessed on 5/30/2023.

[12] Roya Ensafi, David Fifield, Philipp Winter, Nick Feamster, Nicholas Weaver, and Vern Paxson. 2015. Examining How the Great Firewall Discovers Hidden Circumvention Servers. In *Internet Measurement Conference*. ACM. http://conferences2.sigcomm.org/imc/2015/papers/p445.pdf

[13] David Fifield. 2020. Turbo Tunnel, a good way to design censorship circumvention protocols. In *Free and Open Communications on the Internet*. USENIX. https://www.usenix.org/system/files/foci20-paper-fifield.pdf

[14] 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

[15] Gabriel Figueira, Diogo Barradas, and Nuno Santos. 2022. Stegozoa: Enhancing WebRTC Covert Channels with Video Steganography for Internet Censorship Circumvention. In *Asia CCS*. ACM. https://dl.acm.org/doi/pdf/10.1145/3488932.3517419

[16] Sergey Frolov, Fred Douglas, Will Scott, Allison McDonald, Benjamin VanderSloot, Rod Hynes, Adam Kruger, Michalis Kallitsis, David G. Robinson, Steve Schultze, Nikita Borisov, J. Alex Halderman, and Eric Wustrow. 2017. An ISP-Scale Deployment of TapDance. In *Free and Open Communications on the Internet*. USENIX. https://www.usenix.org/system/files/conference/foci17/foci17-paper-frolov_0.pdf

[17] Bridger Hahn, Rishab Nithyanand, Phillipa Gill, and Rob Johnson. 2016. Games Without Frontiers: Investigating Video Games as a Covert Channel. In *European Symposium on Security & Privacy*. IEEE. https://people.cs.umass.edu/~phillipa/papers/castle.pdf

[18] Amir Houmansadr, Thomas Riedl, Nikita Borisov, and Andrew Singer. 2013. I want my voice to be heard: IP over Voice-over-IP for unobservable censorship circumvention. In *Network and Distributed System Security*. The Internet Society. https://people.cs.umass.edu/~amir/papers/FreeWave.pdf

[19] Amir Houmansadr, Edmund L. Wong, and Vitaly Shmatikov. 2014. No Direction Home: The True Cost of Routing Around Decoys. In *Network and Distributed System Security*. The Internet Society. http://dedis.cs.yale.edu/dissent/papers/nodirection.pdf

[20] Shuai Li and Nicholas Hopper. 2016. Mailet: Instant Social Networking under Censorship. *Privacy Enhancing Technologies* 2016, 2 (2016), 1–18. https://www-users.cse.umn.edu/~hoppernj/mailet_popets.pdf

[21] Shuai Li, Mike Schliep, and Nick Hopper. 2014. Facet: Streaming over Videoconferencing for Censorship Circumvention. In *Workshop on Privacy in the Electronic Society*. ACM. https://www-users.cs.umn.edu/~hopper/facet-wpes14.pdf

[22] Anna Harbluk Lorimer, Lindsey Tulloch, Cecylia Bocovich, and Ian Goldberg. 2021. OUStralopithecus: Overt User Simulation for Censorship Circumvention. In *Workshop on Privacy in the Electronic Society*. ACM. https://cypherpunks.ca/~iang/pubs/oustral-wpes21.pdf

[23] Richard McPherson, Amir Houmansadr, and Vitaly Shmatikov. 2016. CovertCast: Using Live Streaming to Evade Internet Censorship. *Privacy Enhancing Technologies* 2016, 3 (2016), 1–14. https://www.cs.cornell.edu/~shmat/covertcast/covertcast.pdf

[24] Mohsen Minaei, Pedro Moreno-Sanchez, and Aniket Kate. 2020. MoneyMorph: Censorship Resistant Rendezvous using Permissionless Cryptocurrencies. *Privacy Enhancing Technologies* 2020, 3 (2020), 404–424. https://petsymposium.org/2020/files/papers/issue3/popets-2020-0058.pdf

[25] Pluggable Transports 2023. What Pluggable Transports do. https://www.pluggabletransports.info/how-transports/ Accessed on 5/30/2023.

[26] Marc B. Rosen, James Parker, and Alex J. Malozemoff. 2021. Balboa: Bobbing and Weaving around Network Censorship. In *USENIX Security Symposium*. USENIX. https://www.usenix.org/system/files/sec21-rosen.pdf

[27] Piyush Kumar Sharma, Devashish Gosain, and Sambuddho Chakravarty. 2021. Camoufler: Accessing The Censored Web By Utilizing Instant Messaging Channels. In *Asia CCS*. ACM. https://censorbib.nymity.ch/pdf/Sharma2021a.pdf

[28] Piyush Kumar Sharma, Rishi Sharma, Kartikey Singh, Mukulika Maity, and Sambuddho Chakravarty. 2023. Dolphin: A Cellular Voice Based Internet Shutdown Resistance System. *Privacy Enhancing Technologies* 2023, 1 (2023). https://petsymposium.org/popets/2023/popets-2023-0034.pdf

[29] swig 2023. Welcome to SWIG. https://swig.org/ Accessed on 5/30/2023.

[30] Lindsey Tulloch and Ian Goldberg. 2023. Lox: Protecting the Social Graph in Bridge Distribution. *Privacy Enhancing Technologies* 2023, 1 (2023). https://petsymposium.org/popets/2023/popets-2023-0029.pdf

[31] Paul Vines and Tadayoshi Kohno. 2015. Rook: Using Video Games as a Low-Bandwidth Censorship Resistant Communication Platform. In *Workshop on Privacy in the Electronic Society*. ACM. https://censorbib.nymity.ch/pdf/Vines2015a.pdf

[32] 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.

[33] Qiyan Wang, Xun Gong, Giang T. K. Nguyen, Amir Houmansadr, and Nikita Borisov. 2012. CensorSpoofer: Asymmetric Communication using IP Spoofing for Censorship-Resistant Web Browsing. In *Computer and Communications Security*. ACM. https://censorbib.nymity.ch/pdf/Wang2012a.pdf

[34] 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

[35] Philipp Winter, Tobias Pulls, and Juergen Fuss. 2013. ScrambleSuit: A Polymorphic Network Protocol to Circumvent Censorship. In *Workshop on Privacy in the Electronic Society*. ACM. https://censorbib.nymity.ch/pdf/Winter2013b.pdf

[36] Mingshi Wu, Jackson Sippe, Danesh Sivakumar, Jack Burg, Peter Anderson, Xiaokang Wang, Kevin Bock, Amir Houmansadr, Dave Levin, and Eric Wustrow. 2023. How the Great Firewall of China Detects and Blocks Fully Encrypted Traffic. In *USENIX Security Symposium*. USENIX. https://www.usenix.org/system/files/sec23fall-prepub-234-wu-mingshi.pdf

[37] Wenxuan Zhou, Amir Houmansadr, Matthew Caesar, and Nikita Borisov. 2013. SWEET: Serving the Web by Exploiting Email Tunnels. In *Hot Topics in Privacy Enhancing Technologies*. Springer. https://petsymposium.org/2013/papers/zhou-censorship.pdf