1. The designed channel works in steps below:

1.1. Data Encapsulation

DNS works on UDP as a connectionless protocol with no guarantee for packet delivery. Packets need to be traced in case of packet lost. Therefore, we benefit of TCP principles in data encapsulation. The sequence numbers are used to track packets and prevent of retransmissions. In contrast to TCP, only one sequence number is used in both sides to save more space for data and make the channel simple. The structure is shown in Fig. 1.

Flags	**ID**	**Sequence number**	**Message**	**HMAC**

Fig. 1. *The packet structure in the proposed channel*

Flags: they are used for control purposes. Three bits are dedicated to flags, one bit per each flag. Flags are defined as below:

- Data_type: Both of communication parties need to know the type of data in order to decide how to behave with it. The data type is distinguished into text and binary. Before data transmission, its type is checked and the Data_type flag will be set. The 0 value is for the binary data and 1 for the text data.
- False integrity: it checks the integrity of data by checking the HMAC part. When the HMAC code of a packet is incorrect, the receiver of packet sets this flag to 1 and sends another packet. The sequence number in this packet is supposed to be the same number as corrupted packet. When the sender receives such packet, it will notice there was a problem in the packet and resend it.
- Fin: each side sets this flag to 1 while they have no data to send.

ID: a one byte random number as the client identification. Each client will get a unique ID.

Sequence Number: a two byte random number to represent the packets sequence. The client selects the initial sequence number (ISN) and the number is incremented one by one for each packet.

Message: the data needs to be sent. It can be a file, string, etc.

HMAC: the HMAC code is calculated to check the integrity of data and then attached to the end of the packet.

1.2. Candidate Record Type Selection

The covert channel is supposed to work in a network, since this environment is constantly changing, the network condition needs to be analysed to adapt the channel activity with its working network. Thus, before the channel starts working, a real time process checks the network and DNS record types with more frequency in the current traffic will be selected. These record types are considered as candidate records and will be stored. Whenever the channel has data to transfer, one candidate record type will be selected randomly. This approach makes the channel obscure and adaptable with its network.

1.3. Encoding Type

In order to prevent raise of attention to the channel, queries need to look normal like standard DNS packets. Therefore, Base32 is used to encode queries from the client. Base32 encoding is designed to represent arbitrary sequences of octets in a form to be case insensitive, but not human readable. Base32 or 5-bit encoding is commonly used for requests from the client [1] and also references such as DNScurve [2] and RFC5155 use Base32 encoding in queries for security purposes. Base32 utilization as an encoding method in one hand leads to lower capacity (5 bits per character), but on the other hand it makes the channel more adapted with normal traffics. DNS queries and responses pass through the ISP’s DNS infrastructure, they must not be too different with normal DNS packets. The responses in record types such as CNAME which have a similar response structure with requests also encode in Base32. In order to encode more data, Base64 or 6-bit encoding is used for TXT record responses.

1.4. Variable Packet Length

Some detection methods consider DNS queries and response length, which look at all hostname requests longer than 52 characters. DNS covert channels usually try to embed as much as possible data in requests and response, therefore, they will have long labels up to 63 characters and long overall names up to 255 characters [1].

Based on RFC1034, the hostname in DNS query is limited to 255 characters and it is structured in labels. Labels must be 63 characters or less. In this implementation, test.com is considered as the channel domain. For this domain, the hostname with maximum amount of data would be look like this:

63chars.63chars.63chars.54chars.test.com

In encoding with Base32, The maximum amount of data would be 1215 bits (151 bytes) per packet, definitely in Base64 this amount will be more. In order to provide more reliability and prevent of recognition by length based detection methods, we don't embed data to the maximum amount. After encoding data in Base32, the volume of data decreases because of 5 bits per character, in order to not make large packets length and payload, a range with a maximum of 10 and a minimum of 2 bytes is considered. Whenever a query needs to be sent, a random number in range (2, 10) will be selected as the total amount of message, also the labels length will be changed randomly. This range can be modified based on the statistics in network evaluation phase as mentioned in section 1.2. It means, if most of the normal packets in the network are large, bigger numbers will be selected for the range, otherwise the range will have small numbers like this current implementation.

This method works for upstream data (client to server) which the data is always encapsulated in the hostnames. It can also work for downstream data (server to client) with record types such as CNAME, which have a similar response structure with requests. For other record types in downstream like TXT, we can only change the total amount of encapsulated data. The header length of encapsulated data is invariable for all packets, but with altering the amount of message, the total length of packets will change and this approach prevents of having uniformed packets, which can be suspicious.

1.5. A Lightweight Obfuscation Method to Obscure the Data Stream

The data in channel needs to get defaced to prevent of leaking information even after the channel gets discovered. However, Iodine [3] transfers data in plain text and Feederbot just uses a simple encryption method like RC4 [4], Dnscat2 benefits from some stronger encryption methods like salsa20[5].

In the proposed channel, we just need to deface the data without imposing more overhead on the channel. Therefore the data get obfuscated not encrypted, which needs key exchange. It is assumed there is a pre-shared secret between communication parties. A pseudorandom number generator (PRNG) is used. PRNG seed is generated based on (1):

$$seed = Preshared\_secret \oplus ISN, \\ R_i = random(seed) \qquad (1)$$

Where ISN is the initial sequence number has exchanged in the beginning of the connection establishment. ISN number is supposed to be unique for each client. Whenever a packet needs to be sent, Ri will be generated and i is the number of the generated packet. The data is get obfuscated based on (2).

$$Obfuscate\_data = data \oplus R_i \qquad (2)$$

After obfuscation, the data will be encoded in Base32 or Base64 and encapsulate in the packet.

1.6. Integrity Check

Feederbot uses a simple CRC32 code to check the data integrity and Dnscat2 signs the data. In the proposed channel, data integrity is provided by HMAC code.

Hence we want to use HMAC only for integrity check and we rely on obfuscation for security purposes. MD5 is used as the hash function for HMAC. MD5 has the shorter output with 128 bits in comparing with SHA family, and shorter output gives us the opportunity to transfer more data per packet. The HMAC key generation is stated in (3).

$$Key = MD5(Preshared\_secret||ISN) \qquad (3)$$

And the HMAC function works like (4):

$$HMAC = MD5(data||ISN||Preshared\_secret) \quad (4)$$

The HMAC code is added after the encoded data. The data in HMAC part is the message before obfuscation and encoding, it doesn't include header which has flags, ID and sequence number. The header needs to transfer in plain text for control purposes. When a packet is received, first the data is decoded and deobfuscated, then the message is obtained. The other party has to generate the HMAC code and compares it with the received code, if they are equal it means the data has transferred properly. Otherwise the false integrity flag will be set to 1 and a packet will be sent with the same sequence number and an arbitrary string in the message section. When the sender of packet receives a packet with the same sequence number as its previous packet has just sent, it will notice there was a problem in that packet and resend it.

1.7. Packet Loss and Reordering

For being undetectable, DNS queries in the proposed channel pass through the pre-configured DNS resolver on the host and unlike Feederbot, the channel does not query the server directly. Bypassing the pre-configured DNS resolver can be suspicious. Therefore the pre-configured DNS server on the host is responsible for receiving the replies and our DNS queries will behaved like other queries in the host. In the server side, since there is one ID per client, until the server doesn't receive a DNS query from the client with Fin flag 1, receiving packets will continue, and the server keeps the connection open. The client can encounter with a problem and the server doesn't receive any query from the client for a long time, the server needs to terminate the connection one sided.

For this approach, a time interval is set based on normal TTL values for DNS records. If this interval is expired and the server had not received any packet from the client, the server terminates the connection and the client ID will be removed from the server side. If the client connects the server after this time expiration, the server considers this connection as a new one.

As sequence numbers are utilized in the channel, both sides have to discard packets which have an equal or smaller sequence number with the last packet they received, for server this is done per each ID. There is an exception has explained in section 1.6 that packets with the same sequence number won't be discarded.

1.8. The Communication Phases

This section is about the phases in which client and server as two sides of covert channel start a communication, transfer the data and terminate the connection after they finish sending data.

2.8.1 Connection Establishment: The proposed channel works on DNS, the client needs to start the communication and the server responses to the client request. First the client generates one byte random number as its ID and a two bytes random number as initial sequence number (ISN). Encapsulated data will be structured as explained in section 1.1.

2.8.2 Data Transmission from Client Side: Data transfer from the client side will be done in the following steps:

1. Before sending data, its data type will be checked and the Data_type flag will be set to an appropriate value.
2. The amount of message which needs to transfer is selected randomly in the given range as described in section 1.4
3. The message gets obfuscated and encoded with the header part based on the method discussed in section 1.5 and the HMAC code is added as explained in section 1.6.
4. The record type is selected randomly among of candidate record types as described in section 1.2.
5. Data from the client is always transferred as a request, therefore we need to embed the data in the hostname for example: data_from_client.test.com.

If there is no data left for sending but the server still have data, the client sets its Fin flag to 1 and it indicates the client finished sending data. Then a random string is generated and encapsulated in the message part. The random string with flags, ID and sequence number follow steps like data packets. Then, it will be encapsulated in the hostname of the query. The proposed scheme is shown in Fig. 2.

2.8.3 Data Transmission from Server Side: When a request in defined domain (for example: test.com) receives, first the server decodes and deobfuscates the data as (5), (6):

$$Obfuscated\_data = Base32.decode(Recieved\_data)(5)$$

$$R_i = random(seed), data = Obfuscated\_data \oplus R_i \text{ (6)}$$

If the server has data to send, it follows the steps 1 to 3 as described above in section 1.8.2. Data from the server is embedded in rdata part of the response. In case of no data to send, since response to a DNS request is compulsory, whether there is data in the client side or not, the value of Fin flag is set to 1 to demonstrate the server has no data to send. Then a random string is generated and placed in the message part. The random string with flags and ID and sequence number follow the steps like data packets, then encapsulated in rdata part of the response and will be sent. The proposed scheme is shown in Fig. 3. Steps in Fig.3 are the same ones in Fig. 2.

2.8.4 Connection Termination: When each part of the communication finish sending data, they set the fin flag value to 1, besides a random string will be located in the message part and encapsulate with flags, ID and sequence number in a packet. Data receiving will continue until completion of data transfer by the other side, which will be determined by setting the fin flag to 1. When one side is about to terminate the connection, it needs to check whether the other side has declared finishing data or not. While the fin flag has set to 1 in both sides, each of parties can terminate the connection.

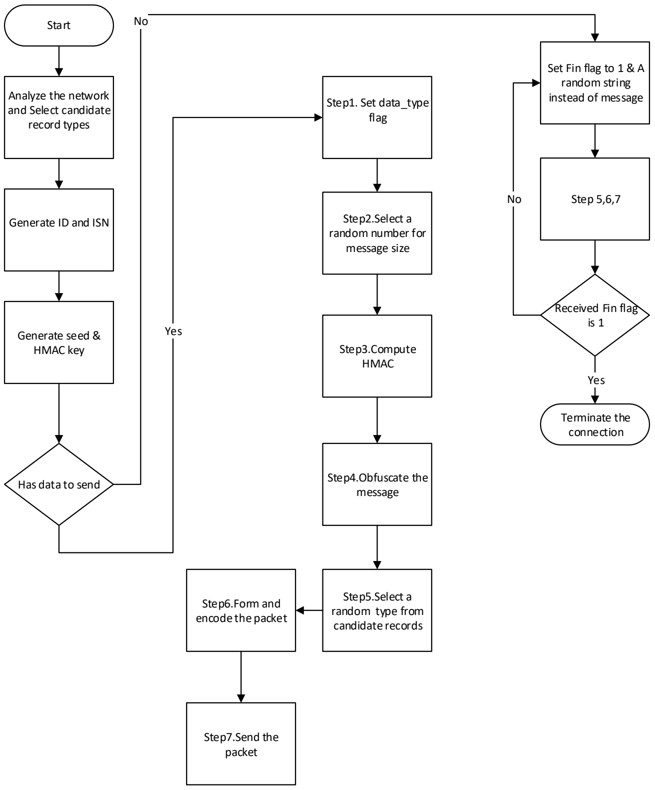
Start
Analyze the network and Select candidate record types
Generate ID and ISN
Generate seed & HMAC key
Has data to send
No
Yes
Step1. Set data_type flag
Step2.Select a random number for message size
Step3.Compute HMAC
Step4.Obfuscate the message
Step5.Select a random type from candidate records
Step6.Form and encode the packet
Step7.Send the packet
Set Fin flag to 1 & A random string instead of message
Step 5,6,7
Received Fin flag is 1
No
Yes
Terminate the connection

Fig. 2. *Data transmission from the client side*

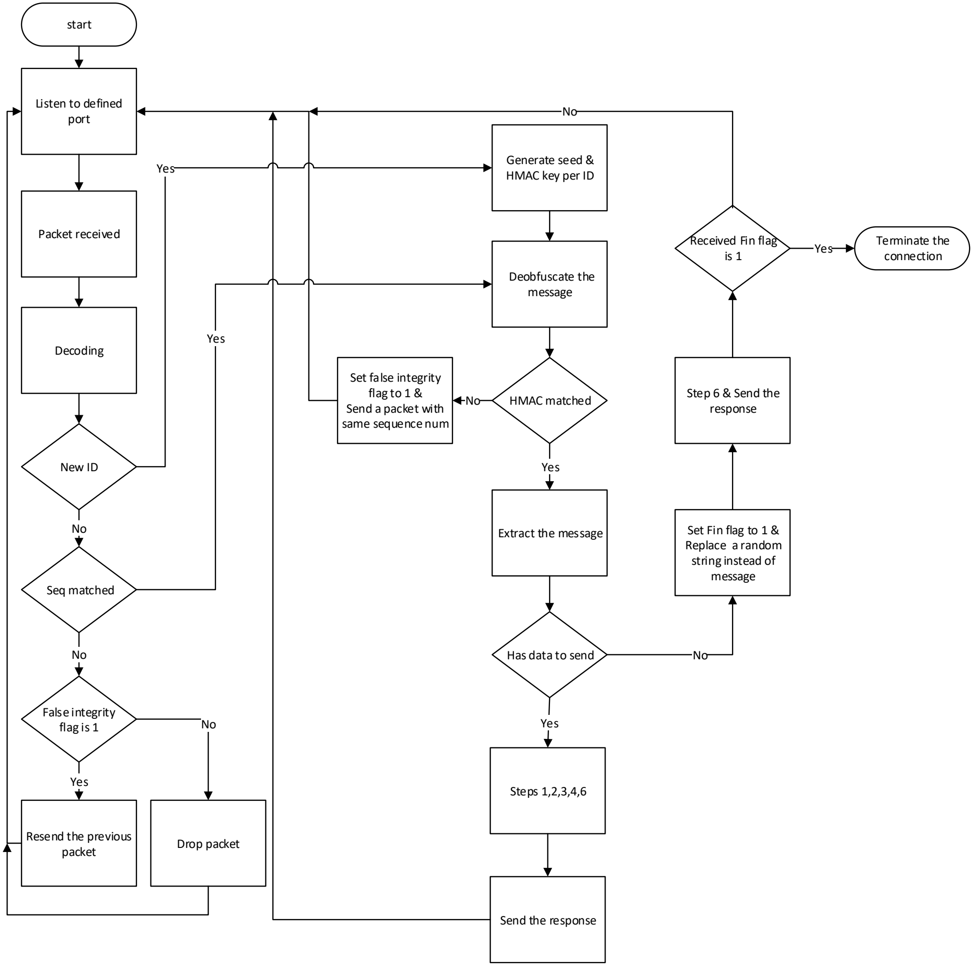

Fig. 3. *Data transmission from the server side*

2. Evaluation and results

The proposed channel is implemented in python programing language. The candidate record types are A, CNAME and TXT in this implementation. In order to check its adaptability with normal traffics, the proposed channel statistics were compared with two datasets of normal network traffic.

Dataset A: a normal capture of traffic in a home network. Capture duration is 1.9 hours and the number of packets is 5966. The performed actions are included: - Music streaming from 20songstogo.com - Gmail - Twiter - Jitsi chat connected to gtalk, SIP and CVUT jabber. - Some ssh - cacti web - normal webs with chrome. The capture has done in 2015-03-24.

Dataset B: a normal capture in a Linux Debian notebook computer in an xDSL network. Capture duration is 1281.885508 seconds (21 minutes). Applications and actions in the normal computer are: Deluge P2P on linux notebook. Downloading some large files. The p2p was running for 1 hour when the capture was started. At the beginning there is also an mtr sending icmp packages to www.google.com .Some web pages were accessed: twitter, facebook, etc. All using the Chrome browser. The capture has done in 2013-12-17.

Files with different random sizes were used as the message, then they were transferred by proposed channel. Tests had been done in a local network with two virtual machines as client and server. The operating system is Kali-Linux-2016.1 in the server and Ubuntu-14.04.2 in the client.

2.1. Length and Size

Table 1 provides the statistics for Datasets A and B that gained by Wireshark. The proposed channel was evaluated in payload size, qname length and packet length with Datasets A, B as normal DNS traffics to check the similarity with normal traffics.

Tables 2 and 3 show the statistics for 12 times running proposed channel with different file sizes as exchanged data. Results show, the minimum and maximum values for payload size, qname and packet length are variable as defined in section 1.4. The Average value of payload size in the proposed channel is 68.28. It is noticeable that values of proposed channel are compatible with average values in Dataset A with 71.65 and Dataset B with 70.62. The Average value of packet length in the proposed channel is 110.28. This results prove that our channel is adaptable with Datasets A and B in the similar circumstance. Average qname length is 37.71 in the proposed channel. This value is 19.94 and 27.48 for Dataset A and Dataset B respectively. However, still there is a difference for qname lengths with normal traffics. Base32 is used in encoding records, which cause longer lengths, but this doesn't make the queries abnormal. Also the maximum qname length in our proposed method is 46 and 47, and the maximum values for Datasets as normal DNS traffic are 44 and 36. Therefore, the proposed channel qname length is adaptable with maximum values of normal traffics and this difference won't be noticeable in huge amount of DNS traffics which are transferring daily.

Table 1 Statistics for datasets A and B

Dataset	Packet count	Payload size			Qname len			Packet len		
		min	max	avg	min	max	avg	min	max	avg
A	5966	22	377	71.65	6	44	19.94	64	419	113.56
B	2147	27	262	70.62	11	36	27.48	69	304	112.62

Table 2 The statistics for proposed channel and Dnscat2 in transferring 12 different data sizes

Total transferred bytes	Payload size			Qname length		
	min	max	avg	min	max	avg
12	53	95	67.17	37	38	37.33
19	53	107	72.63	37	46	39.75
23	53	105	70.20	37	40	37.80
30	50	102	67.83	34	37	36
41	48	107	69	32	39	37.38
52	46	107	69.33	30	46	37.11
69	50	103	68.50	34	47	38.07
76	44	97	66.36	28	40	36.36
87	46	110	67.56	30	46	37.06
93	46	96	64.84	30	46	36.94
100	50	103	69.80	34	46	40.60
112	46	103	66.11	30	46	38.11

Table 3 Packet length for proposed channel and Dnscat2 in transferring 12 different data sizes

Total transferred bytes	min	max	avg
12	95	137	109.17
19	95	149	114.63
23	95	147	112.20
30	92	144	109.83
41	90	149	111.00
52	88	149	111.33
69	92	145	110.50
76	86	139	108.36
87	88	152	109.56
93	88	138	106.84
100	92	145	111.80
112	88	145	108.11

2.2. *Standard Looking Queries*

As Table 4 shows most of the queries in normal traffics fall into 4th level or 3rd level. In the proposed channel all of the queries are in 4th level. This result indicates that queries in the proposed channel do not differ with normal queries and follow the normal standards.

Table 4 Label statistics for queries in datasets A and B

Dataset	4th Level or More	3rd Level	2nd Level	1st Level
A	253	2575	197	4
B	1007	60	7	1

2.3. *Security Consideration*

HMAC code is applied to check the integrity of data, so data corruption would be detectable in exchange. The channel can meet data integrity besides error detection by HMAC code utilization. The data obfuscation is implied not only to deface the data, but also to make it unreadable by an intruder, which satisfies confidentiality properties. In order to not impose a more overhead on the channel, any kind of encryption and authentication schemes are not used in the proposed scheme. Instead, a pre-shared value is used as the secret parameter to build a shared key between the communication parties.

2.4. *Channel Capacity*

Table 5 displays the amount of data as the number of packets which are needed to complete data transfer. In order to meet adaptability and prevent of making abnormal traffic, the encapsulated data in a single packet is not high, however this option can be changeable due to channel environment as explained in section 1.4. For this implementation and with these current parameters, the average capacity of channel would be 2.65 bytes of data per packet. The only overhead in data encapsulation is because of double issues. First, the header needs to repeat in every packet and the second is the encoding type as discussed in section 1.3.

Previous channels, keep their session alive with constantly pulling the server, in result they inject many DNS packets in network traffic. This situation can make anomaly and overhead in the network even for transferring a low amount of data. In contrast, the client in the proposed channel doesn’t have to pull the server for keeping the connection alive, because the channel is implied for data transmission and the connection will be terminated after it.

The proposed channel capacity is acceptable since there is no extra packets as pulling ones and the packets just transfer data without imposing overhead on the network. Furthermore, some confidential data such as secret keys are not big, they can be transferred with the proposed channel that provides them obscurity while exchange.

Table 5 Channel capacity

Total transferred bytes	Packet count	Bytes per packet
12	6	2
19	8	2.37
23	10	2.3
30	12	2.5
41	16	2.56
52	18	2.88
69	28	2.46
76	28	2.71
87	32	2.71
93	32	2.90
100	30	3.33
112	36	3.1

2.5. Detectability

For evaluating channel detectability, the proposed channel was tested with Opnsense 17.1 as an open source SPI firewall and Suricata 3.2.1 as an open source IDS. Suricata rules have been updated to the date of 2017-09-28. Two test scenarios were designed, one with Opnsense and another for Suricata.

As the client and server are running in virtual machines, Opnsense firewall was installed on another virtual machine and set as a gateway between the client and the server. All traffics from the client to the server and reverse pass through the firewall. The firewall was configured to monitor both incoming and outgoing traffics. After running Opnsense, the channel started running and transferring files. The firewall didn’t record anything suspicious about the channel.

In the next scenario, the channel detectability had been tested by an IDS and in our case Suricata was used. Since in DNS covert channels scenarios is assumed the server is under control and the client is the machine which is not in control, so the IDS was installed on the client side to check channel detectability by IDS updated rules. Similar to the previous scenario, after running Suricata, the channel started working and transferring files. To make an accurate estimate, the tests were repeated for varied data sizes as files. The IDS didn’t record any suspicious activity.

2.6. RFC Compliant

In this section, the proposed channel is evaluated with the following items to show the channel compatibility with standards in RFCs.

2.6.1 Queries and Responses Size: The proposed covert channel doesn’t proceed the limited size as mentioned in RFCs. While most of covert channels try to put as much as data in the packets, the proposed channel tries to make a good balance between undetectability and capacity.

2.6.2 Common Record Types: There is an evaluation phase before the channel starts working and this leads to selecting common records in current traffic. This approach prevents of selecting uncommon records that makes the channel detectable.

2.6.3 Encoding Type: The encoding methods were selected for encoding queries and responses are RFC compliant and commonly used.

3. Conclusion

We proposed a covert channel based on DNS protocol for covert data transmission. The channel can get adaptable with its network environment and works stealthy. The channel works with common DNS record types, encoding, and it is RFC compliant. A lightweight obfuscation method has implied for data encapsulation without imposing overhead on the channel. Data integrity in the channel has been provided by HMAC. Experimental results show the channel has a good compatibility with normal traffics. The evaluation with two security software, Opnsesne as an SPI firewall and Suricata as an open source IDS with latest updated rules, didn’t record anything suspicious. The channel capacity is an average 2.65 bytes of data per packet, which is acceptable because of the encoding type and undetectablility. We can also improve the capacity by adding another phase to evaluate the packet sizes of the network and set the channel parameters based on them. This option can lead to a higher capacity without drawing attention. The proposed channel has a good capacity for transferring confidential data such as key exchange, user credentials and other secret information.

4. References

[1] 13Farnham, G.: “Detecting DNS Tunneling” (SANS Institute, 2013)

[2] 21Andersson, B.: “Iodine by kryo,” http://code.kryo.se/iodine/, accessed December 2017

[3] 22Bowes, R.: “Dnscat2,” https://github.com/iagox86/dnscat2/blob/master/doc/protocol.md, accessed December 2017

[4] 23Dietrich, C.J., Rossow, C., Freiling, F.C., *et al*.: “On Botnets That Use DNS for Command and Control”*2011 Seventh European Conference on Computer Network Defense*, 2011.

[5] Josefsson, S.: “The Base16, Base32, and Base64 Data Encodings,” https://tools.ietf.org/html/rfc4648, accessed December 2017

[6] 25“DNSCurve: Usable security for DNS,” https://dnscurve.org/, accessed December 2017