dnsop
Internet-Draft
Intended status: Standards Track
Expires: February 11, 2019

J. Dickinson
J. Hague
S. Dickinson
Sinodun IT
T. Manderson
J. Bond
ICANN

August 10, 2018

C-DNS: A DNS Packet Capture Format
draft-ietf-dnsop-dns-capture-format-08

Abstract

This document describes a data representation for collections of DNS messages. The format is designed for efficient storage and transmission of large packet captures of DNS traffic; it attempts to minimize the size of such packet capture files but retain the full DNS message contents along with the most useful transport metadata. It is intended to assist with the development of DNS traffic monitoring applications.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on February 11, 2019.

Table of Contents

1. Introduction

There has long been a need to collect DNS queries and responses on authoritative and recursive name servers for monitoring and analysis. This data is used in a number of ways including traffic monitoring, analyzing network attacks and "day in the life" (DITL) [ditl] analysis.

A wide variety of tools already exist that facilitate the collection of DNS traffic data, such as DSC [dsc], packetq [packetq], dnscap [dnscap] and dnstap [dnstap]. However, there is no standard exchange format for large DNS packet captures. The PCAP [pcap] or PCAP-NG [pcapng] formats are typically used in practice for packet captures, but these file formats can contain a great deal of additional information that is not directly pertinent to DNS traffic analysis and thus unnecessarily increases the capture file size.

There has also been work on using text based formats to describe DNS packets such as [I-D.daley-dnsxml], [I-D.hoffman-dns-in-json], but these are largely aimed at producing convenient representations of single messages.

Many DNS operators may receive hundreds of thousands of queries per second on a single name server instance so a mechanism to minimize the storage size (and therefore upload overhead) of the data collected is highly desirable.

The format described in this document, C-DNS (Compacted-DNS), focusses on the problem of capturing and storing large packet capture files of DNS traffic with the following goals in mind:

- Minimize the file size for storage and transmission.
- Minimize the overhead of producing the packet capture file and the cost of any further (general purpose) compression of the file.

This document contains:

- A discussion of some common use cases in which DNS data is collected, see Section 3.
- A discussion of the major design considerations in developing an efficient data representation for collections of DNS messages, see Section 4.
- A description of why CBOR [RFC7049] was chosen for this format, see Section 5.
- A conceptual overview of the C-DNS format, see Section 6.
- The definition of the C-DNS format for the collection of DNS messages, see Section 7.
- Notes on converting C-DNS data to PCAP format, see Section 9.
- Some high level implementation considerations for applications designed to produce C-DNS, see Section 10.

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

"Packet" refers to an individual IPv4 or IPv6 packet. Typically packets are UDP datagrams, but may also be part of a TCP data stream. "Message", unless otherwise qualified, refers to a DNS payload extracted from a UDP datagram or a TCP data stream.

The parts of DNS messages are named as they are in [RFC1035]. Specifically, the DNS message has five sections: Header, Question, Answer, Authority, and Additional.

Pairs of DNS messages are called a Query and a Response.

3. Data collection use cases

In an ideal world, it would be optimal to collect full packet captures of all packets going in or out of a name server. However, there are several design choices or other limitations that are common to many DNS installations and operators.

- DNS servers are hosted in a variety of situations:
  - Self-hosted servers
  - Third party hosting (including multiple third parties)
  - Third party hardware (including multiple third parties)
- Data is collected under different conditions:
  - On well-provisioned servers running in a steady state
  - On heavily loaded servers
  - On virtualized servers
  - On servers that are under DoS attack
  - On servers that are unwitting intermediaries in DoS attacks
- Traffic can be collected via a variety of mechanisms:
  - Within the name server implementation itself
  - On the same hardware as the name server itself
  - Using a network tap on an adjacent host to listen to DNS traffic
  - Using port mirroring to listen from another host

- The capabilities of data collection (and upload) networks vary:
  - Out-of-band networks with the same capacity as the in-band network
  - Out-of-band networks with less capacity than the in-band network
  - Everything being on the in-band network

Thus, there is a wide range of use cases from very limited data collection environments (third party hardware, servers that are under attack, packet capture on the name server itself and no out-of-band network) to "limitless" environments (self hosted, well provisioned servers, using a network tap or port mirroring with an out-of-band networks with the same capacity as the in-band network). In the former, it is infeasible to reliably collect full packet captures, especially if the server is under attack. In the latter case, collection of full packet captures may be reasonable.

As a result of these restrictions, the C-DNS data format is designed with the most limited use case in mind such that:

- data collection will occur on the same hardware as the name server itself
- collected data will be stored on the same hardware as the name server itself, at least temporarily
- collected data being returned to some central analysis system will use the same network interface as the DNS queries and responses
- there can be multiple third party servers involved

Because of these considerations, a major factor in the design of the format is minimal storage size of the capture files.

Another significant consideration for any application that records DNS traffic is that the running of the name server software and the transmission of DNS queries and responses are the most important jobs of a name server; capturing data is not. Any data collection system co-located with the name server needs to be intelligent enough to carefully manage its CPU, disk, memory and network utilization. This leads to designing a format that requires a relatively low overhead to produce and minimizes the requirement for further potentially costly compression.

However, it is also essential that interoperability with less restricted infrastructure is maintained. In particular, it is highly desirable that the collection format should facilitate the re-creation of common formats (such as PCAP) that are as close to the original as is realistic given the restrictions above.

4. Design considerations

This section presents some of the major design considerations used in the development of the C-DNS format.

1. The basic unit of data is a combined DNS Query and the associated Response (a "Q/R data item"). The same structure will be used for unmatched Queries and Responses. Queries without Responses will be captured omitting the response data. Responses without queries will be captured omitting the Query data (but using the Question section from the response, if present, as an identifying QNAME).

   * Rationale: A Query and Response represents the basic level of a client's interaction with the server. Also, combining the Query and Response into one item often reduces storage requirements due to commonality in the data of the two messages.

   In the context of generating a C-DNS file it is assumed that only those DNS payloads which can be parsed to produce a well-formed DNS message are stored in the C-DNS format and that all other messages will be (optionally) recorded as malformed messages. Parsing a well-formed message means as a minimum:

   * The packet has a well-formed 12 byte DNS Header with a recognised OPCODE.

   * The section counts are consistent with the section contents.

   * All of the resource records can be fully parsed.

2. All top level fields in each Q/R data item will be optional.

   * Rationale: Different users will have different requirements for data to be available for analysis. Users with minimal requirements should not have to pay the cost of recording full data, though this will limit the ability to perform certain kinds of data analysis and also to reconstruct packet captures. For example, omitting the resource records from a Response will reduce the C-DNS file size; in principle responses can be synthesized if there is enough context.

3. Multiple Q/R data items will be collected into blocks in the format. Common data in a block will be abstracted and referenced from individual Q/R data items by indexing. The maximum number of Q/R data items in a block will be configurable.

   * Rationale: This blocking and indexing provides a significant reduction in the volume of file data generated. Although this introduces complexity, it provides compression of the data that makes use of knowledge of the DNS message structure.

   * It is anticipated that the files produced can be subject to further compression using general purpose compression tools. Measurements show that blocking significantly reduces the CPU required to perform such strong compression. See Appendix C.2.

   * Examples of commonality between DNS messages are that in most cases the QUESTION RR is the same in the query and response, and that there is a finite set of query signatures (based on a subset of attributes). For many authoritative servers there is very likely to be a finite set of responses that are generated, of which a large number are NXDOMAIN.

4. Traffic metadata can optionally be included in each block. Specifically, counts of some types of non-DNS packets (e.g. ICMP, TCP resets) sent to the server may be of interest.

5. The wire format content of malformed DNS messages may optionally be recorded.

   * Rationale: Any structured capture format that does not capture the DNS payload byte for byte will be limited to some extent in that it cannot represent malformed DNS messages. Only those messages that can be fully parsed and transformed into the structured format can be fully represented. Note, however, this can result in rather misleading statistics. For example, a malformed query which cannot be represented in the C-DNS format will lead to the (well formed) DNS responses with error code FORMERR appearing as 'unmatched'. Therefore it can greatly aid downstream analysis to have the wire format of the malformed DNS messages available directly in the C-DNS file.

5. Choice of CBOR

This document presents a detailed format description using CBOR, the Concise Binary Object Representation defined in [RFC7049].

The choice of CBOR was made taking a number of factors into account.

- CBOR is a binary representation, and thus is economical in storage space.

- Other binary representations were investigated, and whilst all had attractive features, none had a significant advantage over CBOR. See Appendix C for some discussion of this.

- CBOR is an IETF standard and familiar to IETF participants. It is based on the now-common ideas of lists and objects, and thus requires very little familiarization for those in the wider industry.

- CBOR is a simple format, and can easily be implemented from scratch if necessary. More complex formats require library support which may present problems on unusual platforms.

- CBOR can also be easily converted to text formats such as JSON ([RFC8259]) for debugging and other human inspection requirements.

- CBOR data schemas can be described using CDDL [I-D.ietf-cbor-cddl].

6. C-DNS format conceptual overview

The following figures show purely schematic representations of the C-DNS format to convey the high-level structure of the C-DNS format. Section 7 provides a detailed discussion of the CBOR representation and individual elements.

Figure 1 shows the C-DNS format at the top level including the file header and data blocks. The Query/Response data items, Address/Event Count data items and Malformed Message data items link to various Block tables.

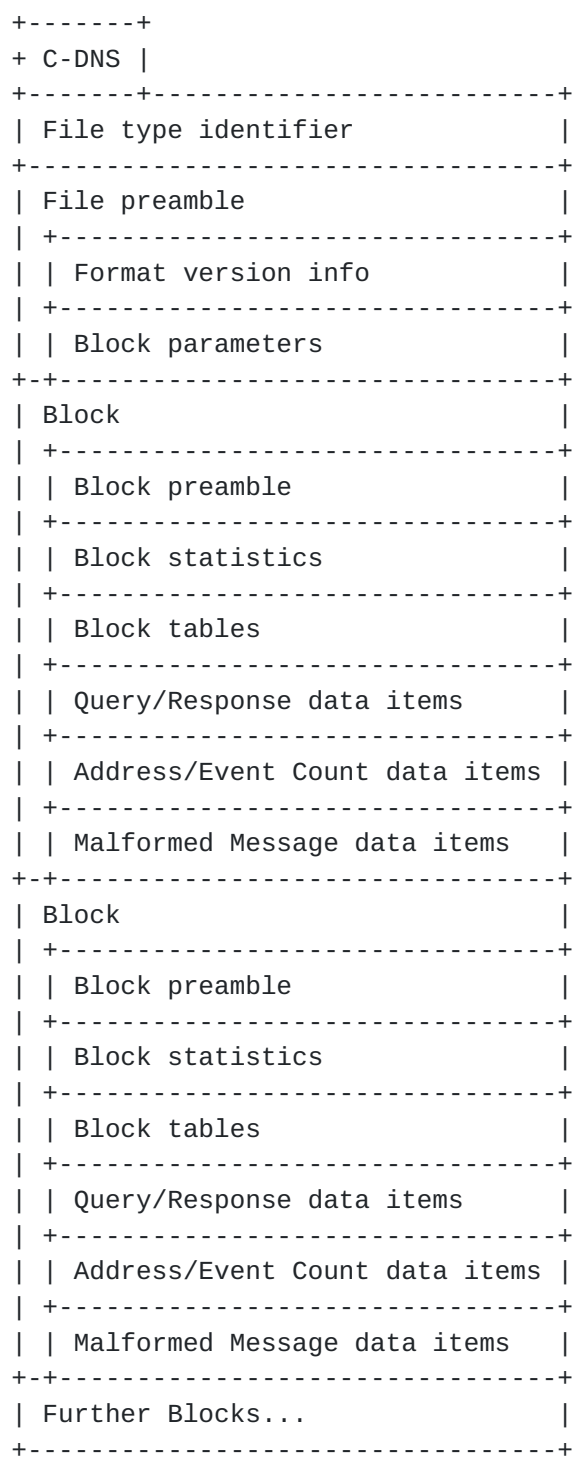
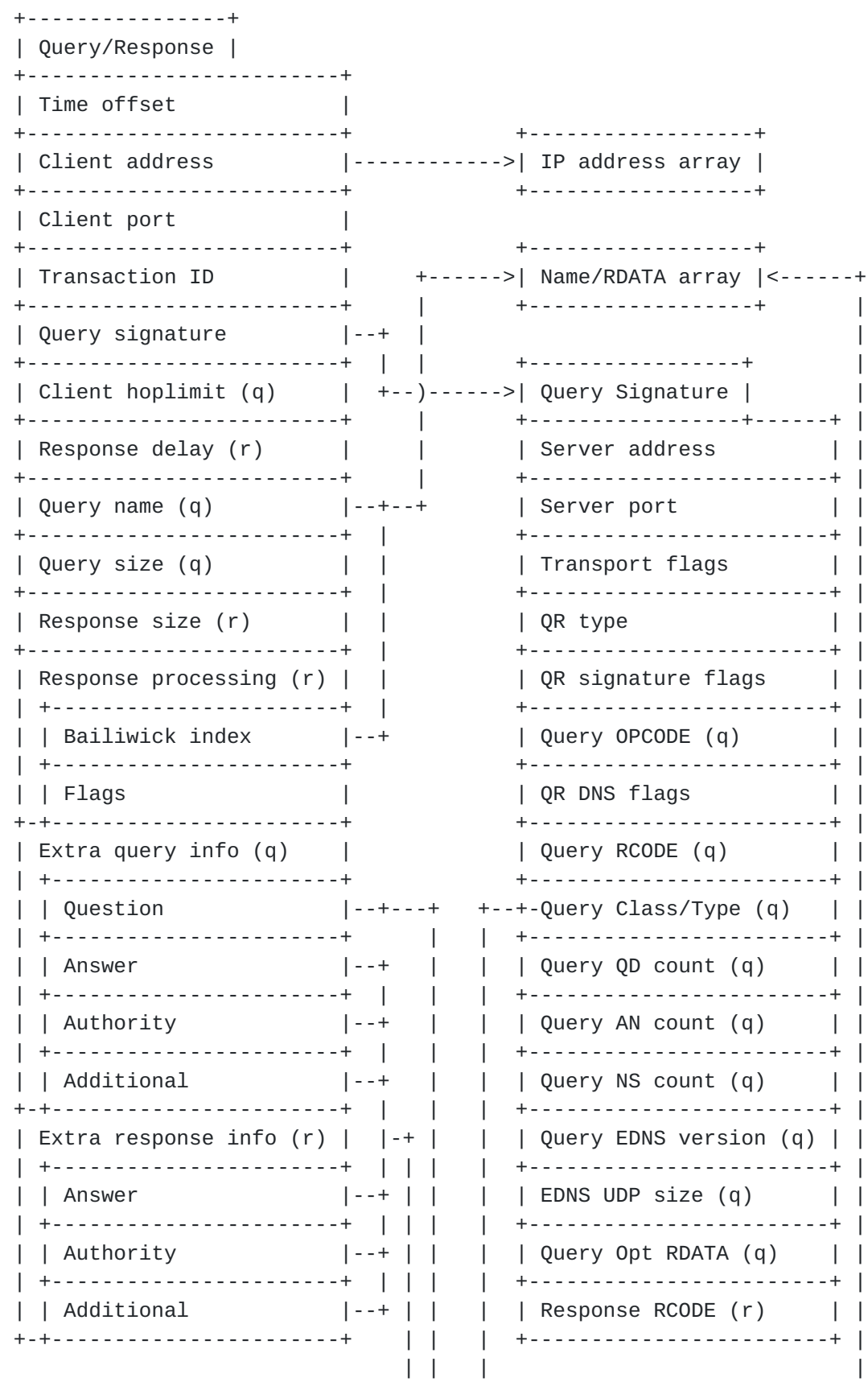


Figure 1: The C-DNS format.

Figure 2 shows some more detailed relationships within each block, specifically those between the Query/Response data item and the relevant Block tables.


Query/Response
Time offset
Client address
Client port
Transaction ID
Query signature
Client hoplimit (q)
Response delay (r)
Query name (q)
Query size (q)
Response size (r)
Response processing (r)
Bailiwick index
Flags
Extra query info (q)
Question
Answer
Authority
Additional
Extra response info (r)
Answer
Authority
Additional
IP address array
Name/RDATA array
Query Signature
Server address
Server port
Transport flags
QR type
QR signature flags
Query OPCODE (q)
QR DNS flags
Query RCODE (q)
Query Class/Type (q)
Query QD count (q)
Query AN count (q)
Query NS count (q)
Query EDNS version (q)
EDNS UDP size (q)
Query Opt RDATA (q)
Response RCODE (r)

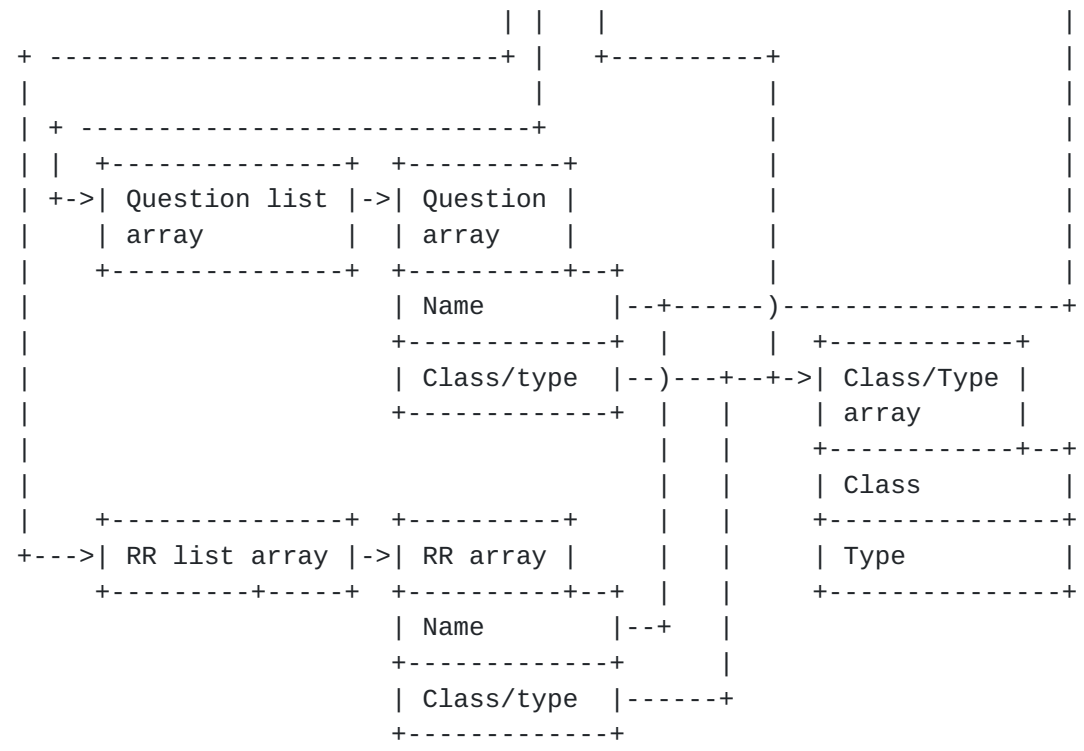


Figure 2: The Query/Response data item and subsidiary tables.

In Figure 2 data items annotated (q) are only present when a query/response has a query, and those annotated (r) are only present when a query/response response is present.

A C-DNS file begins with a file header containing a File Type Identifier and a File Preamble. The File Preamble contains information on the file Format Version and an array of Block Parameters items (the contents of which include Collection and Storage Parameters used for one or more blocks).

The file header is followed by a series of data Blocks.

A Block consists of a Block Preamble item, some Block Statistics for the traffic stored within the Block and then various arrays of common data collectively called the Block Tables. This is then followed by an array of the Query/Response data items detailing the queries and responses stored within the Block. The array of Query/Response data items is in turn followed by the Address/Event Counts data items (an array of per-client counts of particular IP events) and then Malformed Message data items (an array of malformed messages that stored in the Block).

The exact nature of the DNS data will affect what block size is the best fit, however sample data for a root server indicated that block

sizes up to 10,000 Q/R data items give good results. See Appendix C.6 for more details.

6.1. Block Parameters

The details of the Block Parameters items are not shown in the diagrams but are discussed here for context.

An array of Block Parameters items is stored in the File Preamble (with a minimum of one item at index 0); a Block Parameters item consists of a collection of Storage and Collection Parameters that applies to any given Block. An array is used in order to support use cases such as wanting to merge C-DNS files from different sources. The Block Preamble item then contains an optional index for the Block Parameters item that applies for that Block; if not present the index defaults to 0. Hence, in effect, a global Block Parameters item is defined which can then be overridden per Block.

6.2. Storage Parameters

The Block Parameters item includes a Storage Parameters item - this contains information about the specific data fields stored in the C-DNS file.

These parameters include:

- The sub-second timing resolution used by the data.
- Information (hints) on which optional data are omitted. See Section 6.2.1.
- Recorded OPCODES and RR types. See Section 6.2.2.
- Flags indicating, for example, whether the data is sampled or anonymised. See Section 6.2.3.
- Client and server IPv4 and IPv6 address prefixes. See Section 6.2.4

6.2.1. Optional data items

To enable implementations to store data to their precise requirements in as space-efficient manner as possible, all fields in the following arrays are optional:

- Query/Response
- Query Signature

- Malformed messages

In other words, an implementation can choose to omit any data item that is not required for its use case. In addition, implementations may be configured to not record all RRs, or only record messages with certain OPCODES.

This does, however, mean that a consumer of a C-DNS file faces two problems:

1. How can it quickly determine if a file definitely does not contain the data items it requires to complete a particular task (e.g. reconstructing query traffic or performing a specific piece of data analysis)?

2. How can it determine if a data item is not present because it was:

   * explicitly not recorded or

   * the data item was not available/present.

For example, capturing C-DNS data from within a nameserver implementation makes it unlikely that the Client Hoplimit can be recorded. Or, if there is no query ARCount recorded and no query OPT RDATA recorded, is that because no query contained an OPT RR, or because that data was not stored?

The Storage Parameters therefore also contains a Storage Hints item which specifies which items the encoder of the file omits from the stored data. An implementation decoding that file can then use these to quickly determine whether the input data is rich enough for its needs.

6.2.2. Optional RRs and OPCODEs

Also included in the Storage Parameters are explicit arrays listing the RR types and the OPCODEs to be recorded. These remove any ambiguity over whether messages containing particular OPCODEs or RR types are not present because they did not occur, or because the implementation is not configured to record them.

In the case of OPCODEs, for a message to be fully parsable, the OPCODE must be known to the collecting implementation. Any message with an OPCODE unknown to the collecting implementation cannot be validated as correctly formed, and so must be treated as malformed. Messages with OPCODES known to the recording application but not

listed in the Storage Parameters are discarded (regardless of whether they are malformed or not).

In the case of RR records, each record in a message must be fully parsable, including parsing the record RDATA, as otherwise the message cannot be validated as correctly formed. Any RR record with an RR type not known to the collecting implementation cannot be validated as correctly formed, and so must be treated as malformed.

Once a message is correctly parsed, an implementation is free to record only a subset of the RR records present.

6.2.3. Storage flags

The Storage Parameters contains flags that can be used to indicate if:

- the data is anonymised,
- the data is produced from sample data, or
- names in the data have been normalised (converted to uniform case).

The Storage Parameters also contains optional fields holding details of the sampling method used and the anonymisation method used. It is RECOMMENDED these fields contain URIs pointing to resources describing the methods used.

6.2.4. IP Address storage

The format contains fields to indicate if only IP prefixes were stored. If IP address prefixes are given, only the prefix bits of addresses are stored. For example, if a client IPv4 prefix of 16 is specified, a client address of 192.0.2.1 will be stored as 0xc000 (192.0), reducing address storage space requirements.

7. C-DNS format detailed description

The CDDL definition for the C-DNS format is given in Appendix A.

7.1. Map quantities and indexes

All map keys are integers with values specified in the CDDL. String keys would significantly bloat the file size.

All key values specified are positive integers under 24, so their CBOR representation is a single byte. Positive integer values not

currently used as keys in a map are reserved for use in future standard extensions.

Implementations may choose to add additional implementation-specific entries to any map. Negative integer map keys are reserved for these values. Key values from -1 to -24 also have a single byte CBOR representation, so such implementation-specific extensions are not at any space efficiency disadvantage.

An item described as an index is the index of the data item in the referenced array. Indexes are 0-based.

7.2. Tabular representation

The following sections present the C-DNS specification in tabular format with a detailed description of each item.

In all quantities that contain bit flags, bit 0 indicates the least significant bit, i.e. flag "n" in quantity "q" is on if "(q & (1 << n)) != 0".

For the sake of readability, all type and field names defined in the CDDL definition are shown in double quotes. Type names are by convention camel case (e.g. "BlockTable"), field names are lower-case with hyphens (e.g. "block-tables").

For the sake of brevity, the following conventions are used in the tables:

- The column O marks whether items in a map are optional.
  - O - Optional. The item may be omitted.
  - M - Mandatory. The item must be present.
- The column T gives the CBOR data type of the item.
  - U - Unsigned integer
  - I - Signed integer
  - B - Byte string
  - T - Text string
  - M - Map
  - A - Array

In the case of maps and arrays, more information on the type of each value, include the CDDL definition name if applicable, is given in the description.

7.3. "File"

A C-DNS file has an outer structure "File", a map that contains the following:

Field	O	T	Description
file-type-id	M	T	String "C-DNS" identifying the file type.
file-preamble	M	M	Version and parameter information for the whole file. Map of type "FilePreamble", see Section 7.4.
file-blocks	M	A	Array of items of type "Block", see Section 7.5. The array may be empty if the file contains no data.

7.4. "FilePreamble"

Information about data in the file. A map containing the following:

Field	O	T	Description
major-format-version	M	U	Unsigned integer '1'. The major version of format used in file.
minor-format-version	M	U	Unsigned integer '0'. The minor version of format used in file.
private-version	O	U	Version indicator available for private use by implementations.
block-parameters	M	A	Array of items of type "BlockParameters", see Section 7.4.1. The array must contain at least one entry. (The "block-parameters-index" item in each "BlockPreamble" indicates which array entry applies to that "Block".)

7.4.1. "BlockParameters"

Parameters relating to data storage and collection which apply to one or more items of type "Block". A map containing the following:

Field	O	T	Description
storage-parameters	M	M	Parameters relating to data storage in a "Block" item. Map of type "StorageParameters", see Section 7.4.1.1.
collection-parameters	O	M	Parameters relating to collection of the data in a "Block" item. Map of type "CollectionParameters", see Section 7.4.2.

7.4.1.1. "StorageParameters"

Parameters relating to how data is stored in the items of type "Block". A map containing the following:

Field	O	T	Description
ticks-per-second	M	U	Sub-second timing is recorded in ticks. This specifies the number of ticks in a second.
max-block-items	M	U	The maximum number of items stored in any of the arrays in a "Block" item (Q/R items, address event counts or malformed messages). An indication to a decoder of the resources needed to process the file.
storage-hints	M	M	Collection of hints as to which fields are omitted in the arrays that have optional fields. Map of type "StorageHints", see Section 7.4.1.1.1.
opcodes	M	A	Array of OPCODES (unsigned integers) recorded by the collection implementation. See Section 6.2.2.

rr-types	M	A	Array of RR types (unsigned integers) recorded by the collection implementation. See Section 6.2.2.	
storage-flags	O	U	Bit flags indicating attributes of stored data. Bit 0. 1 if the data has been anonymised. Bit 1. 1 if the data is sampled data. Bit 2. 1 if the names have been normalised (converted to uniform case).	
client-address-prefix-ipv4	O	U	IPv4 client address prefix length. If specified, only the address prefix bits are stored.	
client-address-prefix-ipv6	O	U	IPv6 client address prefix length. If specified, only the address prefix bits are stored.	
server-address-prefix-ipv4	O	U	IPv4 server address prefix length. If specified, only the address prefix bits are stored.	
server-address-prefix-ipv6	O	U	IPv6 server address prefix length. If specified, only the address prefix bits are stored.	
sampling-method	O	T	Information on the sampling method used. See Section 6.2.3.	
anonymisation-method	O	T	Information on the anonymisation method used. See Section 6.2.3.	

7.4.1.1.1. "StorageHints"

An indicator of which fields the collecting implementation omits in the arrays with optional fields. A map containing the following:

Field	O	T	Description
query-response-hints	M	U	Hints indicating which "QueryResponse" fields are omitted, see section Section 7.6. If the field is omitted the bit is unset.

			Bit 0. time-offset
			Bit 1. client-address-index
			Bit 2. client-port
			Bit 3. transaction-id
			Bit 4. qr-signature-index
			Bit 5. client-hoplimit
			Bit 6. response-delay
			Bit 7. query-name-index
			Bit 8. query-size
			Bit 9. response-size
			Bit 10. response-processing-data
			Bit 11. query-question-sections
			Bit 12. query-answer-sections
			Bit 13. query-authority-sections
			Bit 14. query-additional-sections
			Bit 15. response-answer-sections
			Bit 16. response-authority-sections
			Bit 17. response-additional-sections
query-response -signature-hints	M	U	Hints indicating which "QueryResponseSignature" fields are omitted, see section Section 7.5.3.2. If the field is omitted the bit is unset.
			Bit 0. server-address
			Bit 1. server-port
			Bit 2. qr-transport-flags
			Bit 3. qr-type
			Bit 4. qr-sig-flags
			Bit 5. query-opcode
			Bit 6. dns-flags
			Bit 7. query-rcode
			Bit 8. query-class-type
			Bit 9. query-qdcount
			Bit 10. query-ancount
			Bit 11. query-nscount
			Bit 12. query-arcount
			Bit 13. query-edns-version
			Bit 14. query-udp-size
			Bit 15. query-opt-rdata
			Bit 16. response-rcode
rr-hints	M	U	Hints indicating which optional "RR" fields are omitted, see Section 7.5.3.4. If the field is omitted the bit is unset.
			Bit 0. ttl
			Bit 1. rdata-index

other-data-hints	M	U	Hints indicating which other data types are are omitted. If the data type is are omitted the bit is unset. Bit 0. malformed-messages Bit 1. address-event-counts

7.4.2. "CollectionParameters"

Parameters relating to how data in the file was collected.

These parameters have no default. If they do not appear, nothing can be inferred about their value.

A map containing the following items:

Field	O	T	Description
query-timeout	O	U	To be matched with a query, a response must arrive within this number of seconds.
skew-timeout	O	U	The network stack may report a response before the corresponding query. A response is not considered to be missing a query until after this many micro-seconds.
snaplen	O	U	Collect up to this many bytes per packet.
promisc	O	U	1 if promiscuous mode was enabled on the interface, 0 otherwise.
interfaces	O	A	Array of identifiers (of type text string) of the interfaces used for collection.
server-addresses	O	A	Array of server collection IP addresses (of type byte string). Hint for downstream analysers; does not affect collection.
vlan-ids	O	A	Array of identifiers (of type unsigned integer) of VLANs selected for collection.
filter	O	T	"tcpdump" [pcap] style filter for input.
generator-id	O	T	String identifying the collection method.
host-id	O	T	String identifying the collecting host. Empty if converting an existing packet capture file.

7.5. "Block"

Container for data with common collection and and storage parameters. A map containing the following:

Field	O	T	Description
block-preamble	M	M	Overall information for the "Block" item. Map of type "BlockPreamble", see Section 7.5.1.
block-statistics	O	M	Statistics about the "Block" item. Map of type "BlockStatistics", see Section 7.5.2.
block-tables	O	M	The arrays containing data referenced by individual "QueryResponse" or "MalformedMessage" items. Map of type "BlockTables", see Section 7.5.3.
query-responses	O	A	Details of individual DNS Q/R data items. Array of items of type "QueryResponse", see Section 7.6. If present, the array must not be empty.
address-event-counts	O	A	Per client counts of ICMP messages and TCP resets. Array of items of type "AddressEventCount", see Section 7.7. If present, the array must not be empty.
malformed-messages	O	A	Details of malformed DNS messages. Array of items of type "MalformedMessage", see Section 7.8. If present, the array must not be empty.

7.5.1. "BlockPreamble"

Overall information for a "Block" item. A map containing the following:

Field	O	T	Description
earliest-time	O	A	A timestamp (2 unsigned integers, "Timestamp") for the earliest record in the "Block" item. The first integer is the number of seconds since the Posix epoch ("time_t"). The second integer is the number of ticks since the start of the second. This timestamp can only be omitted if all block items containing a time offset from the start of the block also omit that time offset.
block-parameters-index	O	U	The index of the item in the "block-parameters" array (in the "file-premable" item) applicable to this block. If not present, index 0 is used. See Section 7.4.1.

7.5.2. "BlockStatistics"

Basic statistical information about a "Block" item. A map containing the following:

Field	O	T	Description
processed-messages	O	U	Total number of DNS messages processed from the input traffic stream during collection of data in this "Block" item.
qr-data-items	O	U	Total number of Q/R data items in this "Block" item.
unmatched-queries	O	U	Number of unmatched queries in this "Block" item.
unmatched-responses	O	U	Number of unmatched responses in this "Block" item.
discarded-opcode	O	U	Number of DNS messages processed from the input traffic stream during collection of data in this "Block" item but not recorded because their OPCODE is not in the list to be collected.
malformed-items	O	U	Number of malformed messages found in input for this "Block" item.

7.5.3. "BlockTables"

Arrays containing data referenced by individual "QueryResponse" or "MalformedMessage" items in this "Block". Each element is an array which, if present, must not be empty.

An item in the "qlist" array contains indexes to values in the "qrr" array. Therefore, if "qlist" is present, "qrr" must also be present. Similarly, if "rrlist" is present, "rr" must also be present.

The map contains the following items:

Field	O	T	Description
ip-address	O	A	Array of IP addresses, in network byte order (of type byte string). If client or server address prefixes are set, only the address prefix bits are stored. Each string is therefore up

			to 4 bytes long for an IPv4 address, or up to 16 bytes long for an IPv6 address. See Section 7.4.1.1.
classtype	O	A	Array of RR class and type information. Type is "ClassType", see Section 7.5.3.1.
name-rdata	O	A	Array where each entry is the contents of a single NAME or RDATA (of type byte string). Note that NAMEs, and labels within RDATA contents, are full domain names or labels; no DNS style name compression is used on the individual names/labels within the format.
qr-sig	O	A	Array Q/R data item signatures. Type is "QueryResponseSignature", see Section 7.5.3.2.
qlist	O	A	Array of type "QuestionList". A "QuestionList" is an array of unsigned integers, indexes to "Question" items in the "qrr" array.
qrr	O	A	Array of type "Question". Each entry is the contents of a single question, where a question is the second or subsequent question in a query. See Section 7.5.3.3.
rrlist	O	A	Array of type "RRList". An "RRList" is an array of unsigned integers, indexes to "RR" items in the "rr" array.
rr	O	A	Array of type "RR". Each entry is the contents of a single RR. See Section 7.5.3.4.
malformed-message-data	O	A	Array of the contents of malformed messages. Array of type "MalformedMessageData", see Section 7.5.3.5.

7.5.3.1. "ClassType"

RR class and type information. A map containing the following:

Field	O	T	Description
type	M	U	TYPE value.
class	M	U	CLASS value.

7.5.3.2. "QueryResponseSignature"

Elements of a Q/R data item that are often common between multiple individual Q/R data items. A map containing the following:

Field	O	T	Description
server-address-index	O	U	The index in the item in the "ip-address" array of the server IP address. See Section 7.5.3.
server-port	O	U	The server port.
qr-transport-flags	O	U	Bit flags describing the transport used to service the query. Bit 0. IP version. 0 if IPv4, 1 if IPv6 Bit 1-4. Transport. 4 bit unsigned value where 0 = UDP, 1 = TCP, 2 = TLS, 3 = DTLS. Values 4-15 are reserved for future use. Bit 5. 1 if trailing bytes in query packet. See Section 11.2.
qr-type	O	U	Type of Query/Response transaction. 0 = Stub. A query from a stub resolver. 1 = Client. An incoming query to a recursive resolver. 2 = Resolver. A query sent from a recursive resolver to an authorative resolver. 3 = Authorative. A query to an authorative resolver. 4 = Forwarder. A query sent from a

			recursive resolver to an upstream recursive resolver. 5 = Tool. A query sent to a server by a server tool.
qr-sig-flags	O	U	Bit flags explicitly indicating attributes of the message pair represented by this Q/R data item (not all attributes may be recorded or deducible). Bit 0. 1 if a Query was present. Bit 1. 1 if a Response was present. Bit 2. 1 if a Query was present and it had an OPT Resource Record. Bit 3. 1 if a Response was present and it had an OPT Resource Record. Bit 4. 1 if a Query was present but had no Question. Bit 5. 1 if a Response was present but had no Question (only one query-name-index is stored per Q/R item).
query-opcode	O	U	Query OPCODE.
qr-dns-flags	O	U	Bit flags with values from the Query and Response DNS flags. Flag values are 0 if the Query or Response is not present. Bit 0. Query Checking Disabled (CD). Bit 1. Query Authenticated Data (AD). Bit 2. Query reserved (Z). Bit 3. Query Recursion Available (RA). Bit 4. Query Recursion Desired (RD). Bit 5. Query TrunCation (TC). Bit 6. Query Authoritative Answer (AA). Bit 7. Query DNSSEC answer OK (DO). Bit 8. Response Checking Disabled (CD). Bit 9. Response Authenticated Data (AD). Bit 10. Response reserved (Z). Bit 11. Response Recursion Available (RA). Bit 12. Response Recursion Desired (RD).

			Bit 13. Response TrunCation (TC).
			Bit 14. Response Authoritative Answer (AA).
query-rcode	O	U	Query RCODE. If the Query contains OPT, this value incorporates any EXTENDED_RCODE_VALUE.
query-classtype-index	O	U	The index to the item in the the "classtype" array of the CLASS and TYPE of the first Question. See Section 7.5.3.
query-qd-count	O	U	The QDCOUNT in the Query, or Response if no Query present.
query-an-count	O	U	Query ANCOUNT.
query-ns-count	O	U	Query NSCOUNT.
query-ar-count	O	U	Query ARCOUNT.
edns-version	O	U	The Query EDNS version.
udp-buf-size	O	U	The Query EDNS sender's UDP payload size.
opt-rdata-index	O	U	The index in the "name-rdata" array of the OPT RDATA. See Section 7.5.3.
response-rcode	O	U	Response RCODE. If the Response contains OPT, this value incorporates any EXTENDED_RCODE_VALUE.

7.5.3.3. "Question"

Details on individual Questions in a Question section. A map containing the following:

Field	O	T	Description
name-index	M	U	The index in the "name-rdata" array of the QNAME. See Section 7.5.3.
classtype-index	M	U	The index in the "classtype" array of the CLASS and TYPE of the Question. See Section 7.5.3.

7.5.3.4. "RR"

Details on individual Resource Records in RR sections. A map containing the following:

Field	O	T	Description
name-index	M	U	The index in the "name-rdata" array of the NAME. See Section 7.5.3.
classtype-index	M	U	The index in the "classtype" array of the CLASS and TYPE of the RR. See Section 7.5.3.
ttl	O	U	The RR Time to Live.
rdata-index	O	U	The index in the "name-rdata" array of the RR RDATA. See Section 7.5.3.

7.5.3.5. "MalformedMessageData"

Details on malformed message items in this "Block" item. A map containing the following:

Field	O	T	Description
server-address-index	O	U	The index in the "ip-address" array of the server IP address. See Section 7.5.3.
server-port	O	U	The server port.
mm-transport-flags	O	U	Bit flags describing the transport used to service the query. Bit 0 is the least significant bit. Bit 0. IP version. 0 if IPv4, 1 if IPv6 Bit 1-4. Transport. 4 bit unsigned value where 0 = UDP, 1 = TCP, 2 = TLS, 3 = DTLS. Values 4-15 are reserved for future use.
mm-payload	O	B	The payload (raw bytes) of the DNS message.

7.6. "QueryResponse"

Details on individual Q/R data items.

Note that there is no requirement that the elements of the "query-responses" array are presented in strict chronological order.

A map containing the following items:

Field	O	T	Description
time-offset	O	U	Q/R timestamp as an offset in ticks from "earliest-time". The timestamp is the timestamp of the Query, or the Response if there is no Query.
client-address-index	O	U	The index in the "ip-address" array of the client IP address. See Section 7.5.3.
client-port	O	U	The client port.
transaction-id	O	U	DNS transaction identifier.

qr-signature-index	O	U	The index in the "qr-sig" array of the "QueryResponseSignature" item. See Section 7.5.3.
client-hoplimit	O	U	The IPv4 TTL or IPv6 Hoplimit from the Query packet.
response-delay	O	I	The time difference between Query and Response, in ticks. Only present if there is a query and a response. The delay can be negative if the network stack/capture library returns packets out of order.
query-name-index	O	U	The index in the "name-rdata" array of the item containing the QNAME for the first Question. See Section 7.5.3.
query-size	O	U	DNS query message size (see below).
response-size	O	U	DNS query message size (see below).
response-processing-data	O	M	Data on response processing. Map of type "ResponseProcessingData", see Section 7.6.1.
query-extended	O	M	Extended Query data. Map of type "QueryResponseExtended", see Section 7.6.2.
response-extended	O	M	Extended Response data. Map of type "QueryResponseExtended", see Section 7.6.2.

The "query-size" and "response-size" fields hold the DNS message size. For UDP this is the size of the UDP payload that contained the DNS message. For TCP it is the size of the DNS message as specified in the two-byte message length header. Trailing bytes in UDP queries are routinely observed in traffic to authoritative servers and this value allows a calculation of how many trailing bytes were present.

7.6.1. "ResponseProcessingData"

Information on the server processing that produced the response. A map containing the following:

Field	O	T	Description
bailiwick-index	O	U	The index in the "name-rdata" array of the owner name for the response bailiwick. See Section 7.5.3.
processing-flags	O	U	Flags relating to response processing. Bit 0. 1 if the response came from cache.

7.6.2. "QueryResponseExtended"

Extended data on the Q/R data item.

Each item in the map is present only if collection of the relevant details is configured.

A map containing the following items:

Field	O	T	Description
question-index	O	U	The index in the "qlist" array of the entry listing any second and subsequent Questions in the Question section for the Query or Response. See Section 7.5.3.
answer-index	O	U	The index in the "rrlist" array of the entry listing the Answer Resource Record sections for the Query or Response. See Section 7.5.3.
authority-index	O	U	The index in the "rrlist" array of the entry listing the Authority Resource Record sections for the Query or Response. See Section 7.5.3.
additional-index	O	U	The index in the "rrlist" array of the entry listing the Additional Resource Record sections for the Query or Response. See Section 7.5.3. Note that Query OPT RR data can be optionally stored in the QuerySignature.

7.7. "AddressEventCount"

Counts of various IP related events relating to traffic with individual client addresses. A map containing the following:

Field	O	T	Description
ae-type	M	U	The type of event. The following events types are currently defined: 0. TCP reset. 1. ICMP time exceeded. 2. ICMP destination unreachable. 3. ICMPv6 time exceeded. 4. ICMPv6 destination unreachable. 5. ICMPv6 packet too big.
ae-code	O	U	A code relating to the event.
ae-address-index	M	U	The index in the "ip-address" array of the client address. See Section 7.5.3.
ae-count	M	U	The number of occurrences of this event during the block collection period.

7.8. "MalformedMessage"

Details of malformed messages. A map containing the following:

Field	O	T	Description
time-offset	O	U	Message timestamp as an offset in ticks from "earliest-time".
client-address-index	O	U	The index in the "ip-address" array of the client IP address. See Section 7.5.3.
client-port	O	U	The client port.
message-data-index	O	U	The index in the "malformed-message-data" array of the message data for this message. See Section 7.5.3.

8. Versioning

The C-DNS file preamble includes a file format version; a major and minor version number are required fields. The document defines version 1.0 of the C-DNS specification. This section describes the intended use of these version numbers in future specifications.

It is noted that version 1.0 includes many optional fields and therefore consumers of version 1.0 should be inherently robust to parsing files with variable data content.

Within a major version, a new minor version MUST be a strict superset of the previous minor version, with no semantic changes to existing fields. New keys MAY be added to existing maps, and new maps MAY be added. A consumer capable of reading a particular major.minor version MUST also be capable of reading all previous minor versions of the same major version. It SHOULD also be capable of parsing all subsequent minor versions ignoring any keys or maps that it does not recognise.

A new major version indicates changes to the format that are not backwards compatible with previous major versions. A consumer capable of only reading a particular major version (greater than 1) is not required to and has no expectation to be capable of reading a previous major version.

9. C-DNS to PCAP

It is possible to re-construct PCAP files from the C-DNS format in a lossy fashion. Some of the issues with reconstructing both the DNS payload and the full packet stream are outlined here.

The reconstruction depends on whether or not all the optional sections of both the query and response were captured in the C-DNS file. Clearly, if they were not all captured, the reconstruction will be imperfect.

Even if all sections of the response were captured, one cannot reconstruct the DNS response payload exactly due to the fact that some DNS names in the message on the wire may have been compressed. Section 9.1 discusses this is more detail.

Some transport information is not captured in the C-DNS format. For example, the following aspects of the original packet stream cannot be re-constructed from the C-DNS format:

- IP fragmentation

- TCP stream information:
  - Multiple DNS messages may have been sent in a single TCP segment
  - A DNS payload may have be split across multiple TCP segments
  - Multiple DNS messages may have be sent on a single TCP session
- Malformed DNS messages if the wire format is not recorded
- Any Non-DNS messages that were in the original packet stream e.g. ICMP

Simple assumptions can be made on the reconstruction: fragmented and DNS-over-TCP messages can be reconstructed into single packets and a single TCP session can be constructed for each TCP packet.

Additionally, if malformed messages and Non-DNS packets are captured separately, they can be merged with packet captures reconstructed from C-DNS to produce a more complete packet stream.

9.1. Name compression

All the names stored in the C-DNS format are full domain names; no DNS style name compression is used on the individual names within the format. Therefore when reconstructing a packet, name compression must be used in order to reproduce the on the wire representation of the packet.

[RFC1035] name compression works by substituting trailing sections of a name with a reference back to the occurrence of those sections earlier in the message. Not all name server software uses the same algorithm when compressing domain names within the responses. Some attempt maximum recompression at the expense of runtime resources, others use heuristics to balance compression and speed and others use different rules for what is a valid compression target.

This means that responses to the same question from different name server software which match in terms of DNS payload content (header, counts, RRs with name compression removed) do not necessarily match byte-for-byte on the wire.

Therefore, it is not possible to ensure that the DNS response payload is reconstructed byte-for-byte from C-DNS data. However, it can at least, in principle, be reconstructed to have the correct payload length (since the original response length is captured) if there is enough knowledge of the commonly implemented name compression

algorithms. For example, a simplistic approach would be to try each algorithm in turn to see if it reproduces the original length, stopping at the first match. This would not guarantee the correct algorithm has been used as it is possible to match the length whilst still not matching the on the wire bytes but, without further information added to the C-DNS data, this is the best that can be achieved.

Appendix B presents an example of two different compression algorithms used by well-known name server software.

10. Data collection

This section describes a non-normative proposed algorithm for the processing of a captured stream of DNS queries and responses and production of a stream of query/response items, matching queries/responses where possible.

For the purposes of this discussion, it is assumed that the input has been pre-processed such that:

1. All IP fragmentation reassembly, TCP stream reassembly, and so on, has already been performed.

2. Each message is associated with transport metadata required to generate the Primary ID (see Section 10.2.1).

3. Each message has a well-formed DNS header of 12 bytes and (if present) the first Question in the Question section can be parsed to generate the Secondary ID (see below). As noted earlier, this requirement can result in a malformed query being removed in the pre-processing stage, but the correctly formed response with RCODE of FORMERR being present.

DNS messages are processed in the order they are delivered to the implementation.

It should be noted that packet capture libraries do not necessarily provide packets in strict chronological order. This can, for example, arise on multi-core platforms where packets arriving at a network device are processed by different cores. On systems where this behaviour has been observed, the timestamps associated with each packet are consistent; queries always have a timestamp prior to the response timestamp. However, the order in which these packets appear in the packet capture stream is not necessarily strictly choronological; a response can appear in the capture stream before the query that provoked the response. For this discussion, this non-chronological delivery is termed "skew".

In the presence of skew, a response packets can arrive for matching before the corresponding query. To avoid generating false instances of responses without a matching query, and queries without a matching response, the matching algorithm must take account of the possibility of skew.

10.1. Matching algorithm

A schematic representation of the algorithm for matching Q/R data items is shown in Figure 3. It takes individual DNS query or response messages as input, and outputs matched Q/R items. The numbers in the figure identify matching operations listed in Table 1. Specific details of the algorithm, for example queues, timers and identifiers, are given in the following sections.

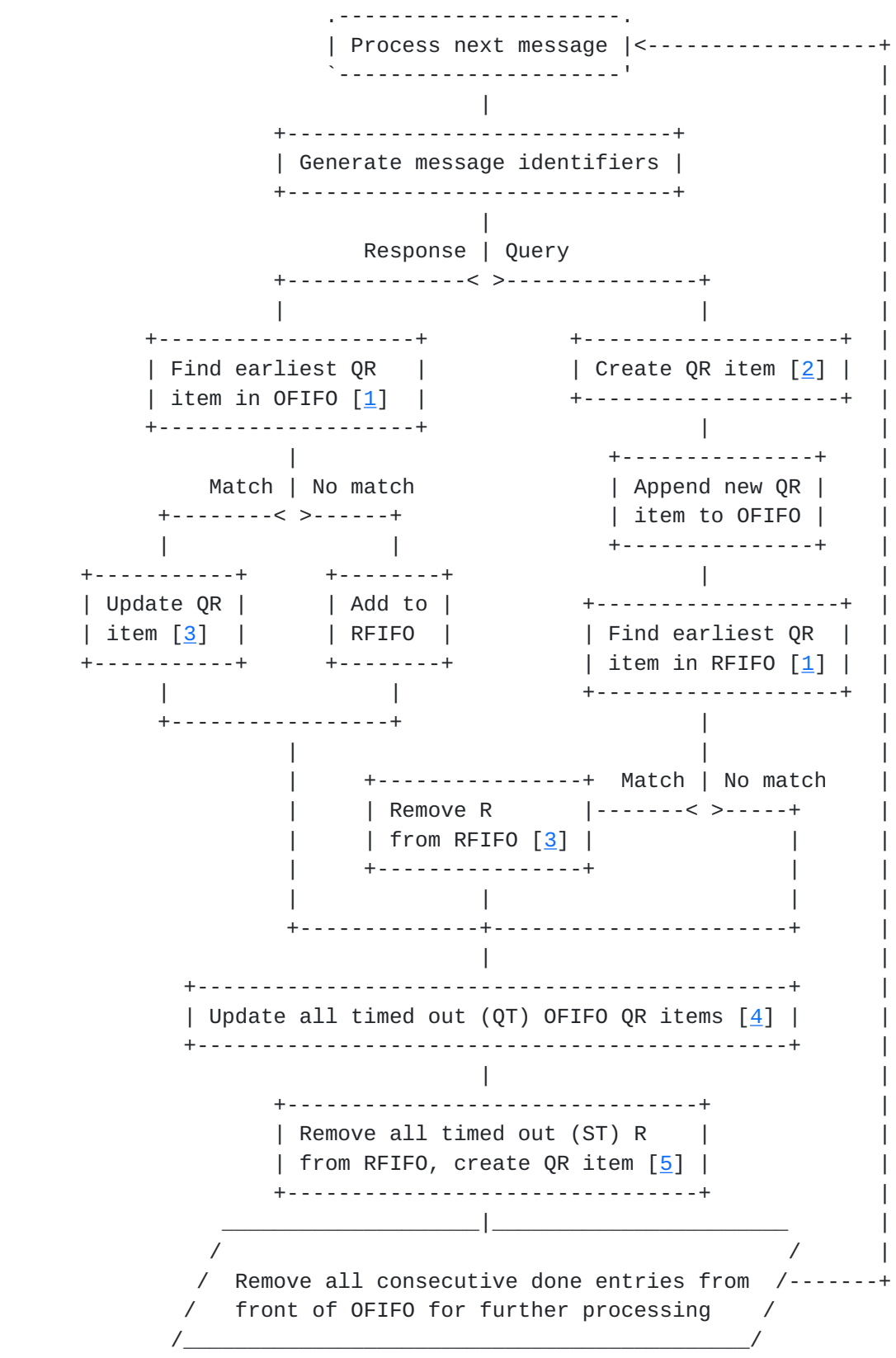


Figure 3: Query/Response matching algorithm

Ref	Operation
[1]	Find earliest QR item in FIFO where: * QR.done = false * QR.Q.PrimaryID == R.PrimaryID and, if both QR.Q and R have SecondaryID: * QR.Q.SecondaryID == R.SecondaryID
[2]	Set: QR.Q := Q QR.R := nil QR.done := false
[3]	Set: QR.R := R QR.done := true
[4]	Set: QR.done := true
[5]	Set: QR.Q := nil QR.R := R QR.done := true

Table 1: Operations used in the matching algorithm

10.2. Message identifiers

10.2.1. Primary ID (required)

A Primary ID is constructed for each message. It is composed of the following data:

1. Source IP Address

2. Destination IP Address

3. Source Port

4. Destination Port

5. Transport

6. DNS Message ID

10.2.2. Secondary ID (optional)

If present, the first Question in the Question section is used as a secondary ID for each message. Note that there may be well formed DNS queries that have a QDCOUNT of 0, and some responses may have a QDCOUNT of 0 (for example, responses with RCODE=FORMERR or NOTIMP). In this case the secondary ID is not used in matching.

10.3. Algorithm parameters

1. Query timeout, QT. A query arrives with timestamp t1. If no response matching that query has arrived before other input arrives timestamped later than (t1 + QT), a query/response item containing only a query item is recorded. The query timeout value is typically of the order of 5 seconds.

2. Skew timeout, ST. A response arrives with timestamp t2. If a response has not been matched by a query before input arrives timestamped later than (t2 + ST), a query/response item containing only a response is recorded. The skew timeout value is typically a few microseconds.

10.4. Algorithm requirements

The algorithm is designed to handle the following input data:

1. Multiple queries with the same Primary ID (but different Secondary ID) arriving before any responses for these queries are seen.

2. Multiple queries with the same Primary and Secondary ID arriving before any responses for these queries are seen.

3. Queries for which no later response can be found within the specified timeout.

4. Responses for which no previous query can be found within the specified timeout.

10.5. Algorithm limitations

For cases 1 and 2 listed in the above requirements, it is not possible to unambiguously match queries with responses. This algorithm chooses to match to the earliest query with the correct Primary and Secondary ID.

10.6. Workspace

The algorithm employs two FIFO queues:

- OFIFO, an output FIFO containing Q/R items in chronological order,
- RFIFO, a FIFO holding responses without a matching query in order of arrival.

10.7. Output

The output is a list of Q/R data items. Both the Query and Response elements are optional in these items, therefore Q/R data items have one of three types of content:

1. A matched pair of query and response messages
2. A query message with no response
3. A response message with no query

The timestamp of a list item is that of the query for cases 1 and 2 and that of the response for case 3.

10.8. Post processing

When ending capture, all items in the responses FIFO are timed out immediately, generating response-only entries to the Q/R data item FIFO. These and all other remaining entries in the Q/R data item FIFO should be treated as timed out queries.

11. Implementation guidance

Whilst this document makes no specific recommendations with respect to Canonical CBOR (see Section 3.9 of [RFC7049]) the following guidance may be of use to implementors.

Adherence to the first two rules given in Section 3.9 of [RFC7049] will minimise file sizes.

Adherence to the last two rules given in Section 3.9 of [RFC7049] for all maps and arrays would unacceptably constrain implementations, for example, in the use case of real-time data collection in constrained environments.

11.1. Optional data

When decoding C-DNS data some of the items required for a particular function that the consumer wishes to perform may be missing. Consumers should consider providing configurable default values to be used in place of the missing values in their output.

11.2. Trailing bytes

A DNS query message in a UDP or TCP payload can be followed by some additional (spurious) bytes, which are not stored in C-DNS.

When DNS traffic is sent over TCP, each message is prefixed with a two byte length field which gives the message length, excluding the two byte length field. In this context, trailing bytes can occur in two circumstances with different results:

1. The number of bytes consumed by fully parsing the message is less than the number of bytes given in the length field (i.e. the length field is incorrect and too large). In this case, the surplus bytes are considered trailing bytes in an analogous manner to UDP and recorded as such. If only this case occurs it is possible to process a packet containing multiple DNS messages where one or more has trailing bytes.

2. There are surplus bytes between the end of a well-formed message and the start of the length field for the next message. In this case the first of the surplus bytes will be processed as the first byte of the next length field, and parsing will proceed from there, almost certainly leading to the next and any subsequent messages in the packet being considered malformed. This will not generate a trailing bytes record for the processed well-formed message.

11.3. Limiting collection of RDATA

Implementations should consider providing a configurable maximum RDATA size for capture, for example, to avoid memory issues when confronted with large XFR records.

12. Implementation status

[Note to RFC Editor: please remove this section and reference to [RFC7942] prior to publication.]

This section records the status of known implementations of the protocol defined by this specification at the time of posting of this Internet-Draft, and is based on a proposal described in [RFC7942].

The description of implementations in this section is intended to assist the IETF in its decision processes in progressing drafts to RFCs. Please note that the listing of any individual implementation here does not imply endorsement by the IETF. Furthermore, no effort has been spent to verify the information presented here that was supplied by IETF contributors. This is not intended as, and must not be construed to be, a catalog of available implementations or their features. Readers are advised to note that other implementations may exist.

According to [RFC7942], "this will allow reviewers and working groups to assign due consideration to documents that have the benefit of running code, which may serve as evidence of valuable experimentation and feedback that have made the implemented protocols more mature. It is up to the individual working groups to use this information as they see fit".

12.1. DNS-STATS Compactor

ICANN/Sinodun IT have developed an open source implementation called DNS-STATS Compactor. The Compactor is a suite of tools which can capture DNS traffic (from either a network interface or a PCAP file) and store it in the Compacted-DNS (C-DNS) file format. PCAP files for the captured traffic can also be reconstructed. See Compactor [1].

This implementation:

- covers the whole of the specification described in the -03 draft with the exception of support for malformed messages and pico second time resolution. (Note: this implementation does allow malformed messages to be recorded separately in a PCAP file).

- is released under the Mozilla Public License Version 2.0.

- has a users mailing list available, see dns-stats-users [2].

There is also some discussion of issues encountered during development available at Compressing Pcap Files [3] and Packet Capture [4].

This information was last updated on 3rd of May 2018.

13. IANA considerations

None

14. Security considerations

Any control interface MUST perform authentication and encryption.

Any data upload MUST be authenticated and encrypted.

15. Acknowledgements

The authors wish to thank CZ.NIC, in particular Tomas Gavenciak, for many useful discussions on binary formats, compression and packet matching. Also Jan Vcelak and Wouter Wijngaards for discussions on name compression and Paul Hoffman for a detailed review of the document and the C-DNS CDDL.

Thanks also to Robert Edmonds, Jerry Lundstroem, Richard Gibson, Stephane Bortzmeyer and many other members of DNSOP for review.

Also, Miek Gieben for mmark [5]

16. Changelog

draft-ietf-dnsop-dns-capture-format-08

- Convert diagrams to ASCII
- Describe versioning
- Fix unused group warning in CDDL

draft-ietf-dnsop-dns-capture-format-07

- Resolve outstanding questions and TODOs
- Make RR RDATA optional
- Update matching diagram and explain skew
- Add count of discarded messages to block statistics
- Editorial clarifications and improvements

draft-ietf-dnsop-dns-capture-format-06

- Correct BlockParameters type to map
- Make RR ttl optional
- Add storage flag indicating name normalisation

- Add storage parameter fields for sampling and anonymisation methods
- Editorial clarifications and improvements

draft-ietf-dnsop-dns-capture-format-05

- Make all data items in Q/R, QuerySignature and Malformed Message arrays optional
- Re-structure the FilePreamble and ConfigurationParameters into BlockParameters
- BlockParameters has separate Storage and Collection Parameters
- Storage Parameters includes information on what optional fields are present, and flags specifying anonymisation or sampling
- Addresses can now be stored as prefixes.
- Switch to using a variable sub-second timing granularity
- Add response bailiwick and query response type
- Add specifics of how to record malformed messages
- Add implementation guidance
- Improve terminology and naming consistency

draft-ietf-dnsop-dns-capture-format-04

- Correct query-d0 to query-do in CDDL
- Clarify that map keys are unsigned integers
- Add Type to Class/Type table
- Clarify storage format in section 7.12

draft-ietf-dnsop-dns-capture-format-03

- Added an Implementation Status section

draft-ietf-dnsop-dns-capture-format-02

- Update qr_data_format.png to match CDDL

- Editorial clarifications and improvements

draft-ietf-dnsop-dns-capture-format-01

- Many editorial improvements by Paul Hoffman
- Included discussion of malformed message handling
- Improved Appendix C on Comparison of Binary Formats
- Now using C-DNS field names in the tables in section 8
- A handful of new fields included (CDDL updated)
- Timestamps now include optional picoseconds
- Added details of block statistics

draft-ietf-dnsop-dns-capture-format-00

- Changed dnstap.io to dnstap.info
- qr_data_format.png was cut off at the bottom
- Update authors address
- Improve wording in Abstract
- Changed DNS-STAT to C-DNS in CDDL
- Set the format version in the CDDL
- Added a TODO: Add block statistics
- Added a TODO: Add extend to support pico/nano. Also do this for Time offset and Response delay
- Added a TODO: Need to develop optional representation of malformed messages within C-DNS and what this means for packet matching. This may influence which fields are optional in the rest of the representation.
- Added section on design goals to Introduction
- Added a TODO: Can Class be optimised? Should a class of IN be inferred if not present?

draft-dickinson-dnsop-dns-capture-format-00

- Initial commit

17. References

17.1. Normative References

[RFC1035] Mockapetris, P., "Domain names - implementation and specification", STD 13, RFC 1035, DOI 10.17487/RFC1035, November 1987, <https://www.rfc-editor.org/info/rfc1035>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC7049] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", RFC 7049, DOI 10.17487/RFC7049, October 2013, <https://www.rfc-editor.org/info/rfc7049>.

17.2. Informative References

[ditl] DNS-OARC, "DITL", 2016, <https://www.dns-oarc.net/oarc/data/ditl>.

[dnscap] DNS-OARC, "DNSCAP", 2016, <https://www.dns-oarc.net/tools/dnscap>.

[dnstap] dnstap.info, "dnstap", 2016, <http://dnstap.info/>.

[dsc] Wessels, D. and J. Lundstrom, "DSC", 2016, <https://www.dns-oarc.net/tools/dsc>.

[I-D.daley-dnsxml] Daley, J., Morris, S., and J. Dickinson, "dnsxml - A standard XML representation of DNS data", draft-daley-dnsxml-00 (work in progress), July 2013.

[I-D.hoffman-dns-in-json] Hoffman, P., "Representing DNS Messages in JSON", draft-hoffman-dns-in-json-16 (work in progress), May 2018.

[I-D.ietf-cbor-cddl] Birkholz, H., Vigano, C., and C. Bormann, "Concise data definition language (CDDL): a notational convention to express CBOR data structures", draft-ietf-cbor-cddl-03 (work in progress), July 2018.

[packetq] .SE - The Internet Infrastructure Foundation, "PacketQ", 2014, <https://github.com/dotse/PacketQ>.

[pcap] tcpdump.org, "PCAP", 2016, <http://www.tcpdump.org/>.

[pcapng] Tuexen, M., Risso, F., Bongertz, J., Combs, G., and G. Harris, "pcap-ng", 2016, <https://github.com/pcapng/pcapng>.

[RFC7942] Sheffer, Y. and A. Farrel, "Improving Awareness of Running Code: The Implementation Status Section", BCP 205, RFC 7942, DOI 10.17487/RFC7942, July 2016, <https://www.rfc-editor.org/info/rfc7942>.

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/info/rfc8259>.

17.3. URIs

[1] https://github.com/dns-stats/compactor/wiki

[2] https://mm.dns-stats.org/mailman/listinfo/dns-stats-users

[3] https://www.sinodun.com/2017/06/compressing-pcap-files/

[4] https://www.sinodun.com/2017/06/more-on-debian-jessieubuntu-trusty-packet-capture-woes/

[5] https://github.com/miekg/mmark

[6] https://www.nlnetlabs.nl/projects/nsd/

[7] https://www.knot-dns.cz/

[8] https://avro.apache.org/

[9] https://developers.google.com/protocol-buffers/

[10] http://cbor.io

[11] https://github.com/kubo/snzip

[12] http://google.github.io/snappy/

[13] http://lz4.github.io/lz4/

[14] http://www.gzip.org/

[15] http://facebook.github.io/zstd/

[16] http://tukaani.org/xz/

[17] https://github.com/dns-stats/draft-dns-capture-format/blob/master/file-size-versus-block-size.png

[18] https://github.com/dns-stats/draft-dns-capture-format/blob/master/file-size-versus-block-size.svg

Appendix A. CDDL

This appendix gives a CDDL [I-D.ietf-cbor-cddl] specification for C-DNS.

CDDL does not permit a range of allowed values to be specified for a bitfield. Where necessary, those values are given as a CDDL group, but the group definition is commented out to prevent CDDL tooling from warning that the group is unused.

```
; CDDL specification of the file format for C-DNS,
; which describes a collection of DNS messages and
; traffic meta-data.

;
; The overall structure of a file.
;
File = [
    file-type-id  : tstr .regexp "C-DNS",
    file-preamble : FilePreamble,
    file-blocks   : [* Block],
]

;
; The file preamble.
;
FilePreamble = {
    major-format-version => uint .eq 1,
    minor-format-version => uint .eq 0,
    ? private-version    => uint,
    block-parameters     => [+ BlockParameters],
}
major-format-version = 0
minor-format-version = 1
private-version      = 2
block-parameters     = 3
```

```
BlockParameters = {
    storage-parameters      => StorageParameters,
    ? collection-parameters => CollectionParameters,
}
storage-parameters    = 0
collection-parameters = 1

  StorageParameters = {
      ticks-per-second             => uint,
      max-block-items              => uint,
      storage-hints                => StorageHints,
      opcodes                      => [+ uint],
      rr-types                     => [+ uint],
      ? storage-flags              => StorageFlags,
      ? client-address-prefix-ipv4 => uint,
      ? client-address-prefix-ipv6 => uint,
      ? server-address-prefix-ipv4 => uint,
      ? server-address-prefix-ipv6 => uint,
      ? sampling-method            => tstr,
      ? anonymisation-method       => tstr,
  }
  ticks-per-second           = 0
  max-block-items            = 1
  storage-hints              = 2
  opcodes                    = 3
  rr-types                   = 4
  storage-flags              = 5
  client-address-prefix-ipv4 = 6
  client-address-prefix-ipv6 = 7
  server-address-prefix-ipv4 = 8
  server-address-prefix-ipv6 = 9
  sampling-method            = 10
  anonymisation-method       = 11

    ; A hint indicates if the collection method will output the
    ; item or will ignore the item if present.
    StorageHints = {
        query-response-hints           => QueryResponseHints,
        query-response-signature-hints => QueryResponseSignatureHints,
        rr-hints                       => RRHints,
        other-data-hints               => OtherDataHints,
    }
    query-response-hints           = 0
    query-response-signature-hints = 1
    rr-hints                       = 2
    other-data-hints               = 3

      QueryResponseHintValues = &(
```

```
        time-offset                 : 0,
        client-address-index        : 1,
        client-port                 : 2,
        transaction-id              : 3,
        qr-signature-index          : 4,
        client-hoplimit             : 5,
        response-delay              : 6,
        query-name-index            : 7,
        query-size                  : 8,
        response-size               : 9,
        response-processing-data    : 10,
        query-question-sections     : 11,    ; Second & subsequent questions
        query-answer-sections       : 12,
        query-authority-sections    : 13,
        query-additional-sections   : 14,
        response-answer-sections    : 15,
        response-authority-sections : 16,
        response-additional-sections : 17,
    )
    QueryResponseHints = uint .bits QueryResponseHintValues

    QueryResponseSignatureHintValues = &(
        server-address     : 0,
        server-port        : 1,
        qr-transport-flags : 2,
        qr-type            : 3,
        qr-sig-flags       : 4,
        query-opcode       : 5,
        dns-flags          : 6,
        query-rcode        : 7,
        query-class-type   : 8,
        query-qdcount      : 9,
        query-ancount      : 10,
        query-arcount      : 11,
        query-nscount      : 12,
        query-edns-version : 13,
        query-udp-size     : 14,
        query-opt-rdata    : 15,
        response-rcode     : 16,
    )
    QueryResponseSignatureHints = uint .bits QueryResponseSignatureHintValues

    RRHintValues = &(
        ttl         : 0,
        rdata-index : 1,
    )
    RRHints = uint .bits RRHintValues
```

```
      OtherDataHintValues = &(
          malformed-messages   : 0,
          address-event-counts : 1,
      )
      OtherDataHints = uint .bits OtherDataHintValues

    StorageFlagValues = &(
        anonymised-data      : 0,
        sampled-data         : 1,
        normalised-names     : 2,
    )
    StorageFlags = uint .bits StorageFlagValues

  CollectionParameters = {
      ? query-timeout      => uint,
      ? skew-timeout       => uint,
      ? snaplen            => uint,
      ? promisc            => uint,
      ? interfaces         => [+ tstr],
      ? server-addresses   => [+ IPAddress], ; Hint for later analysis
      ? vlan-ids           => [+ uint],
      ? filter             => tstr,
      ? generator-id       => tstr,
      ? host-id            => tstr,
  }
  query-timeout      = 0
  skew-timeout       = 1
  snaplen            = 2
  promisc            = 3
  interfaces         = 4
  server-addresses   = 5
  vlan-ids           = 6
  filter             = 7
  generator-id       = 8
  host-id            = 9

;
; Data in the file is stored in Blocks.
;
Block = {
    block-preamble          => BlockPreamble,
    ? block-statistics      => BlockStatistics, ; Much of this could be derived
    ? block-tables          => BlockTables,
    ? query-responses       => [+ QueryResponse],
    ? address-event-counts  => [+ AddressEventCount],
    ? malformed-messages    => [+ MalformedMessage],
}
block-preamble       = 0
```

```
block-statistics      = 1
block-tables          = 2
query-responses       = 3
address-event-counts  = 4
malformed-messages    = 5

;
; The (mandatory) preamble to a block.
;
BlockPreamble = {
    ? earliest-time          => Timestamp,
    ? block-parameters-index => uint .default 0,
}
earliest-time          = 0
block-parameters-index = 1

; Ticks are subsecond intervals. The number of ticks in a second is file/block
; metadata. Signed and unsigned tick types are defined.
ticks = int
uticks = uint

Timestamp = [
    timestamp-secs   : uint,
    timestamp-uticks : uticks,
]

;
; Statistics about the block contents.
;
BlockStatistics = {
    ? processed-messages  => uint,
    ? qr-data-items       => uint,
    ? unmatched-queries   => uint,
    ? unmatched-responses => uint,
    ? discarded-opcode    => uint,
    ? malformed-items     => uint,
}
processed-messages  = 0
qr-data-items       = 1
unmatched-queries   = 2
unmatched-responses = 3
discarded-opcode    = 4
malformed-items     = 5

;
; Tables of common data referenced from records in a block.
;
BlockTables = {
```

```
    ? ip-address             => [+ IPAddress],
    ? classtype              => [+ ClassType],
    ? name-rdata             => [+ bstr],    ; Holds both Name RDATA and RDATA
    ? qr-sig                 => [+ QueryResponseSignature],
    ? QuestionTables,
    ? RRTables,
    ? malformed-message-data => [+ MalformedMessageData],
}
ip-address             = 0
classtype              = 1
name-rdata             = 2
qr-sig                 = 3
qlist                  = 4
qrr                    = 5
rrlist                 = 6
rr                     = 7
malformed-message-data = 8

IPv4Address = bstr .size 4
IPv6Address = bstr .size 16
IPAddress = IPv4Address / IPv6Address

ClassType = {
    type  => uint,
    class => uint,
}
type  = 0
class = 1

QueryResponseSignature = {
    ? server-address-index  => uint,
    ? server-port           => uint,
    ? qr-transport-flags    => QueryResponseTransportFlags,
    ? qr-type               => QueryResponseType,
    ? qr-sig-flags          => QueryResponseFlags,
    ? query-opcode          => uint,
    ? qr-dns-flags          => DNSFlags,
    ? query-rcode           => uint,
    ? query-classtype-index => uint,
    ? query-qd-count        => uint,
    ? query-an-count        => uint,
    ? query-ns-count        => uint,
    ? query-ar-count        => uint,
    ? edns-version          => uint,
    ? udp-buf-size          => uint,
    ? opt-rdata-index       => uint,
    ? response-rcode        => uint,
}
```

```
server-address-index  = 0
server-port           = 1
qr-transport-flags    = 2
qr-type               = 3
qr-sig-flags          = 4
query-opcode          = 5
qr-dns-flags          = 6
query-rcode           = 7
query-classtype-index = 8
query-qd-count        = 9
query-an-count        = 10
query-ns-count        = 12
query-ar-count        = 12
edns-version          = 13
udp-buf-size          = 14
opt-rdata-index       = 15
response-rcode        = 16

  ; Transport gives the values that may appear in bits 1..4 of
  ; TransportFlags. There is currently no way to express this in
  ; CDDL, so Transport is unused. To avoid confusion when used
  ; with CDDL tools, it is commented out.
  ;
  ; Transport = &(
  ;     udp               : 0,
  ;     tcp               : 1,
  ;     tls               : 2,
  ;     dtls              : 3,
  ; )

  TransportFlagValues = &(
      ip-version         : 0,     ; 0=IPv4, 1=IPv6
  ) / (1..4)
  TransportFlags = uint .bits TransportFlagValues

  QueryResponseTransportFlagValues = &(
      query-trailingdata : 5,
  ) / TransportFlagValues
  QueryResponseTransportFlags = uint .bits QueryResponseTransportFlagValues

  QueryResponseType = &(
      stub      : 0,
      client    : 1,
      resolver  : 2,
      auth      : 3,
      forwarder : 4,
      tool      : 5,
  )
```

```
  QueryResponseFlagValues = &(
      has-query               : 0,
      has-reponse             : 1,
      query-has-opt           : 2,
      response-has-opt        : 3,
      query-has-no-question   : 4,
      response-has-no-question: 5,
  )
  QueryResponseFlags = uint .bits QueryResponseFlagValues

  DNSFlagValues = &(
      query-cd   : 0,
      query-ad   : 1,
      query-z    : 2,
      query-ra   : 3,
      query-rd   : 4,
      query-tc   : 5,
      query-aa   : 6,
      query-do   : 7,
      response-cd: 8,
      response-ad: 9,
      response-z : 10,
      response-ra: 11,
      response-rd: 12,
      response-tc: 13,
      response-aa: 14,
  )
  DNSFlags = uint .bits DNSFlagValues

QuestionTables = (
    qlist => [+ QuestionList],
    qrr   => [+ Question]
)

  QuestionList = [+ uint]           ; Index of Question

  Question = {                      ; Second and subsequent questions
      name-index      => uint,      ; Index to a name in the name-rdata table
      classtype-index => uint,
  }
  name-index      = 0
  classtype-index = 1

RRTables = (
    rrlist => [+ RRList],
    rr     => [+ RR]
)
```

```
  RRList = [+ uint]                    ; Index of RR

  RR = {
      name-index      => uint,          ; Index to a name in the name-rdata
table
      classtype-index => uint,
      ? ttl           => uint,
      ? rdata-index   => uint,          ; Index to RDATA in the name-rdata
table
  }
  ; Other map key values already defined above.
  ttl         = 2
  rdata-index = 3

MalformedMessageData = {
    ? server-address-index   => uint,
    ? server-port            => uint,
    ? mm-transport-flags     => TransportFlags,
    ? mm-payload             => bstr,
}
; Other map key values already defined above.
mm-transport-flags      = 2
mm-payload              = 3

;
; A single query/response pair.
;
QueryResponse = {
    ? time-offset              => uticks,     ; Time offset from start of block
    ? client-address-index     => uint,
    ? client-port              => uint,
    ? transaction-id           => uint,
    ? qr-signature-index       => uint,
    ? client-hoplimit          => uint,
    ? response-delay           => ticks,
    ? query-name-index         => uint,
    ? query-size               => uint,       ; DNS size of query
    ? response-size            => uint,       ; DNS size of response
    ? response-processing-data => ResponseProcessingData,
    ? query-extended           => QueryResponseExtended,
    ? response-extended        => QueryResponseExtended,
}
time-offset             = 0
client-address-index    = 1
client-port             = 2
transaction-id          = 3
qr-signature-index      = 4
client-hoplimit         = 5
```

```
response-delay           = 6
query-name-index         = 7
```

```
query-size              = 8
response-size           = 9
response-processing-data = 10
query-extended          = 11
response-extended       = 12

ResponseProcessingData = {
    ? bailiwick-index  => uint,
    ? processing-flags => ResponseProcessingFlags,
}
bailiwick-index = 0
processing-flags = 1

  ResponseProcessingFlagValues = &(
      from-cache : 0,
  )
  ResponseProcessingFlags = uint .bits ResponseProcessingFlagValues

QueryResponseExtended = {
    ? question-index   => uint,       ; Index of QuestionList
    ? answer-index     => uint,       ; Index of RRList
    ? authority-index  => uint,
    ? additional-index => uint,
}
question-index   = 0
answer-index     = 1
authority-index  = 2
additional-index = 3

;
; Address event data.
;
AddressEventCount = {
    ae-type          => &AddressEventType,
    ? ae-code        => uint,
    ae-address-index => uint,
    ae-count         => uint,
}
ae-type          = 0
ae-code          = 1
ae-address-index = 2
ae-count         = 3

AddressEventType = (
    tcp-reset              : 0,
    icmp-time-exceeded     : 1,
    icmp-dest-unreachable  : 2,
    icmpv6-time-exceeded   : 3,
```

```
    icmpv6-dest-unreachable: 4,
    icmpv6-packet-too-big  : 5,
)

;
; Malformed messages.
;
MalformedMessage = {
    ? time-offset           => uticks,   ; Time offset from start of block
    ? client-address-index  => uint,
    ? client-port           => uint,
    ? message-data-index    => uint,
}
; Other map key values already defined above.
message-data-index = 3
```

Appendix B. DNS Name compression example

The basic algorithm, which follows the guidance in [RFC1035], is simply to collect each name, and the offset in the packet at which it starts, during packet construction. As each name is added, it is offered to each of the collected names in order of collection, starting from the first name. If labels at the end of the name can be replaced with a reference back to part (or all) of the earlier name, and if the uncompressed part of the name is shorter than any compression already found, the earlier name is noted as the compression target for the name.

The following tables illustrate the process. In an example packet, the first name is example.com.

N	Name	Uncompressed	Compression Target
1	example.com		

The next name added is bar.com. This is matched against example.com. The com part of this can be used as a compression target, with the remaining uncompressed part of the name being bar.

N	Name	Uncompressed	Compression Target
1	example.com		
2	bar.com	bar	1 + offset to com

The third name added is www.bar.com. This is first matched against example.com, and as before this is recorded as a compression target, with the remaining uncompressed part of the name being www.bar. It is then matched against the second name, which again can be a compression target. Because the remaining uncompressed part of the name is www, this is an improved compression, and so it is adopted.

N	Name	Uncompressed	Compression Target
1	example.com		
2	bar.com	bar	1 + offset to com
3	www.bar.com	www	2

As an optimization, if a name is already perfectly compressed (in other words, the uncompressed part of the name is empty), then no further names will be considered for compression.

B.1. NSD compression algorithm

Using the above basic algorithm the packet lengths of responses generated by NSD [6] can be matched almost exactly. At the time of writing, a tiny number (<.01%) of the reconstructed packets had incorrect lengths.

B.2. Knot Authoritative compression algorithm

The Knot Authoritative [7] name server uses different compression behavior, which is the result of internal optimization designed to balance runtime speed with compression size gains. In brief, and omitting complications, Knot Authoritative will only consider the QNAME and names in the immediately preceding RR section in an RRSET as compression targets.

A set of smart heuristics as described below can be implemented to mimic this and while not perfect it produces output nearly, but not quite, as good a match as with NSD. The heuristics are:

1. A match is only perfect if the name is completely compressed AND the TYPE of the section in which the name occurs matches the TYPE of the name used as the compression target.

2. If the name occurs in RDATA:

   * If the compression target name is in a query, then only the first RR in an RRSET can use that name as a compression target.

  * The compression target name MUST be in RDATA.

  * The name section TYPE must match the compression target name section TYPE.

  * The compression target name MUST be in the immediately preceding RR in the RRSET.

Using this algorithm less than 0.1% of the reconstructed packets had incorrect lengths.

B.3. Observed differences

In sample traffic collected on a root name server around 2-4% of responses generated by Knot had different packet lengths to those produced by NSD.

Appendix C. Comparison of Binary Formats

Several binary serialisation formats were considered, and for completeness were also compared to JSON.

- Apache Avro [8]. Data is stored according to a pre-defined schema. The schema itself is always included in the data file. Data can therefore be stored untagged, for a smaller serialisation size, and be written and read by an Avro library.

  * At the time of writing, Avro libraries are available for C, C++, C#, Java, Python, Ruby and PHP. Optionally tools are available for C++, Java and C# to generate code for encoding and decoding.

- Google Protocol Buffers [9]. Data is stored according to a pre-defined schema. The schema is used by a generator to generate code for encoding and decoding the data. Data can therefore be stored untagged, for a smaller serialisation size. The schema is not stored with the data, so unlike Avro cannot be read with a generic library.

  * Code must be generated for a particular data schema to to read and write data using that schema. At the time of writing, the Google code generator can currently generate code for encoding and decoding a schema for C++, Go, Java, Python, Ruby, C#, Objective-C, Javascript and PHP.

- CBOR [10]. Defined in [RFC7049], this serialisation format is comparable to JSON but with a binary representation. It does not use a pre-defined schema, so data is always stored tagged.

However, CBOR data schemas can be described using CDDL [I-D.ietf-cbor-cddl] and tools exist to verify data files conform to the schema.

* CBOR is a simple format, and simple to implement. At the time of writing, the CBOR website lists implementations for 16 languages.

Avro and Protocol Buffers both allow storage of untagged data, but because they rely on the data schema for this, their implementation is considerably more complex than CBOR. Using Avro or Protocol Buffers in an unsupported environment would require notably greater development effort compared to CBOR.

A test program was written which reads input from a PCAP file and writes output using one of two basic structures; either a simple structure, where each query/response pair is represented in a single record entry, or the C-DNS block structure.

The resulting output files were then compressed using a variety of common general-purpose lossless compression tools to explore the compressibility of the formats. The compression tools employed were:

o snzip [11]. A command line compression tool based on the Google Snappy [12] library.

o lz4 [13]. The command line compression tool from the reference C LZ4 implementation.

o gzip [14]. The ubiquitous GNU zip tool.

o zstd [15]. Compression using the Zstandard algorithm.

o xz [16]. A popular compression tool noted for high compression.

In all cases the compression tools were run using their default settings.

Note that this draft does not mandate the use of compression, nor any particular compression scheme, but it anticipates that in practice output data will be subject to general-purpose compression, and so this should be taken into consideration.

"test.pcap", a 662Mb capture of sample data from a root instance was used for the comparison. The following table shows the formatted size and size after compression (abbreviated to Comp. in the table headers), together with the task resident set size (RSS) and the user

time taken by the compression. File sizes are in Mb, RSS in kb and user time in seconds.

Format	File size	Comp.	Comp. size	RSS	User time
PCAP	661.87	snzip	212.48	2696	1.26
		lz4	181.58	6336	1.35
		gzip	153.46	1428	18.20
		zstd	87.07	3544	4.27
		xz	49.09	97416	160.79
JSON simple	4113.92	snzip	603.78	2656	5.72
		lz4	386.42	5636	5.25
		gzip	271.11	1492	73.00
		zstd	133.43	3284	8.68
		xz	51.98	97412	600.74
Avro simple	640.45	snzip	148.98	2656	0.90
		lz4	111.92	5828	0.99
		gzip	103.07	1540	11.52
		zstd	49.08	3524	2.50
		xz	22.87	97308	90.34
CBOR simple	764.82	snzip	164.57	2664	1.11
		lz4	120.98	5892	1.13
		gzip	110.61	1428	12.88
		zstd	54.14	3224	2.77
		xz	23.43	97276	111.48
PBuf simple	749.51	snzip	167.16	2660	1.08
		lz4	123.09	5824	1.14
		gzip	112.05	1424	12.75
		zstd	53.39	3388	2.76
		xz	23.99	97348	106.47
JSON block	519.77	snzip	106.12	2812	0.93
		lz4	104.34	6080	0.97
		gzip	57.97	1604	12.70
		zstd	61.51	3396	3.45
		xz	27.67	97524	169.10
Avro block	60.45	snzip	48.38	2688	0.20
		lz4	48.78	8540	0.22
		gzip	39.62	1576	2.92
		zstd	29.63	3612	1.25
		xz	18.28	97564	25.81

CBOR block	75.25	snzip	53.27	2684	0.24
		lz4	51.88	8008	0.28
		gzip	41.17	1548	4.36
		zstd	30.61	3476	1.48
		xz	18.15	97556	38.78
PBuf block	67.98	snzip	51.10	2636	0.24
		lz4	52.39	8304	0.24
		gzip	40.19	1520	3.63
		zstd	31.61	3576	1.40
		xz	17.94	97440	33.99

The above results are discussed in the following sections.

C.1. Comparison with full PCAP files

An important first consideration is whether moving away from PCAP offers significant benefits.

The simple binary formats are typically larger than PCAP, even though they omit some information such as Ethernet MAC addresses. But not only do they require less CPU to compress than PCAP, the resulting compressed files are smaller than compressed PCAP.

C.2. Simple versus block coding

The intention of the block coding is to perform data de-duplication on query/response records within the block. The simple and block formats above store exactly the same information for each query/response record. This information is parsed from the DNS traffic in the input PCAP file, and in all cases each field has an identifier and the field data is typed.

The data de-duplication on the block formats show an order of magnitude reduction in the size of the format file size against the simple formats. As would be expected, the compression tools are able to find and exploit a lot of this duplication, but as the de-duplication process uses knowledge of DNS traffic, it is able to retain a size advantage. This advantage reduces as stronger compression is applied, as again would be expected, but even with the strongest compression applied the block formatted data remains around 75% of the size of the simple format and its compression requires roughly a third of the CPU time.

C.3. Binary versus text formats

Text data formats offer many advantages over binary formats, particularly in the areas of ad-hoc data inspection and extraction. It was therefore felt worthwhile to carry out a direct comparison, implementing JSON versions of the simple and block formats.

Concentrating on JSON block format, the format files produced are a significant fraction of an order of magnitude larger than binary formats. The impact on file size after compression is as might be expected from that starting point; the stronger compression produces files that are 150% of the size of similarly compressed binary format, and require over 4x more CPU to compress.

C.4. Performance

Concentrating again on the block formats, all three produce format files that are close to an order of magnitude smaller that the original "test.pcap" file. CBOR produces the largest files and Avro the smallest, 20% smaller than CBOR.

However, once compression is taken into account, the size difference narrows. At medium compression (with gzip), the size difference is 4%. Using strong compression (with xz) the difference reduces to 2%, with Avro the largest and Protocol Buffers the smallest, although CBOR and Protocol Buffers require slightly more compression CPU.

The measurements presented above do not include data on the CPU required to generate the format files. Measurements indicate that writing Avro requires 10% more CPU than CBOR or Protocol Buffers. It appears, therefore, that Avro's advantage in compression CPU usage is probably offset by a larger CPU requirement in writing Avro.

C.5. Conclusions

The above assessments lead us to the choice of a binary format file using blocking.

As noted previously, this draft anticipates that output data will be subject to compression. There is no compelling case for one particular binary serialisation format in terms of either final file size or machine resources consumed, so the choice must be largely based on other factors. CBOR was therefore chosen as the binary serialisation format for the reasons listed in Section 5.

C.6. Block size choice

Given the choice of a CBOR format using blocking, the question arises of what an appropriate default value for the maximum number of query/response pairs in a block should be. This has two components; what is the impact on performance of using different block sizes in the format file, and what is the impact on the size of the format file before and after compression.

The following table addresses the performance question, showing the impact on the performance of a C++ program converting "test.pcap" to C-DNS. File size is in Mb, resident set size (RSS) in kb.

Block size	File size	RSS	User time
1000	133.46	612.27	15.25
5000	89.85	676.82	14.99
10000	76.87	752.40	14.53
20000	67.86	750.75	14.49
40000	61.88	736.30	14.29
80000	58.08	694.16	14.28
160000	55.94	733.84	14.44
320000	54.41	799.20	13.97

Increasing block size, therefore, tends to increase maximum RSS a little, with no significant effect (if anything a small reduction) on CPU consumption.

The following figure plots the effect of increasing block size on output file size for different compressions.

Figure showing effect of block size on file size (PNG) [17]

Figure showing effect of block size on file size (SVG) [18]

From the above, there is obviously scope for tuning the default block size to the compression being employed, traffic characteristics, frequency of output file rollover etc. Using a strong compression, block sizes over 10,000 query/response pairs would seem to offer limited improvements.

Authors' Addresses

John Dickinson
Sinodun IT
Magdalen Centre
Oxford Science Park
Oxford OX4 4GA
United Kingdom

Email: jad@sinodun.com

Jim Hague
Sinodun IT
Magdalen Centre
Oxford Science Park
Oxford OX4 4GA
United Kingdom

Email: jim@sinodun.com

Sara Dickinson
Sinodun IT
Magdalen Centre
Oxford Science Park
Oxford OX4 4GA
United Kingdom

Email: sara@sinodun.com

Terry Manderson
ICANN
12025 Waterfront Drive
Suite 300
Los Angeles CA 90094-2536

Email: terry.manderson@icann.org

John Bond
ICANN
12025 Waterfront Drive
Suite 300
Los Angeles CA 90094-2536

Email: john.bond@icann.org