Fountain codes in censorship circumvention rendezvous

Eleanor Cawthon
University of California, Berkeley
eleanor@eecs.berkeley.edu

David Fifield
david@bamsoftware.com

Abstract

Censorship circumvention protocols often require a *rendezvous* step to exchange information needed to set up a connection. We investigate the potential of fountain codes, also known as rateless erasure codes, for this purpose. Fountain codes reliably transmit a message by breaking it into many small, specially encoded pieces. Taking inspiration from how they are used in the Collage and Assemblage circumvention systems, we narrow our focus to rendezvous, and consider the usefulness of fountain codes as a design component in themselves, not necessarily adjunct to steganography.

Our specific motivation is to enable rendezvous over encrypted DNS, which has attractive anti-blocking properties, but is difficult to use for generic data exchange. Fountain codes offer an uncomplicated way to send discrete, bounded-length messages over covert channels that, like DNS, may convey only small amounts of data at a time, or are unordered or unreliable.

We provide a proof-of-concept implementation over UDP that demonstrates the essential elements of fountain code–based rendezvous: breaking a message into pieces, transmitting them over a lossy channel, and reconstructing them at the other end. We work out a prospective design for rendezvous over encrypted DNS, and comment on design considerations in general.

Keywords

censorship circumvention, rendezvous, fountain code, DNS

1 Rendezvous in censorship circumvention

The task of censorship circumvention is to connect a client in a restricted network to the outside world, often with the help of an intermediary proxy or relay. Many circumvention designs involve a preliminary *rendezvous* or *signaling* step to exchange a small amount of information before being able to bootstrap a full connection. Rendezvous protocols, like circumvention protocols in general, must be resistant to blocking by a censor. They are limited in that they start from zero: they must work without the client knowing any non-public information, such as a secret IP address or encryption key. Their compensating advantage is that, since rendezvous occurs infrequently and involves only small amounts of data, they may use covert channels that would be too slow or expensive for bulk data transport. Rendezvous designs often use common protocols in uncommon ways, finding creative ways to embed covert data into what appears to be ordinary traffic. Vines et al. [18] provide an overview and systematization of the rendezvous problem, as well as a framework, Raceboat, for modularizing rendezvous channels.

Free and Open Communications on the Internet 2026(2), 7–11

Our focus in this paper is rendezvous over fragmented or unreliable network protocols: ones that cannot send a complete rendezvous message in one piece, or that do not guarantee in-order delivery. We have in mind, specifically, encrypted DNS: we wish to transmit rendezvous messages through a censorship firewall by way of encrypted DNS queries and responses. As we will develop in section 4, a single DNS query can encode only about 140 bytes of data—too few for many kinds of rendezvous. We will adopt, as a motivating use case, the rendezvous of the Snowflake circumvention system [4 §2.1]. Snowflake's rendezvous messages are on the larger side: before establishing a peer-to-peer connection with a proxy, a client must send a message of about 1500 bytes upstream, and receive a message of about 1800 bytes downstream. These sizes drop to about 350 and 400 bytes, respectively, after zlib compression [10]; still, they are too big for DNS. What can be done?

One option is the Turbo Tunnel approach [9]: establish a full-fledged reliable transport on top of the underlying unreliable protocol, as TCP is built on IP. This option requires interactive communication, and relatively many packets in both directions, but is fully general. This is what Kindling [14] does, in its DNS mode. Alternatively, one might invent a custom reliability protocol tailored for rendezvous, with, for example, sequence numbers, acknowledgments, and retransmissions. This is roughly the approach taken by Raceboat [18 §4.2.1] and DNS-Morph [1 §6].

Without invalidating these other approaches, our purpose in this paper is to suggest another way, based on *fountain codes*. It is flexible, easy to implement, and lightweight in terms of network communication. In our view, fountain codes may do for discrete messages something like what Turbo Tunnel [9] does for streams, enabling a range of daring new designs and capabilities.

2 Background on fountain codes

Fountain codes, also known as rateless erasure codes, are a class of error correction codes that break a message into a practically unlimited number of small chunks called *encoding symbols*, such that the original message can be recovered after sufficiently many encoding symbols have been received. The amazing property of these codes is that, with high probability, *any* sufficiently large subset of encoding symbols will work: symbols may be lost in transit, duplicated, or reordered; sender and receiver do not need to exchange feedback about which have been received and which have not. As long as the sender keeps generating encoding symbols, and the receiver keeps feeding what it receives into a decoder, the message will eventually be delivered. The combined length of the number of encoding symbols required is scarcely greater than the length of the original message. These properties make fountain codes a good fit for the problem of rendezvous over a fragmented or unreliable communications channel—one that permits sending only a small amount of data at a time, or in which the order or delivery of individual units of communication is not guaranteed.

We will not be concerned with the technical details of how fountain codes work, only with the properties they offer. Byers et al. [6] (1998) outlined requirements for an ideal fountain code, which include efficiency (linear-time encoding and decoding) and that the number of chunks required for decoding be equal to the number of chunks in the original message. The standard code RaptorQ [15] (2011) closely approximates the ideal. We use RaptorQ in the prototype implementation of section 3.

Fountain codes have been applied in censorship circumvention before, in two works that inspire this one: Collage [5] in 2010 and Assemblage [11] in 2026. Their goal is a bit different from ours: they aim to enable an exchange of messages between users on opposite sides of a censorship firewall, making use of some amount of prior, out-of-band coordination.[1] More complex applications, like news aggregation, may be built atop this basic operation. Collage and Assemblage rely centrally on steganography, being based in sharing information over public forums. A sender breaks a message into chunks using a fountain code, steganographically encodes the chunks into text or media files (generically called *vectors*), and posts the vectors to a public forum, such as a photo-sharing site. The receiver downloads a subset of vectors, undoes the steganography, and decodes the embedded chunks to reconstitute the message.

Fountain codes have the potential to benefit censorship circumvention in other ways. They are particularly well-suited to rendezvous, and to covert channels where we may rely on encryption rather than steganography to hide information. Rendezvous is often conceptualized with an assumed correspondence between message and medium: a message is atomic and is to be transferred as a unit, in *an* HTTP request or *an* email message, for example. Fountain codes offer a way to loosen this correspondence and expand the space of usable covert channels and rendezvous designs.

3 Prototype implementation

We made a small proof-of-concept implementation in Python to demonstrate the use of fountain codes for rendezvous. It is not integrated with a blocking-resistant protocol like the design we will sketch in section 4 (the carrier is simple UDP datagrams), but it demonstrates the essential functions of breaking a message into encoding symbols, sending the symbols over a network, and decoding them to recover the message (in two directions). The source code is available from https://repo.or.cz/erasure-code-rendezvous.git.

The prototype consists of two programs, a sender and a receiver. The sender reads a message from a file and uses a third-party RaptorQ module [3] to generate a sequence of encoding symbols from it. It sends the symbols, one at a time, enclosed in UDP datagrams, to the receiver. The receiver extracts the encoding symbols from incoming datagrams and feeds them into a RaptorQ decoder. After collecting enough encoding symbols to decode the message, the receiver composes a response message (simulated by converting the sender's message to uppercase), generates its own sequence of encoding symbols, and begins sending them, one at a time, as responses to the sender's datagrams. Simulating a request–response protocol, the sender continues sending encoding symbols (now superfluous to the receiver) until it, too, has collected enough encoding symbols to decode the response message.

The size of the message and of encoding symbols must be communicated out of band. We will say more on this point in section 5.

Many senders may interact simultaneously with the same receiver. To enable the receiver to distinguish different senders, upstream datagrams are tagged with a *session identifier*, as in Turbo Tunnel [9 §2]. A session identifier is an 8-byte random string, generated independently by each sender, that remains constant throughout a rendezvous session. The receiver has not one RaptorQ decoder, but many, one for every session identifier it sees. (Which gives rise to state management concerns we will discuss in section 5.) One might think to distinguish senders by source IP address, rather than by a separate session identifier. That approach fails, however, when different senders pass their traffic through a shared intermediary, as is common in circumvention (and is the case with the recursive DNS name servers we will discuss in section 4).

4 A design for DNS-based rendezvous

Here we sketch a design for a rendezvous system based on fountain codes, encrypted DNS, and public recursive name servers. Encrypted DNS offers a degree of blocking resistance because a censor cannot inspect and block individual queries and responses—it can only block encrypted name servers generally.[2]

What makes DNS work as a covert channel is that a recursive name server is, effectively, a proxy. It receives queries from DNS clients and forwards them to a name server that is authoritative for the name being queried. We arrange things so that the message receiver *is* the authoritative name server for some domain name we control: queries for any subdomain will consequently be forwarded to the receiver. To send a short message, the sender encodes it as a DNS query for a subdomain of the designated domain name, and sends the query to any public, third-party recursive name server. The recursive name server forwards the query to the authoritative name server; that is, the receiver. The receiver may send a response message encoded as a DNS response, which is likewise forwarded back to the sender via the recursive name server. This basic principle underlies any DNS tunnel.

Encryption is a necessary element of any such scheme, under a censorship circumvention threat model. The sender's queries unavoidably contain the domain name of the receiver—that is how the recursive name server knows where to forward them. Responses contain the domain name as well. A censor might inspect plaintext DNS messages and block any that are destined for, or coming from, a known rendezvous receiver. Therefore, the link between the sender and the recursive name server must be encrypted—using, for example, DNS over HTTPS or DNS over TLS. We will assume the recursive name server is outside the censor's network, so one "hop" of encryption is sufficient. The link between the recursive name server and the receiver may be plaintext (UDP or TCP port 53).

[1] These papers use the word "rendezvous," but in a different sense from how we use it. Their rendezvous refers to the process by which a sender and receiver agree on how, when, and where to exchange steganographic covertexts. For example: using a certain search keyword, on a certain photo-sharing site, on a certain date. For us, the receiver is fixed, and the means of reaching it may be considered a global constant (for example, "send encrypted DNS queries for subdomains of a certain well-known domain name"). Importantly, rendezvous, as we use the term, may not depend on preshared secrets: indeed, it may *be* the mechanism by which shared secrets are established.

[2] To be sure, some censors *do* try to discover and block encrypted DNS servers, as Lange et al. have shown [13 §6]. That's fine—we are not trying to develop a silver bullet that is effective against all censors everywhere; but rather to support new designs and make possible more communications channels in more places.

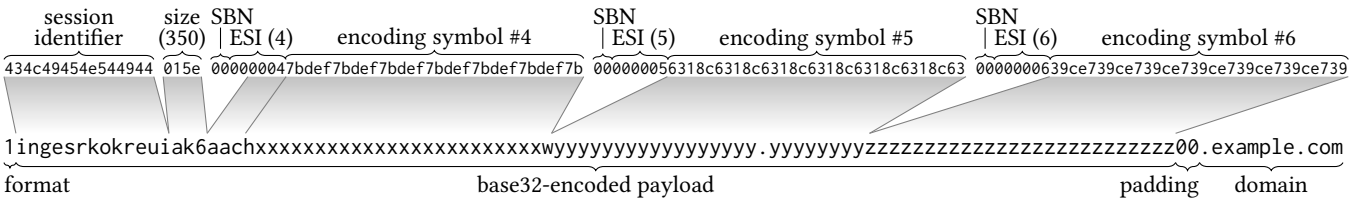

Figure 1: A sample encoded DNS query QNAME. This one contains three encoding symbols, numbered #4, #5, and #6. Up to eight encoding symbols may be packed into one query. The upper, hexadecimal portion shows the payload components before base32 encoding. Only the first Encoding Symbol ID (ESI) is encoded explicitly, compressed to 2 bytes.

Encrypted recursive name servers let us use DNS as a covert proxy; fountain codes break a large message into small chunks. We must now develop the details of how those chunks are encoded as DNS queries and responses. (In Collage/Assemblage terms, each DNS query or response is a *vector*.) The encoding we will arrive at is similar to the one used in the DNS tunnel dnstt [9 §3.4]. It uses base32-encoded domain names in upstream queries and TXT resource records in downstream responses. We assume that the recursive name server (which sees DNS messages in their unencrypted state) is non-hostile: it does not interfere with messages based on how they are constructed. Therefore the encoding does not need to aspire to deniability in the way of, say, DNS-sly [2].

The format of DNS messages is specified in RFC 1035 [17 §4]. There are two kinds of message: queries and responses. We use different encodings in the two directions, to take better advantage of the affordances of the protocol. Queries are more restricted than responses in their capacity for carrying data, and consequently their encoding requires more care.

4.1 Encoding DNS queries

It is not enough that DNS queries be syntactically valid—we must also encode data in a way that survives transit through a recursive name server (which does not "forward" queries so much as issue them anew [16 §4.3.1], and therefore may modify them in various ways). Though there are many fields in a query that might be used to store data, we are aware of only one that survives recursive resolution: QNAME, the name being queried for [17 §4.1.2].

Restrictions on QNAME are the main reason for the limited capacity of DNS covert channels. There may be only one QNAME per query, of length at most 255 bytes, including an obligatory terminator byte. The QNAME must end with the domain name that is designated for the covert channel, to enable forwarding by recursive name servers. Call the length of this domain name N; we have $254 - N$ bytes to work with at the outset. A DNS name is divided into *labels* of at most 63 bytes, and each label has a 1-byte length prefix [17 §3.1]. In effect, we may use only 63 of every 64 bytes, so we are now down to $\lfloor \frac{63}{64}(254-N) \rfloor$ bytes. We will see later that we want to reserve 1 byte for a format indicator and 1 byte for a padding delimiter, which leaves us with $\lfloor \frac{63}{64}(254 - N) - 2 \rfloor$ bytes.

In principle, DNS names may be composed of any byte values; but, for compatibility, RFC 1035 recommends using only upper- and lowercase ASCII letters, digits, and hyphens [17 §2.3.1]. We may not rely on letter case, either, as DNS names are case-insensitive, and recursive name servers can and do modify case as a mitigation for cache poisoning attacks [19]. These considerations lead us to base32 encoding [12 §6] (without '=' padding), whose alphabet consists of the 26 letters 'a'–'z' and the 6 digits '2'–'7'. Base32 transforms 5 input bits into 8 output bits, so a QNAME with an N-byte domain suffix may ultimately encode up to $\left\lfloor \frac{5}{8} \lfloor \frac{63}{64}(254 - N) - 2 \rfloor \right\rfloor$ bytes. If N is a generous 24 bytes, that is a capacity of 140 bytes per query.

How shall we apportion the capacity? Let us reserve 8 bytes for a session identifier, 2 bytes for the message size, and 2 bytes for "Payload ID" metadata (explained in the next paragraph). In the 128 bytes that remain, we propose to include up to 8 encoding symbols of 16 bytes each. One query may convey up to 128 bytes of useful message data. Sending a zlib-compressed Snowflake rendezvous message of 350 bytes requires a minimum of 3 queries.

RaptorQ associates, with each encoding symbol, a 4-byte *Payload ID* [15 §4.2] that consists of a 1-byte Source Block Number (SBN) and a 3-byte Encoding Symbol ID (ESI) [15 §3.2]. To save space, we may specify that the SBN is always 0, so it does not need to be encoded explicitly. We may additionally require the ESI (a sequential zero-based counter) to be small enough to fit in 2 bytes (65536 encoding symbols per message is more than enough for rendezvous). Within a query, let the ESIs be sequential; then we only need to encode the first one. With these optimizations, Payload ID metadata occupies just 2 payload bytes per query, for any number of encoding symbols.

Why use many small encoding symbols per query, instead of one large one? The reason is traffic shaping. With one large symbol, every query that conveys a nonzero amount of rendezvous data would have a large minimum size—a measurable traffic feature that a censor might try to exploit. Multiple shorter symbols afford the flexibility to send shorter queries while still making forward progress (even if more queries are required overall). It makes the encoding compatible with longer domain suffixes as well: the sender may include 7 encoding symbols rather than 8, if that is all there is room for. We will say more about traffic shaping in section 5.

To control the sizes of queries with fine granularity, we will support padding inside QNAME. Padding will consist of a run of '0' bytes placed between the base32-encoded payload and the domain suffix. (For precise control over sizing, padding must be outside base32 encoding.) '0' is not part of the base32 alphabet, so it is easy to separate from the payload. We will require there to be at least 1 byte of padding, so that a decoder can easily remove both padding and domain suffix together (the padding acts as a delimiter).

Finally, it is prudent to reserve 1 byte for a format code. If we decide, later, to alter some aspect of the encoding—adjust the size of encoding symbols, say, or change how padding works—we can

do so, without breaking compatibility, by defining a new format. Let a format code of '1' stand for the encoding we have worked out: 8-byte session identifier, 2-byte message size, 2-byte initial ESI, 16-byte encoding symbols, base32, '0' padding.

All these elements—the format code, base32-encoded payload, and padding—are concatenated. Label separators (dots) are inserted as needed to divide the string into labels of 63 bytes or less. (The dots themselves do not convey information.) Finally, the domain suffix is appended. See Figure 1 for a sample encoded QNAME.

4.2 Encoding DNS responses

DNS responses are less constrained than queries, and encoding data into them is more straightforward. A DNS response can be seen as a collection of *resource records* of various types; familiar types are A (for a 4-byte IPv4 address) and AAAA (for a 16-byte IPv6 address). The TXT resource record [17 §3.3.14] is a good general-purpose data container: it represents an arbitrary byte string, with an overhead of just 1 byte per 255 bytes. There does not seem to be a reason to seek out anything more complicated. The length of a TXT record is limited by the size of the response it is contained in. By default, UDP DNS messages are limited to 512 bytes [17 §4.2.1], but the widely supported EDNS(0) mechanism allows a DNS client to declare that it supports larger responses [7 §6.2.3]. Response sizes of 1232 bytes or greater are common in practice.

Even with larger responses, a single TXT record may not be able to contain a complete rendezvous response message. We should therefore use a fountain code in the downstream direction as well. As with queries, we may specify that the TXT data consists of a format code, a size field, an initial ESI, and as many encoding symbols as will fit (or as many as traffic shaping calls for), followed by optional padding. There is no need for a session identifier in responses. Depending on the allowed response size, returning a zlib-compressed Snowflake rendezvous message of 400 bytes requires a minimum of 1 or 2 DNS responses.

Putting it together: the sender breaks its rendezvous message into chunks using a fountain code, encodes the chunks as TXT queries, and sends the queries, one at a time or in parallel, via a recursive name server to the receiver. The receiver decodes incoming encoding symbols (using different decoders for different senders, distinguishing them by session identifier). Until the receiver has collected enough encoding symbols to recover the sender's message, it responds to queries with empty responses (or responses consisting of shaped padding). After decoding the sender's message, the receiver starts encoding its own response message into responses. The sender keeps sending queries (to give the receiver something to respond to) until the entire response message is received.

A complete exchange of compressed Snowflake rendezvous messages can take as few as 3 or 4 DNS round trips. (It needs 3 queries and 1 or 2 responses, but one round trip can do double duty, carrying the last upstream encoding symbols in the query and the first downstream encoding symbols in the response.) The time required depends, naturally, on network conditions and the choice of name servers. In our tests with a client and a recursive DNS over HTTPS server (8.8.8.8) located in the U.S., and an authoritative name server in Europe (for ripe.net), completing 4 round trips takes less than 1 s, even when queries are not made in parallel.

5 Discussion

Multi-modal rendezvous. Breaking a message into parts and distinguishing rendezvous sessions by an abstract session identifier (not a feature of the carrier protocol, such as an IP address or port number) allows for the possibility of rendezvous that spans more than one protocol. The sender could send some of its encoding symbols over DNS, some over HTTP, and some over other any other available covert channel. The receiver listens on multiple protocols simultaneously and identifies encoding symbols that belong to the same sender using the session identifier attached to encoding symbols. Multi-modal rendezvous may be helpful for robustness, when some protocols are blocked but not others; or for covertness, to mimic a specified traffic mix.

Implications for traffic shaping. Among the features a censor may use, in deciding whether to block or allow a connection, are the sizes and timing of packets. A censor that looks at a DNS over HTTPS connection, say, and sees packets that are unusually large or frequent, may become suspicious and block the connection, even if it cannot read the plaintext of the DNS messages being exchanged. Generally, one wants a circumvention protocol to support a traffic shaping property [8 §3.6]: it should be possible (at least in principle) to make the sizes and timing of packets conform to a prescribed schedule. Fountain codes, with their atomic encoding symbols, complicate this. The size of encoding symbols imposes a minimum on the size of protocol data units that carry a nonzero amount of information; one may add padding to a symbol but not make it shorter. (We don't want to subdivide encoding symbols into even smaller fragments—that would take us back to square one.) In the DNS encodings of section 4, we facilitated traffic shaping with short, granular encoding symbols and variable padding. Queries that are too small to contain even one encoding symbol are possible; they contribute to traffic shape even if they do not contribute to message decoding.

The need to communicate encoding parameters. The sizes of encoding symbols and messages are not part of the fountain code. They must be agreed on in advance or communicated at runtime. The size of encoding symbols should be chosen to suit the carrier protocol, but may be considered a preconfigured constant. When messages are fixed or bounded in size, the message size may similarly be implicit. The variable size of Snowflake's rendezvous messages led us to reserve space for size fields in the design of section 4. In any event, the representation of parameters should be compact, because it must be attached to every vector (the sender does not know which vectors the receiver will receive). A format code (the '1' of subsection 4.1) can stand for a whole set of parameters, and leaves room for adjustment if a choice of parameters proves unsatisfactory.

State management. In our prototype implementation, the receiver, in order to isolate the sessions of different senders, allocates a RaptorQ decoder for every new session identifier it sees. This creates the possibility for a denial-of-service attack similar to a SYN flood, in which an attacker sends junk session identifiers to cause the receiver to allocate memory. The receiver needs a policy for expiring old and unused state. However, there is a tradeoff: the receiver must also consider an attack that tries to expire in-progress rendezvous sessions of legitimate users.

Acknowledgments

We are grateful to the authors of Collage and Assemblage, especially Tushar Jois, whose FOCI 2026 presentation on Assemblage inspired this line of research. Paul Vines engaged in early discussion about rendezvous over encrypted DNS, and helped us understand how message fragmentation works in Raceboat. Mikulas Plesak and Xiaokang Wang offered ideas and criticism. Cecylia Bocovich, Sam Burnett, Adam Fisk, Anant Sahai, and Eric Wustrow gave us feedback on a draft. We appreciate the helpful feedback of the FOCI reviewers. The RaptorQ implementation we used in our prototype implementation is by Christopher Berner.

We are grateful for support from the Center for Information Technology Research in the Interest of Society and the Banatao Institute (CITRIS).

The source code of the prototype implementation of section 3, and code and data for supporting measurement experiments on Snowflake rendezvous message size and DNS over HTTPS latency, are available from https://repo.or.cz/erasure-code-rendezvous.git.

References

[1] Rami Ailabouni, Orr Dunkelman, and Sara Bitan. 2021. DNS-Morph: UDP-Based Bootstrapping Protocol for Tor. In *Cyber Security Cryptography and Machine Learning*. Springer. https://link.springer.com/chapter/10.1007/978-3-030-78086-9_19

[2] Qurat-Ul-Ann Danyal Akbar, Marcel Flores, and Aleksandar Kuzmanovic. 2016. DNS-sly: Avoiding Censorship through Network Complexity. In *Free and Open Communications on the Internet*. USENIX. https://www.usenix.org/conference/foci16/workshop-program/presentation/akbar

[3] Christopher Berner et al. 2024. *raptorq*. https://github.com/cberner/raptorq

[4] Cecylia Bocovich, Arlo Breault, David Fifield, Serene, and Xiaokang Wang. 2024. Snowflake, a censorship circumvention system using temporary WebRTC proxies. In *USENIX Security Symposium*. USENIX. https://www.bamsoftware.com/papers/snowflake/

[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/conference/usenixsecurity10/chipping-away-censorship-firewalls-user-generated-content

[6] John W. Byers, Michael Luby, Michael Mitzenmacher, and Ashutosh Rege. 1998. A Digital Fountain Approach to Reliable Distribution of Bulk Data. In *SIGCOMM*. ACM. https://dl.acm.org/doi/10.1145/285243.285258

[7] Joao Luis Silva Damas, Michael Graff, and Paul A. Vixie. 2013. Extension Mechanisms for DNS (EDNS(0)). RFC 6891. https://www.rfc-editor.org/info/rfc6891

[8] Ellis Fenske and Aaron Johnson. 2024. Bytes to Schlep? Use a FEP: Hiding Protocol Metadata with Fully Encrypted Protocols. In *Computer and Communications Security*. ACM. https://doi.org/10.1145/3658644.3690198

[9] David Fifield. 2020. Turbo Tunnel, a good way to design censorship circumvention protocols. In *Free and Open Communications on the Internet*. USENIX. https://www.bamsoftware.com/papers/turbotunnel/

[10] David Fifield. 2026. Data compression on Snowflake rendezvous messages. Tor anti-censorship team mailing list. https://lists.torproject.org/mailman3/hyperkitty/list/anti-censorship-team@lists.torproject.org/thread/ZK3KJ6F3BCJRVNS55BMB6MXQNTTEFRTB/

[11] Tushar M. Jois, Cora Rowena Ruiz, and Gabriel Kaptchuk. 2026. Assemblage: Chipping Away at Censorship with Generative Steganography. In *Free and Open Communications on the Internet*. https://www.petsymposium.org/foci/2026/foci-2026-0005.php

[12] Simon Josefsson. 2006. The Base16, Base32, and Base64 Data Encodings. RFC 4648. https://www.rfc-editor.org/info/rfc4648

[13] Felix Lange, Niklas Niere, and Juraj Somorovsky. 2026. Towards Automated DNS Censorship Circumvention. In *Free and Open Communications on the Internet*. https://www.petsymposium.org/foci/2026/foci-2026-0001.php

[14] Lantern. 2026. *Kindling*. https://github.com/getlantern/kindling

[15] Lorenz Minder, Amin Shokrollahi, Mark Watson, Michael Luby, and Thomas Stockhammer. 2011. RaptorQ Forward Error Correction Scheme for Object Delivery. RFC 6330. https://www.rfc-editor.org/info/rfc6330

[16] Paul Mockapetris. 1987. Domain names - concepts and facilities. RFC 1034. https://www.rfc-editor.org/info/rfc1034

[17] Paul Mockapetris. 1987. Domain names - implementation and specification. RFC 1035. https://www.rfc-editor.org/info/rfc1035

[18] 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://petsymposium.org/popets/2024/popets-2024-0027.php

[19] Paul A. Vixie and David Dagon. 2008. *Use of Bit 0x20 in DNS Labels to Improve Transaction Identity*. Internet-Draft draft-vixie-dnsext-dns0x20-00. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-vixie-dnsext-dns0x20/00/