

Fountain codes in censorship circumvention rendezvous

Draft April 14, 2026

Eleanor Cawthon
cawthon.e@gmail.com

David Fifield
david@bamsoftware.com

Abstract

Certain censorship circumvention protocols require a *rendezvous* step in which the parties share information necessary to set up a connection. We investigate the potential of fountain codes, also known as rateless erasure codes, for this purpose. Fountain codes enable the reliable transfer of a message by breaking it into small encoded pieces. Taking inspiration from their use in Collage and Assemblage, we restrict our focus to rendezvous, while also considering contexts not involving steganography.

Our motivating use case is rendezvous over encrypted DNS, which has attractive anti-blocking features but is difficult to use for large messages. Fountain codes offer a convenient way to transfer discrete, bounded-length messages over covert channels that can transfer only small amounts of data at once, or are unordered or unreliable.

We provide a proof-of-concept implementation over UDP that demonstrates the essential components of fountain code-based rendezvous: breaking a message into pieces, transmitting them over a lossy channel, and reconstructing them at the other end. We further present a prospective design for a realistic rendezvous system based on encrypted DNS.

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 must start from zero: they must work without the client knowing any non-public information, such as a secret IP address or an encryption key. Their compensating advantage is that, since rendezvous occurs infrequently and involves only small amounts of data, they may make use of covert channels that would be too expensive, slow, or cumbersome for general data transport. Rendezvous protocols often use common protocols in uncommon ways, finding creative ways to modulate data into what appear to be ordinary traffic flows. Vines et al. [16] provide an overview and

systematization of the rendezvous problem, as well as a framework, Raceboat, for modularizing rendezvous channels.

Our focus in this paper will be on enabling rendezvous over fragmented or unreliable network protocols: ones that cannot transfer even a single rendezvous message all at once, or in which delivery is not guaranteed. We have in mind particularly rendezvous over encrypted DNS, using DNS queries and responses to transfer information through a censorship firewall. As we will develop in section 4, DNS queries can represent only about 140 bytes of data—too small for many kinds of rendezvous, if messages must be sent all in one piece. We will take as a motivating use case rendezvous for the Snowflake circumvention system [2 §2.1]. Snowflake’s rendezvous is on the larger side—the client has to send about 2 KB upstream, and receive about an equal amount downstream, in order to bootstrap a peer-to-peer connection with a proxy. Snowflake’s rendezvous messages are too big for DNS; what can be done?

One option is the Turbo Tunnel approach [9]: establish a full-fledged reliable transport layer on top of the underlying unreliable protocol (DNS), in the way that TCP is built on IP. This option requires interactive, bidirectional communication, and many data exchanges, but it is fully general. This is the approach taken by Oscur0 [6] and Kindling [12] in their DNS modes. Alternatively, one could invent a lightweight reliability layer, with, for example, sequence numbers, acknowledgments, and retransmissions, tailored for the purpose of rendezvous. This is what Raceboat’s built-in message fragmentation support does [16 §4.2.1].

Without invalidating either of those approaches, our purpose in this paper is to suggest a third way based on *fountain codes*. It is flexible, easy to implement, and lightweight in terms of network communication. It has the potential to make new covert channels possible that previously might have been too awkward to use.

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 symbols have been received and which have not. As long as the sender keeps generating encoding symbols and sending them, and the receiver feeds whatever 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 an excellent fit for the problem of rendezvous over a fragmented or unreliable communications

This work is licensed under the Creative Commons Attribution 4.0 International License. To view a copy of this license visit <https://creativecommons.org/licenses/by/4.0/> or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.



YYYY(X), 1–5

© YYYY Copyright held by the owner/author(s).

<https://doi.org/XXXXXXX.XXXXXXX>

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; instead we will treat them as an abstraction and use them for the properties they offer. Byers et al. [5] (1998) outlined requirements of an ideal fountain code, which include efficiency (linear-time encoding and decoding) and that the number of chunks required to decode be equal to the number of chunks in the original message. The standard fountain code RaptorQ [13] (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 [4] in 2010 and Assemblage [10] in 2026. Their purpose is a bit different from ours: they aim to enable the 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 have steganography as a central component, 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.

Our observation is that fountain codes have the potential to benefit censorship circumvention in ways that do not necessarily involve steganography. Fountain codes are well-suited to rendezvous, and to covert channels where we may rely on encryption rather than steganography to hide information. Rendezvous has commonly been implemented with an assumption of correspondence between message and medium: a message is atomic, and is transferred as a unit, in an HTTP request or an email message (for example). Fountain codes offer a way to loosen this correspondence, and do for discrete messages what a Turbo Tunnel design [9] does for streams: increase convenience and flexibility, and expand the space of usable covert channels.

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 protocol is simple UDP datagrams), but it demonstrates the core functions of breaking a message into encoding symbols, sending the symbols over a network, and decoding them to recover a message. The source code is available from <https://repo.or.cz/erasure-code-rendezvous.git>.

¹These 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 coverttexts. 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 to this well-known domain”). 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.

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 [1] to generate a sequence of encoding symbols. It sends each symbol, enclosed in a UDP datagram, to the address of the receiver. The sender keeps generating and sending symbols until it receives a response indicating that the message has been decoded. The receiver extracts the encoding symbols from incoming UDP datagrams and feeds them into a RaptorQ decoder. After enough encoding symbols are received to decode the message, the receiver generates a response message (simulated by converting the sender’s message to uppercase) and sends it back in a single UDP datagram.

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 with the receiver at the same time. To enable the receiver to distinguish different senders, encoding symbols are tagged with a *session identifier*, as in Turbo Tunnel [9 §2]. A session identifier is an 8-byte random string, generated by the sender, that remains consistent throughout a single rendezvous session. The receiver has not one RaptorQ decoder, but many, one for every observed session identifier. (Which gives rise to state management concerns we will discuss in section 5.) One might be tempted to distinguish senders by IP address, rather than by a separate session identifier. That approach fails, however, when different senders pass their traffic through the same intermediary, as commonly happens in circumvention (as 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.

What makes DNS work as a covert channel is that a recursive name server is, effectively, a proxy. A recursive name server receives queries from DNS clients and forwards them to a name server that is authoritative for the name being queried. We arrange it so that the message receiver is the authoritative name server for a domain name we control: queries for any subdomain will consequently be forwarded to the receiver. To send a short message, the sender encodes it into 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 reply message in the form of an encoded 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 component of any such scheme, in a censorship circumvention threat model. The sender’s queries, of necessity, 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. Therefore plaintext won’t cut it: a censor could inspect queries and filter ones that are destined for a known covert channel receiver. Communication 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”

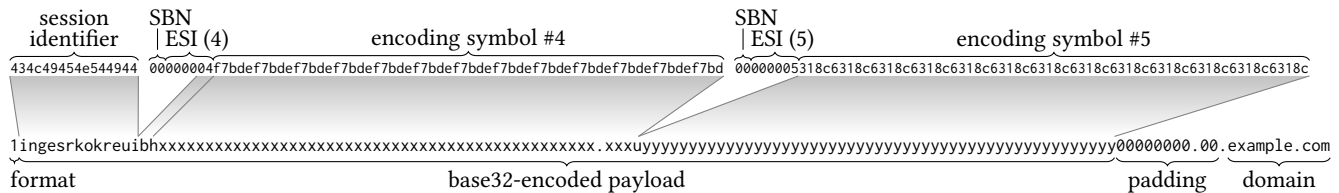


Figure 1: A sample encoded DNS query QNAME. This one contains two encoding symbols, numbered #4 and #5. Up to four encoding symbols are supported. Only the first Encoding Symbol ID (ESI) is encoded explicitly. Upper labels show payload data before base32 encoding. Lower labels indicate “container” data elements, outside base32 encoding.

of encryption is sufficient. The link between the recursive name server and the receiver may be in plaintext (UDP or TCP port 53).

Encrypted recursive name servers let us use DNS as a covert proxy; fountain codes let us 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*; collecting sufficiently many vectors leads to recovery of a message.) The encoding we will arrive at is similar to the one used in the DNS tunnel dnstt [9 §3.4]. It uses base32-encoded query names in upstream queries and TXT resource records in downstream responses. We are not trying to be covert from the recursive name resolver; it is all right if DNS messages are funny-looking. Generally we assume that the recursive name server is non-hostile: it does not need to do anything outside of its usual operation, but it must not actively hinder us.

The format of DNS messages is specified in RFC 1035 [15 §4]. There are two kinds of message: queries and responses. We use different encodings in the two directions in order to make better use 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 encode data in a way that survives transit through a recursive name server (which does not “forward” queries so much as issue them anew [14 §4.3.1], and therefore may change 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 [15 §4.1.2].

Restrictions on QNAME are the primary cause of the limited capacity of DNS covert channels. There may be only one QNAME per query, with a maximum length of 255 bytes (including an obligatory terminator byte). A DNS name is divided into *labels* of at most 63 bytes, each with a 1-byte length prefix [15 §3.1], which reduces the effective capacity to 250 bytes. The last part of the QNAME must be the domain name reserved for the covert channel, to enable forwarding by recursive name servers. Call the length of this domain name N ; we now have $250 - N$ bytes to work with. 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 hyphen [15 §2.3.1]. We may not rely case distinctions, either, as DNS names are defined to be case-insensitive, and recursive name servers can and do modify case as

a mitigation for cache poisoning attacks [17]. These considerations lead us to base32 encoding [11 §6] (without ‘=’ padding), whose alphabet consists of the 26 letters ‘a’–‘z’ and the 6 digits ‘2’–‘7’. Base32 transforms 5 input bytes into 8 output bytes, so a QNAME with an N -byte domain suffix may encode up to $\lfloor \frac{5}{8}(250 - N) \rfloor$ bytes. If N is 25, that’s a capacity of 140 bytes per query.

How shall we apportion the available capacity? Let us reserve 8 bytes for a session identifier. The other 132 bytes may be filled with encoding symbols. We propose to allow up to 4 encoding symbols of length 32 bytes, plus 1 byte for “Payload ID” metadata (explained in the next paragraph). Each query conveys up to 128 bytes of useful message data.

RaptorQ associates, with each encoding symbol, a 4-byte *Payload ID* [13 §4.2] that consists of a 1-byte Source Block Number (SBN) and a 3-byte Encoding Symbol ID (ESI) [13 §3.2]. To save space, we may require the SBN always to be 0, so that it does not need to be encoded explicitly. Because the rendezvous messages we are dealing with are bounded in length, we may additionally require that the ESI (a sequential zero-based counter) be small enough to fit in 1 byte. Let the ESIs of all encoding symbols within the same query be sequential; then we only need to encode the first one. With these optimizations, Payload ID metadata adds just 1 byte of overhead to each query.

Why send several small encoding symbols per query, instead of one larger 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 lets the encoding work with longer domain suffixes: the sender may include three encoding symbols rather than four. We will say a bit more about traffic shaping in section 5.

Speaking of traffic shaping, we want to be able to add padding to a query, to adjust its length by less than the size of an encoding symbol. Padding will consist of a run of ‘0’ characters between the base32-encoded payload and the domain suffix. ‘0’ is not part of the base32 alphabet, so it is easy to separate padding from the payload. Note that padding is outside the base32 encoding, not part of the data payload: this lets us adjust the size with 1-byte granularity.

Finally, it is prudent to reserve one character for a format code. If we decide, later, to alter some aspect of the encoding, adjusting the size of encoding symbols, or changing how padding works, say, we can do so, without breaking compatibility, by defining a new

format. For example, let an initial ‘1’ stands for the encoding we have worked out above: 8-byte session identifier, 32-byte encoding symbols, base32 payload, ‘0’ padding. The format code may also communicate to the receiver the total size of the message (e.g., 2 KB for Snowflake); if larger messages become needed, a new format can be defined. See section 5 for discussion on this point.

All these elements—the format code, base32-encoded payload, and padding—are concatenated and prepended to the domain suffix, with label separators (dots) inserted as needed to ensure that labels are no longer than 63 bytes. See Figure 1 for an example. The label separators themselves do not convey information; the decoder concatenates the labels before the domain suffix before decoding.

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 [15 §3.3.14] contains an arbitrary byte string, with an overhead of just 256/255. There does not seem to be a reason to seek out any more complicated way to store information in responses. The length of a TXT record is limited by the size of the response that contains it. By default, UDP DNS messages are limited to 512 bytes [15 §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]. In practice, response sizes of 1232 bytes or greater are common.

Even with larger responses, a single TXT record may not be enough to return a complete rendezvous response message. We should therefore use a fountain code for responses as well. As with queries, we may specify that TXT record data starts with a format code, which is followed by as many encoding symbols as will fit (or fewer, for traffic shaping purposes), then optional padding. There is no need for a session identifier in the downstream direction. As a DNS response can only be sent in response to a query, the sender must continue sending queries (consisting only of padding, if necessary) until it gets enough encoding symbols to recover the response message.

Putting it together: the sender breaks its rendezvous message into chunks using a fountain code, encodes the chunks into TXT queries, and sends the queries, one by one, 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 collects enough encoding symbols to recover the sender’s message, it responds to the sender’s queries with empty responses (or responses consisting of padding). Once it has recovered the sender’s message, the receiver starts encoding its own response message into responses. The sender continues sending queries until the entire response rendezvous message is received. Then rendezvous is finished.

5 Discussion

Multi-modal rendezvous. Distinguishing rendezvous sessions by an abstract session identifier (rather than a feature of the carrier protocol, such as an IP address or port number) allows for the possibility of rendezvous that uses more than one protocol, even

within the same session. The client may send its encoding symbols over any combination of protocols, in any order. This may be useful for robustness, when one or more of the protocols is blocked by the censor; or for covertness, to mimic a given traffic distribution.

Implications for traffic shaping. The job of a censor is one of classification: for each connection, packet, or other unit of communication, decide whether to block it or allow it. Among the features a censor may use for classification are the size and timing of network packets. A censor that looks at a putative DNS over HTTPS stream and sees packets that are unusually large or frequent, say, may decide something is fishy and block the connection, even if it cannot see the plaintext. Generally, one wants a circumvention protocol to support a traffic shaping property [8 §3.6], so that it is possible (at least in principle) to make the size and timing of sent packets conform to a prescribed schedule. Fountain codes, with their atomic encoding symbols, complicate this. The size of an encoding symbol imposes a minimum on the size of packets that carry effective information: one may add padding to a symbol but not make it shorter. (You don’t want to have to define a meta-protocol to subdivide encoding symbols into even smaller fragments—that takes us back to square one.) This was the reason for using short, granular encoding symbols, and for providing a facility for adding padding, in the DNS query encoding of subsection 4.2. It should be possible to send a QNAME of any size above the static overhead of format character, session identifier, and domain suffix. Even QNAMEs that are too short to contain even one encoding symbol are possible (all-padding queries); they contribute to the traffic shape even if they do not contribute to message decoding.

The need to communicate encoding parameters. The size of encoding symbols and the size of the overall message are not represented as part of the code—sender and receiver must agree on them in advance. For many practical rendezvous scenarios, it suffices to make them fixed constants: we can set an upper bound on the size of rendezvous messages (e.g., 2 KB for Snowflake) and pad all messages to that length, and fix an encoding symbol size that makes sense for traffic shaping and the requirements of whatever carrier protocol is used. For more flexibility, it is also possible to define a compressed representation of these parameters and send it along with every message vector. For example, the ‘1’ format code from subsection 4.2, besides telling the receiver how to parse the remainder of the QNAME, could imply a certain message size. A different format code could be defined if it becomes necessary to support another message size. Adding a field for a quantized, perhaps logarithmic (like the Window Scale option in TCP [3 §2.2]) representation of the parameters may also be appropriate. In any case, the representation should be small, because it will need to be sent along with every vector (since the sender does not know what subset of vectors the receiver will receive).

State management. In our prototype implementation, the receiver, in order to isolate the sessions of different senders, allocates a new RaptorQ decoder for every new session identifier it sees. This raises the possibility of a denial-of-service attack similar to a SYN flood, in which an attacker sends lots of 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 thinking. Mikulas Plesak and Xiaokang Wang offered ideas and criticism. Paul Vines helped us with early discussion about rendezvous over encrypted DNS, and helped us understand how message fragmentation works in Raceboat. The RaptorQ implementation we used in our prototype implementation is by Christopher Berner.

References

- [1] Christopher Berner et al. 2024. *raptorq*. <https://github.com/cberner/raptorq>
- [2] 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/>
- [3] David Borman, Robert T. Braden, Van Jacobson, and Richard Scheffenegger. 2014. TCP Extensions for High Performance. RFC 7323. <https://www.rfc-editor.org/info/rfc7323>
- [4] 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
- [5] 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>
- [6] Mingye Chen, Jack Wampler, Abdulrahman Alaraj, Gaukas Wang, and Eric Wustrow. 2024. Extended Abstract: Oscur0: One-shot Circumvention without Registration. In *Free and Open Communications on the Internet*. <https://www.petsymposium.org/foci/2024/foci-2024-0005.php>
- [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] 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>
- [11] Simon Josefsson. 2006. The Base16, Base32, and Base64 Data Encodings. RFC 4648. <https://www.rfc-editor.org/info/rfc4648>
- [12] Lantern. 2026. *Kindling*. <https://github.com/getlantern/kindling>
- [13] 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>
- [14] Paul Mockapetris. 1987. Domain names - concepts and facilities. RFC 1034. <https://www.rfc-editor.org/info/rfc1034>
- [15] Paul Mockapetris. 1987. Domain names - implementation and specification. RFC 1035. <https://www.rfc-editor.org/info/rfc1035>
- [16] 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>
- [17] 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/>