

FTE_xfil: DNS Tunneling via Format Transforming Encryption and Huffman Coding

BENCE KANYOK
201550717

Supervision by Professor Lars Ramkilde Knudsen

Bachelor Project in Software Engineering

June 2026



The Mærsk Mc-Kinney Møller Institute
University of Southern Denmark

Abstract

This paper presents FTExfil, a proof of concept DNS tunneling tool for file exfiltration using Format Transforming Encryption (FTE) and character frequency shaping with a novel use of Huffman coding. The goal is to reduce payload level indicators commonly used in DNS tunneling detection, especially high hostname entropy and recognizable Base32/Base64 encodings. The tool encrypts and authenticates file chunks, encodes them into DNS query names and reconstructs the original file on an authoritative server. Evaluation on a 1 MB test file shows that the regular expression based FTE encoding achieved higher throughput and fewer DNS packets, with measured hostname entropy of approximately 4.674 bits per character. The weighted Huffman based encoder reduced entropy to approximately 4.180 bits per character, but required more packets and longer transfer times. The results show that entropy and character frequency shaping can reduce some payload level detection indicators, while increasing transmission overhead and leaving traffic level indicators such as query volume unresolved.

Acknowledgement

I would like to thank Professor Lars Ramkilde Knudsen for taking on the role of supervisor for this project and providing guidance.

Disclaimer

This paper is intended solely for academic research purposes. The work presented examines DNS tunneling techniques in order to understand their operation, limitations and potential security implications. The author does not condone the use of DNS tunneling for unauthorized access, data exfiltration or any unlawful activity.

During the project, large language models have been used for AI assisted development of the proof of concept application. As well as, to point out grammatical mistakes by suggesting corrections and/or paraphrasing sentences in this report.

Contents

Contents	ii
1 Introduction	1
2 Background	2
2.1 The DNS protocol	2
2.2 DNS Tunneling	2
2.3 Detection Methods	4
3 Methodology and Requirements	6
3.1 Software Engineering Methodology	6
3.2 Requirements	6
4 Analysis and Design	8
4.1 Encoding	8
4.2 Key Exchange	16
4.3 Tunneling	17
5 Implementation	21
5.1 Implementation Overview	21
5.2 Project Structure	21
5.3 Command Line Interface	21
5.4 Key Exchange	22
5.5 Encryption and Encoding	23
5.6 DNS Tunneling	25
6 Evaluation	27
6.1 Benchmarking Setup	27
6.2 Metrics	27
6.3 Results	27
6.4 Evaluation Summary	28
7 Conclusion	30
A Appendix	31
Bibliography	32

1 Introduction

Throughout history, covert communication channels have been utilised by a wide range of actors looking for secrecy. State sponsored intelligence agencies conducting espionage, dissidents and human rights activists for evasion of censorship and whistleblowers leaking internal documents to journalists. These channels are designed to conceal not only the content of the message (via encryption for example) but the fact that message exchange is taking place at all.

An example could be Karl Muller, a German spy in Britain during World War I who used lemon juice as invisible ink to hide military secrets on post letters¹. In the digital age, steganography aims to hide arbitrary information in images with tools like `steghide`², audio via `wavsteg`³, both of which modifies the Least Significant Bits in these files resulting in changes that are unnoticeable to humans, but can be decoded via the same tools. Many malware uses obfuscated network traffic to perform C2 (Command and Control) and exfiltrate data. Bits can be hidden in packet flags or in the timing differences between sent packages. The options seem endless, though with every new tool and technique released, detection methods evolve and malicious network packets are often disallowed by firewall rules or caught by Intrusion Detection Systems.

A notable and widely used method is performed using the Domain Name System (DNS). An example of intended usage is to query the IP addresses of known websites such as `www.google.com`. Due to the protocol's distributed nature it allows users to freely send information to trusted domain name servers, which then forwards it to a user controlled one. This tactic is called DNS tunneling, as it creates a tunnel between a client computer and a host server through DNS network packets.

In this project I propose a custom DNS tunneling software that focuses on evasion of entropy and regular expression based detection methods using Format Transforming Encryption and a novel use of Huffman coding.

¹<https://www.nationalarchives.gov.uk/explore-the-collection/stories/karl-muller-and-the-fatal-lemon/>

²<https://github.com/StegHigh/steghide>

³<https://github.com/ragibson/Steganography#WavSteg>

2 Background

2.1 The DNS protocol

History

The Domain Name System was developed by Paul Mockapetris at the University of Southern California during the expansion of ARPANET (Advanced Research Projects Agency Network), which was supported by DARPA (Defense Advanced Research Projects Agency).[1] DNS is a distributed naming system that returns resource records associated with requested domain names. These records most commonly contain IP addresses, but DNS can also return other types of information depending on the requested record type.

Distributed Architecture

The Domain Name System is a simple query-response protocol whose messages in both directions have the same format.[2] It is organized as a highly hierarchical distributed system, at the top are the root nameservers, which serve the root zone and direct resolvers to the appropriate top-level domain (TLD) nameservers. Below them are TLD nameservers, which are responsible for domains such as .com and country code TLDs such as .dk. These servers do not usually store the final records for individual hosts, instead they refer resolvers to the authoritative nameservers for a given domain. Authoritative nameservers are responsible for specific DNS zones, such as example.com and provide the actual records for subdomains within those zones such as www.example.com or mail.example.com.

Security

DNS has been exploited for decades and some attacks have had more longevity than others. In 2008, Dan Kaminsky reported a cache poisoning vulnerability affecting many recursive nameserver implementations.¹ By forcing the resolver to query random subdomains an attacker could repeatedly race the legitimate DNS response with forged replies until one matched the expected transaction ID and port. If successful, the resolver would cache malicious DNS data, causing normal looking domain names to resolve to attacker controlled IP addresses.

2.2 DNS Tunneling

DNS has also been used to establish C2 between attacker controlled servers and malware infected machines. This is due to DNS being a foundational protocol of the internet, any website or service that an API call is sent to is first found through DNS, which results in network security and firewall rules being more lenient on DNS traffic and UDP port 53.[3]

By using a DNS tunnel, arbitrary messages within the constraints of the domain name syntax defined in RFC 1035 [4] can be encoded into DNS query names, producing FQDNs

¹<https://www.kb.cert.org/vuls/id/800113>

(Fully Qualified Domain Names) such as

```
my-super-secret-data.dns-tunnel.attacker.com.
```

When the victim sends this DNS query to a recursive resolver such as Google's 8.8.8.8, the resolver follows the DNS delegation chain and eventually forwards the query to the authoritative nameserver for `attacker.com`. If the attacker controls that nameserver, the encoded data in the queried subdomain is delivered to the attacker. The attacker does not need to return a valid IP address for the queried subdomain because the data has already been exfiltrated by the time the DNS query reaches the attacker. From the victim network's perspective, the outbound traffic may appear to be only a DNS request to a trusted recursive resolver although more advanced monitoring could still detect suspicious query names.

Existing Solutions

There are many open source tools made for DNS tunneling. They generally rely on the same basic principles. Application data is encoded into DNS labels and transported through DNS queries and responses. Encodings such as Base32 and Base64 are used because DNS labels only allow a restricted character set and are subject to strict length limitations. As a result, these tools produce long, high entropy domain names, which makes them relevant when discussing DNS tunneling detection.

iodine is a tool for tunneling IPv4 traffic through DNS.² It is intended for situations where general internet access is blocked, but DNS queries are still permitted. Iodine creates a virtual network interface on the client machine and transports IP packets through DNS messages to an authoritative server controlled by the attacker. Before the tunnel begins, the client authenticates to the server using an MD5 based challenge-response mechanism. Iodine does not encrypt the tunneled traffic instead, it focuses on encapsulation and encoding. It supports several encodings, including Base32 (`[a-z0-5]`), Base64 (`[a-zA-Z0-9+-]`), Base64u (`[a-zA-Z0-9_-]`), and Base128 (`[\x80-\xFF]`). After the server received the data, it reassembles it into network packets and handles them accordingly.

dns2tcp is a tool for relaying TCP connections over DNS.³ Unlike iodine, which creates a general IP tunnel, dns2tcp is designed around specific TCP services. The server exposes named resources such as SSH and the client connects to one of these resources through DNS traffic. The tool provides only a simple identification mechanism and does not provide encryption, meaning that DNS is used as a transport layer rather than as a secure channel. This makes dns2tcp narrower in scope than iodine, but still useful for bypassing restrictive network environments where DNS is allowed while direct TCP connections are blocked.

dnscat2 is designed to create an encrypted command and control (C2) channel over DNS.⁴ Its purpose is different from iodine and dns2tcp. Rather than providing a general IP tunnel or forwarding a specific TCP service, the tool focuses on interactive C2 communication. It can be used to send commands and transfer data through DNS while encrypting the application layer content.

In Table 2.1 is a comparison of the above mentioned tools collected by Adiwat et al.[5] extended with the proposed tool, FTExfil.

²<https://github.com/yarrick/iodine>

³<https://github.com/alex-sector/dns2tcp>

⁴<https://github.com/iagox86/dnscat2>

Tool	Language	DNS Record	Encoding	Platforms	Encryption
iodine	C	NULL	Base32/64	Linux, Unix, macOS, Windows	No
DNSCAT2	C and Ruby	CNAME, MX, TXT	HEX and NetBIOS	Linux, Unix, macOS, Windows	Yes
DNS2TCP	C	TXT, KEY	BASE64	Linux	No
FTExfil	Rust	A, TXT	Regex-based / Weighted frequency shaped encoding	Linux, macOS, Android via Termux	Yes

Table 2.1: List of DNS tunneling attack tools[5], extended with the proposed proof of concept FTExfil.

2.3 Detection Methods

For institutions dealing with sensitive data, detection of DNS covert channels is essential in the prevention of data leakage and protection against malware. Thus, detection is a fairly well researched field.

Detection techniques can be divided into two separate categories, payload and traffic analysis.[6] In the case of payload analysis, individual packets are reviewed for any indicators of tunneling activity (seen in Table 2.2).

Indicator	Relevance
Long FQDNs or labels	High payload density per DNS request
High hostname entropy	Encoded payloads deviate from lexical domain patterns
Unusual character composition	Excessive digits or atypical symbol distributions
Uncommon record types	Potential misuse of TXT, NULL, or similar records
Policy violations	Direct external DNS queries bypass enterprise resolvers
Tool-specific signatures	Known tunneling utilities exhibit identifiable artifacts

Table 2.2: Payload level indicators of DNS tunneling.

While traffic analysis observes DNS traffic in a broader sense over time, looking for recurring patterns and anomalies such as unusually high number of requests. A comprehensive list of such indicators and their relevance can be seen in Table 2.3

Indicator	Relevance
High DNS volume per client	Excessive query rates may indicate tunnel activity
High DNS volume per domain	Sustained traffic to a single root domain is anomalous
Many unique hostnames per domain	Tunnels generate distinct FQDNs to carry data and avoid caching
Unusual geographic destinations	DNS traffic to unexpected regions may indicate unauthorized infrastructure
Recent domain or NS changes	Newly configured domains may support temporary tunnel infrastructure
High NXDOMAIN volume	Failed lookups may reflect automated or malformed tunnel queries
Orphan DNS requests	DNS queries without follow-on application traffic suggest nonstandard use
Traffic visualization anomalies	Tunnel activity may appear as distinct temporal or volumetric patterns

Table 2.3: Traffic-level indicators of DNS tunneling.

Character Frequency Analysis

Born and Gustafso [7] proposed detecting DNS tunnels by analysing the character frequencies of domain names. The approach is based on the observation that legitimate domain names do not follow a uniform distribution. Instead, similarly to natural language, they exhibit recognizable unigram, bigram and trigram patterns.

The authors compare English text, popular domain names, randomly generated domains and DNS tunnel traffic. It was found that legitimate domains have biased character distributions, where some characters and n-grams occur much more frequently than others. Randomly generated domains produce a much flatter distribution.

The method was then tested against tunneling tools such as iodine and dns2tcp where the tunneled traffic showed distributions that differed from normal subdomain traffic distribution. However, the method also implies a limitation, if a tunnel deliberately shapes its output to resemble legitimate DNS character frequencies, frequency based detection may become less effective.

Character Entropy Analysis

Another payload level indicator as shown in Table 2.2 is hostname entropy. While character frequency analysis compares the distribution of unigrams, Shannon entropy compresses this into a measure of randomness.

Claude Shannon first defined entropy as a measure of information, choice and uncertainty associated with a probability distribution. [8]

When applied to a DNS label, entropy measures the average uncertainty of each character in that label with the unit of bits/character. Encrypted data produces character distributions that are close to uniform. A uniform distribution has higher entropy than a word like distribution. Born et al, made the observation in [7] that normal domains have recognizable character frequency patterns, while tunneled traffic is more evenly distributed.

3 Methodology and Requirements

3.1 Software Engineering Methodology

This project followed an iterative software engineering methodology. The goal of the project is not to develop a production ready DNS tunneling software, but to design and implement a proof of concept.

The development process was divided into several phases. First, background research was conducted on the DNS protocol as well as tunneling techniques and common detection methods. This phase provided the basis for identifying the technical constraints of the system including DNS label length limits and relevant detection indicators. Based on this analysis, functional and non-functional requirements were written.

The project was planned using the Gantt chart shown in Figure 3.1. The chart was created at the beginning of the thesis period. It served as an initial planning tool rather than a strict development timeline. Since the project involved experimental encoding methods and prototype development, some phases were revisited as implementation constraints became clearer.

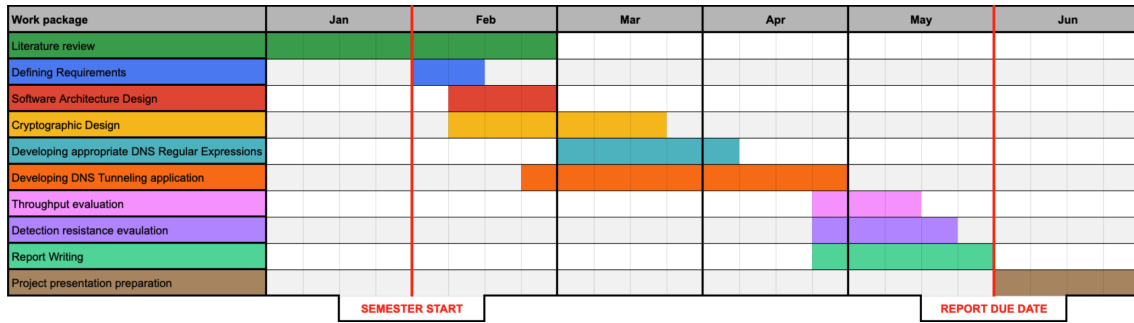


Figure 3.1: Initial project Gantt chart.

3.2 Requirements

Requirements define what the proposed proof of concept application shall do and which constraints it must satisfy.

Functional requirements (Table 3.1) describe the expected behaviour of the system, while Non-Functional requirements (Table 3.2) describe quality attributes.

Functional Requirements

ID	Requirement
FR1	The system shall allow a user to transfer data from a client to a server using DNS queries.
FR2	The system shall transform the input file into smaller chunks that fit into subdomains before transmission.
FR3	The system shall include metadata alongside the file chunks for the receiver to sufficiently reconstruct the file.
FR4	The system shall transform transferred data into query names compatible with DNS constraints.
FR5	The system shall allow the receiver to extract transferred data from incoming DNS queries.
FR6	The system shall allow the receiver to reconstruct the original file from subdomains received in queries.
FR7	The system shall handle duplicate and out of order data chunks.
FR8	The system should support key exchange between client and server.

Table 3.1: Functional requirements for the proof of concept application.

Non-functional Requirements

ID	Non-functional requirement
NFR1	Generated query names shall comply with DNS syntax and length constraints.
NFR2	Reconstructed files shall be byte identical to their original input files.
NFR3	The system shall protect transmitted data against unauthorized reading.
NFR4	The system shall detect unauthorized modification of transmitted data.
NFR5	The system shall be modular that individual components can be modified independently.
NFR6	The system shall expose runtime configuration options without requiring source code changes.

Table 3.2: Non-Functional requirements for the proof of concept application.

4 Analysis and Design

4.1 Encoding

To construct a DNS tunnelling scheme that is less likely to match the detection methods discussed in Section 2.3, the encoded payload must satisfy several constraints. First, it must be valid inside DNS query names, meaning that the output must respect label length limits and use characters that are safe in domain names. It should also avoid character patterns commonly associated with existing tunnelling encodings, such as long Base32 or Base64 strings containing uppercase letters, digits or other symbols. The encoded output should also not resemble uniformly random ciphertext, since a near uniform character distribution increases entropy and can make the query name easier to distinguish from ordinary hostnames.

Format Transforming Encryption

Format Transforming Encryption (FTE) as a cryptographic primitive was first introduced by Dyer et al, in 2013.[9] It extends symmetric key encryption with an encoding layer that turns binary ciphertext into a regular expression (regex) defined format. In its original paper the scheme was used to attempt protocol misidentification in deep packet inspection systems which it did successfully.

The FTE encryption function $Enc(K, F, M)$ uses AES-CTR mode of operation (which essentially turns the block cipher into a stream cipher) with a normal symmetric key K . The ciphertext is then authenticated via an HMAC resulting in an intermediate ciphertext Y . A language $L(F)$ is constructed from a format F defined by a regex. The intermediate ciphertext is treated as an integer in $\mathbb{Z}_{|L(F)|}$. An *unrank* function is applied over $\mathbb{Z}_{|L(F)|} \rightarrow L(F)$, which maps integer Y to a string at its index in $L(F)$ resulting in ciphertext C .

Decryption $Dec(K, F, C)$ is done in a similar fashion, applying the reverse function $rank L(F) \rightarrow \mathbb{Z}_{|L(F)|}$ to retrieve Y and then the plaintext M .

The scheme was developed into a Tor pluggable transport (PT) called FTEProxy. Since Tor traffic results in recognisable patterns, FTE provides a way to match the format of HTTP requests instead of random looking encrypted traffic. To prove its effectiveness a Tor tunnel with FTEProxy enabled was deployed on a VPS hosted in China. Experiments concluded that regular Tor traffic was blocked by the Great Firewall of China, however the obfuscated tunnel remained unblocked.

In other works by Oakley et al,[10] FTE was used to enable a proof of concept covert channel through the Synchronphasor protocol used to stream real time high frequency electrical measurements from power grids. The implementation employs a so called observation based FTE, using a collection of captured Synchronphasor packets to generate a language.

Reducing Shannon Entropy with FTE

Format transforming encryption can also be used as an encoding layer for tunnelled DNS traffic. By choosing a regular expression which conforms to the characters used in FQDNs, we may reduce the effectiveness of detection methods that are based on long labels, unusual

character composition, hostname entropy or signatures that are specific to existing tools or encoding schemes.

The simplest case is reducing the output size such that before our controlled domain there are 25 URL safe Base32 characters:

$$^[A-Z2-7]\{25\}\.example\.com\$$$

this results in a string similar to MZXW6YTBOIABCDE234567AB.example.com. It is short, however it still resembles conventional high entropy DNS tunnelling encodings.

This of course is all dependent on the chosen regex and our willingness to sacrifice throughput for the sake of detectability.

Let $\mathcal{A}$ be the alphabet of characters allowed by the chosen regular expression and let X be the random variable representing one encoded output character. Throughput can be generalized using Shannon entropy with the formula

$$H(X) = - \sum_{x \in \mathcal{A}} P(x) \log_2 P(x)$$

where x is a character from the alphabet $\mathcal{A}$, and $P(x)$ is the probability with which x appears in the encoded output.

Assuming the intermediate ciphertext is computationally indistinguishable from random the ranking step produces outputs that are uniform distributed over the chosen format. Thus, in the regular expression above there are 32 possible characters each equally as likely as the other $P(x) = \frac{1}{32}$ making the theoretical entropy

$$H(X) = - \sum_{x \in \mathcal{A}} \frac{1}{32} \log_2 \left(\frac{1}{32} \right) = -32 \cdot \frac{1}{32} \log_2 \left(\frac{1}{32} \right) = -\log_2 \left(\frac{1}{32} \right) = \log_2 32 = 5$$

this is consistent with the 5 bit/char encoding of Base32 specified in RFC 4648[11].

The throughput T of an encoding with this regular expression (which features 25 uniformly picked characters) can be calculated via

$$T = 25 \cdot H(X) = 25 \cdot 5 = 125$$

This gives a theoretical capacity of 125 bits encoded in a 25 byte subdomain. If the payload is handled in whole bytes, this corresponds to 15 full bytes, with 5 bits unused.

Of course the premise of this paper is to not encode subdomains in already existing, high entropy methods thus we can reduce our language to not feature characters that are uncommon in subdomains. These include capital letters, numbers and hyphens. It would be incorrect to state that these characters do not occur in domains however they are most definitely not as common as regular lowercase letters of the latin alphabet (a-z).

An example of such format is

$$^[a-z]\{25\}\.example\.com\$$$

with the same calculations we can conclude that this format contains $\log_2 26 \approx 4.7$ bits of entropy, making the theoretical throughput $25 \log_2 26 \approx 117$ bits, 14 bytes. These results are closer to a real domain name however trying to transfer a 1MiB file over DNS would result in 74899 packets sent, highly inefficient.

In today's age, Content Distribution Networks (CDNs) create rather long subdomains, so what if we can get away with as long of an FQDN as possible? It would result in the following

$$\wedge [a-z]\{63\} \backslash . [a-z]\{63\} \backslash . [a-z]\{63\} \backslash . [a-z]\{49\} \backslash . \text{example} \backslash . \text{com} \$$$

resulting in a throughput of $238 \log_2 26 \approx 1119$ bits 139 full bytes over a 253 character FQDN. The same 1MiB file now "only" takes 7544 DNS queries, while our theoretical character entropy still sits at $\log_2 26 \approx 4.7$ bits.

Huffman coding

Huffman coding is a lossless data compression method introduced by David A. Huffman in 1952.[12] It reduces the number of bits required to represent symbols by assigning shorter binary codes to more frequent symbols and longer binary codes to less frequent ones.

The method builds a binary tree from the bottom up using the frequencies or estimated frequencies of the symbols in the data to be compressed. These frequencies are treated as weights. Symbols with higher frequencies generally appear closer to the root of the tree, while less frequent symbols appear deeper in the tree.

The compression process is as follows:

1. Collect each unique symbol appearing in the input data.
2. Count or estimate the frequency of each symbol.
3. Create a leaf node for each symbol, weighted by its frequency.
4. Build the Huffman tree by repeatedly combining the two nodes with the smallest weights into a new parent node.
5. Assign binary values to the tree edges, for example 0 for the left edge and 1 for the right edge.
6. Assign each symbol's Huffman code by following the path from the root to the corresponding leaf node.
7. Encode the input data by replacing each symbol with its Huffman code.
8. As a result, more frequent symbols generally receive shorter codes, while less frequent symbols receive longer codes, for example $a := 010$ and $z := 10010$ depending on the input frequencies.

Character frequency shaping with Huffman coding

Using the frequency based ranking property of Huffman coding, it is possible to construct a Huffman tree that is (unlike in its use for compression) not based on frequencies of existing input data, but rather a desired frequency after decompression. A Huffman tree built on the letter frequencies found in the English language according to the Oxford College Math Center at Emory University [13] can be seen in Figure 4.3. Since after compression the result is a binary bitstream similarly to regular ciphertext, we can handle the encrypted bitstream as compressed data and parse it using a Huffman tree built from a target distribution. Each decoded leaf emits one output character. This uses the Huffman tree in the reverse direction of ordinary compression and results in a frequency shaped output.

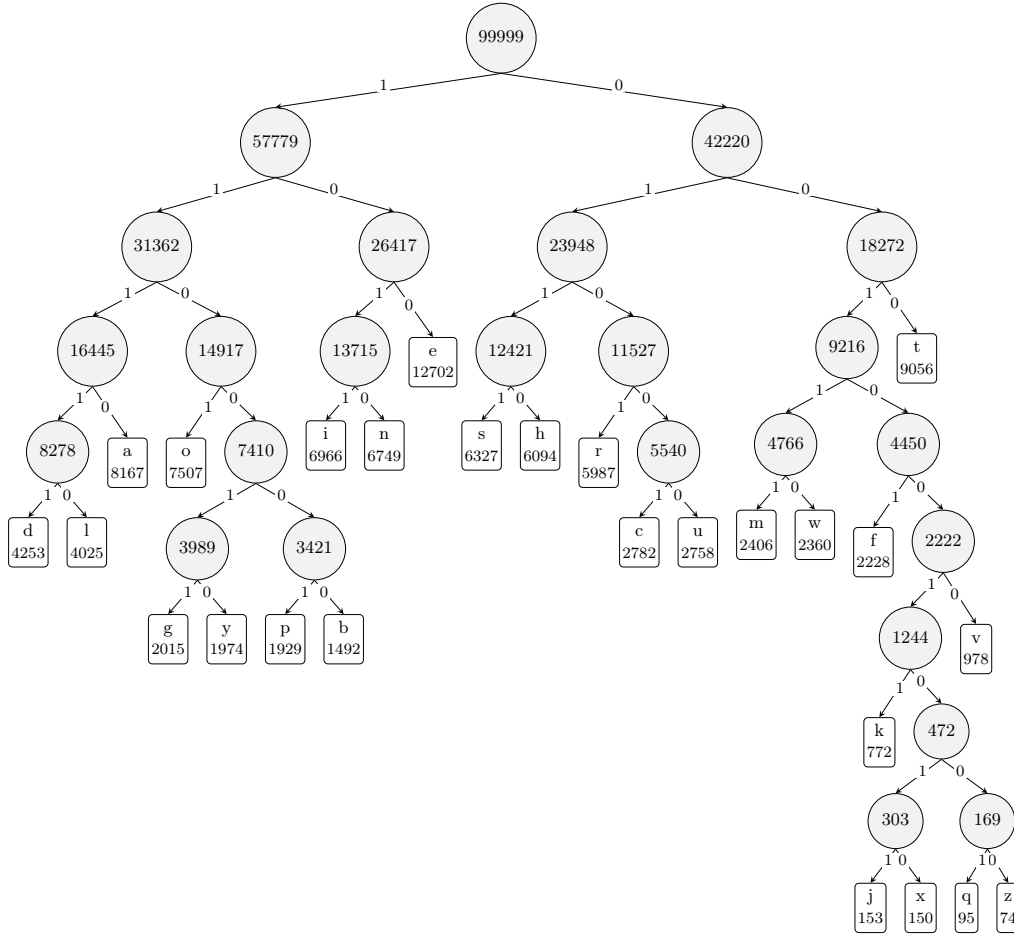


Figure 4.1: Huffman tree built based on the letter frequencies found in the English language

Let M be the plaintext, K the encryption key and let

$$C = \text{Enc}(K, M)$$

be the ciphertext. Since C is the output of an encryption function, a randomly selected output bit b is expected to be (approximately) uniformly random:

$$P_C(b = 0) \approx P_C(b = 1) \approx \frac{1}{2}$$

We transform this uniform ciphertext representation into an output of which character frequencies resemble a target distribution. This target distribution is

$$P_{\text{target}}(x)$$

where x is a character in the output alphabet. For example, P_{target} may be based on English letter frequencies or measured from a domain name dataset.

Instead of constructing a Huffman tree from ciphertext frequencies (which is uniform binary), the tree is constructed from P_{target} . The ciphertext C is then transformed using this Huffman tree producing a shaped output S :

$$S = \text{Huff}_{P_{\text{target}}}(C)$$

The character distribution created by this process is denoted by P_S . This distribution is determined by the lengths of the Huffman codes. For a character x , this gives

$$P_S(x) = 2^{-\ell(x)}$$

where $\ell(x)$ denotes the number of bits in the code for x .

The distribution P_S therefore does not exactly equal P_{target} . Rather the Huffman tree provides an approximation where characters with higher target probabilities in P_{target} receive shorter bit codes and are more likely to occur.

Since per character Shannon entropy in a string depends on the character distribution of that string, changing the output character frequencies also changes the entropy of the resulting representation. Let X be the random variable representing one emitted character of the shaped output. Using the formula for Shannon entropy, the per character entropy of the output can be written as

$$H(X) = - \sum_{x \in \mathcal{A}} P_S(x) \log_2 P_S(x)$$

where $\mathcal{A}$ is the output alphabet. For uniformly distributed ciphertext bits, the probability of reaching the leaf for character x is determined by the length of its bit code, giving $P_S(x) = 2^{-\ell(x)}$. Therefore this can also be written as

$$H(X) = \sum_{x \in \mathcal{A}} 2^{-\ell(x)} \ell(x)$$

Since $P_S(x) = 2^{-\ell(x)}$, this value also describes the expected number of ciphertext bits consumed per output character. Given an encrypted bitstream of length b bits, the expected number of output characters $|S|$ is therefore approximately

$$|S| \approx \frac{b}{H(X)}$$

The Huffman encoding not only affects the visible character distribution, but also the Shannon entropy of the encoded output. A more uneven distribution where characters occur at varying frequencies generally has a lower entropy than a uniform distribution over the same alphabet. However, the entropy measured from a finite output may differ slightly from this theoretical value.

The full process can therefore be summarized as

$$M \longrightarrow C = \text{Enc}(K, M) \longrightarrow S = \text{Huff}_{P_{\text{target}}}(C)$$

An example frequency analysis of the following encodings can be seen in Figure 4.2. This plaintext is 194 byte long with Shannon entropy of ~ 4.25 bits:

“There are two kinds of cryptography in this world: cryptography that will stop your kid sister from reading your files, and cryptography that will stop major governments from reading your files.” [14, preface]

after encrypting and encoding the plaintext in Base64 we get the following 260 byte string with ~ 5.79 bits of entropy (close to the theoretical 6 bit maximum [11]):

```
eDP601yA41C7bTLerBK6Q/0GCY8J002U13961sTe9paLoHQJlWQxydbZ4reXv8Q6y59
if3pZGi7jVEvKHZ1YAi0phQ6A/h5tebBgFB05pLQebUPxa93tIhW+BkCIphHH1/FM3u
K8+X5aQ/3kLiCAXrIVjyettzj7r3fZJ9hMGAPA2ZAMGl0saCsZS1ao/SeccSrmKYyf2
6gcWioiZ6jPFD9J8RQLS8eW/WD8ZsnjnBFpqly3hlHbZm3R6SkjcXNRsg==
```

Meanwhile, after encoding the same encrypted text via our Huffman encoder built around the English language frequencies results in a much larger, 371 byte string with a measured ~ 4.12 bits of entropy:

```
setgdrnhieetmtentihihowrecretenaeemduwtcesekntaoenhiidsnohpwllofnfot
taetccenivhmknioimerhlradavsreiedhkdsnreeheropnnjyemhsrhttvicceftsuts
ltlhoseoetbtrtmcgcfntseonemafnlalovefhdttytverwtestlrdewhlafsedfdf
nurdseeftptvtfaivrhmeelihogesoosiayclbebhtrnoeytththeinrehetnpenr
nirudncastaxiepcchdisrttafnfuecelnwsertsaeedtertticseeseisaitmawhys
esmtvicniryoyeshoeghaesncvopswnwy
```

This length is consistent with the expected output length. Since the encrypted input is 194 bytes it contains 1552 bits. For the English frequency based Huffman tree the theoretical per-character entropy is $H(X) \approx 4.19$. Therefore, the shaped output length can be approximated as

$$|S| \approx \frac{1552}{4.19} \approx 370.4$$

characters.

The change in size can be accredited to the uneven length of Huffman codes. While Base64 maps every 6 bits of input to one output character as specified in RFC 4648 [11], the Huffman encoder uses variable length bit codes. Characters that are common in the target distribution like are represented by shorter codes, while characters that are rare require longer ones. For the English Huffman tree the shortest codes are *e* and *t* with 3 bits, while the longest ones are *j*, *x*, *q* and *z* with 9 bit codes.

In the case of binary data, the encoder cannot exploit the redundancy normally present in English text. Instead it approximates the target frequency distribution onto the output.

This specific aspect is the biggest trade off in comparison to other encodings used in DNS tunneling, since it directly affects the proposed tool's throughput.

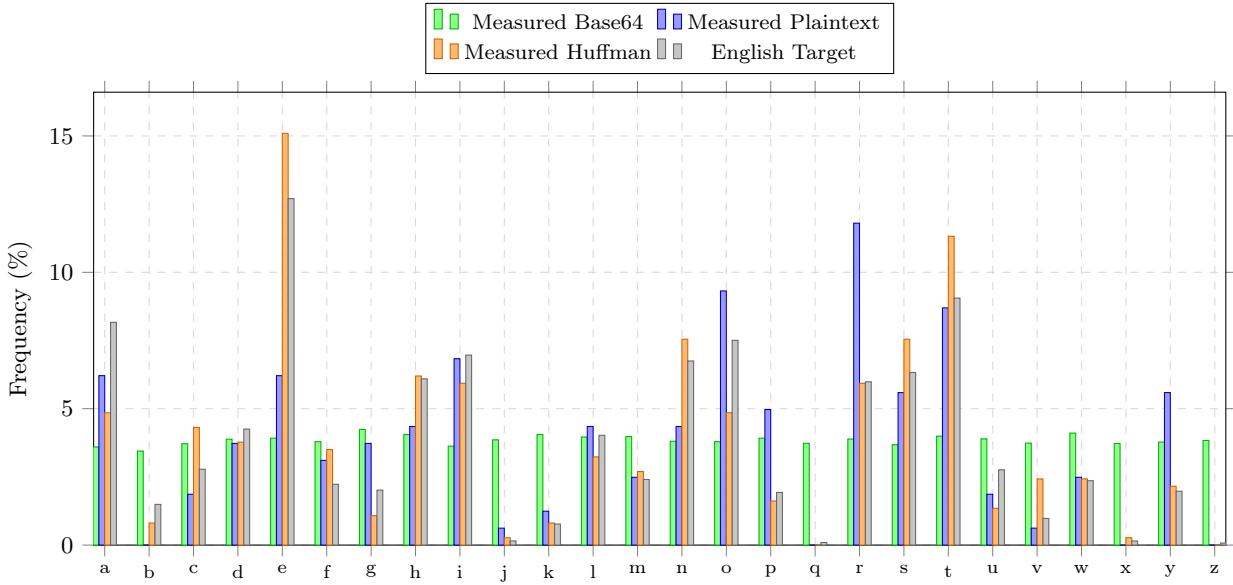


Figure 4.2: Character frequencies in Bruce Schneier quote over a-z for Base64, plaintext, Huffman and English frequency baseline.

Stage	Sample output	Top frequencies
Plaintext	this text is about to be encoded	t 19.23% e 15.38% o 11.54% b 7.69% d 7.69% i 7.69% s 7.69% a 3.85%
AES-CTR → Base64	SWYEnn47MOe+2wJunln73ZLfpOEztLRfcCq6 IddCD67u1G5ONpXXLe0UNCntJFQGDC5yiV9/ gP6HUQnQqli+gg	. 1.98% g 1.83% Q 1.75% A 1.74% z 1.69% 7 1.69% f 1.69% u 1.67% w 1.66%
AES-CTR → Huffman	smeecrsrolllnlhnmisytichit rdduwtfeskwfdrteaientdwsuoot dtuneehywatdtwekatauasshoqt rtivwtrrwunnhastoveoaesttsteiftsstwneco	t 15.45% e 9.76% s 8.94% a 6.50% w 6.50% i 5.69% n 5.69% o 5.69%

Figure 4.3: For some ciphertext after Huffman encoding, the distribution is more aligned

However, as Born et al, [7] also highlights the frequency of characters in the English language and FQDNs do not exactly match. Thus a separate frequency analysis was done on a corpus of crawled hostnames from project scanner.ducks.party¹ containing 2,092,120,183 processed hostnames. The analysis itself was done parallelized on SDU’s ucloud with a 64 core linux instance. Frequencies were computed over normalized hostnames (lowercase

¹<https://ducks.party/ct>

and public suffix removed), then restricted to *a-z*. As seen of Figure 4.4 The resulting distribution clearly differs from standard English frequencies, which motivates using a DNS specific target distribution when shaping output distribution by Huffman trees. A Huffman tree constructed based on the DNS derived character frequencies can be seen on Figure 4.5.

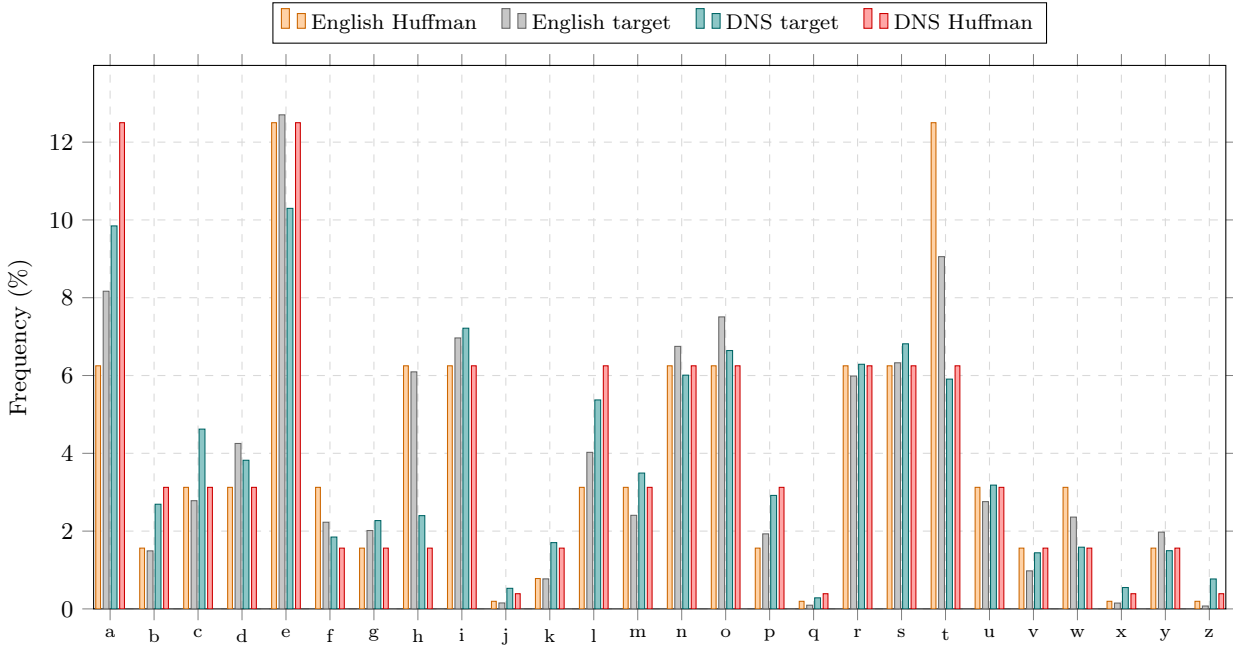
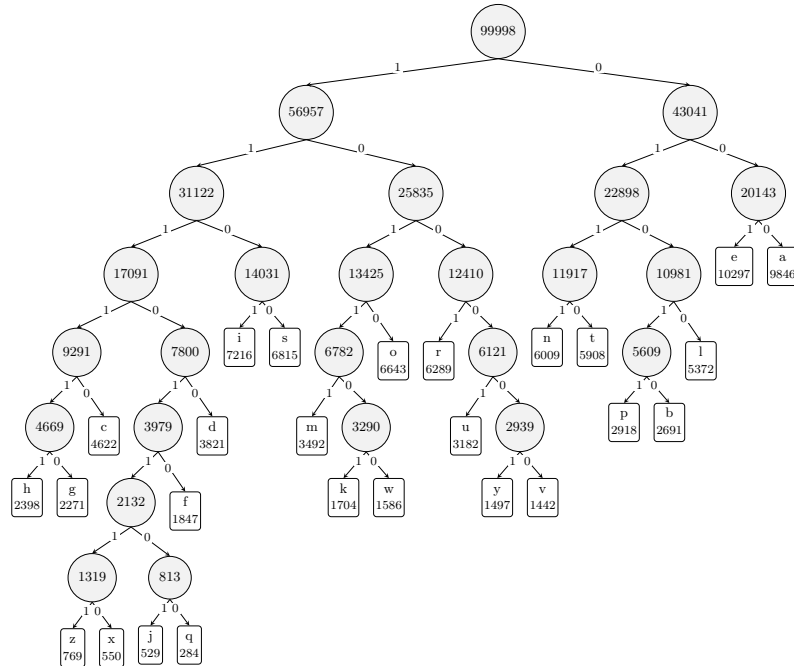


Figure 4.4: Character frequency comparison over a-z between English and DNS target and Huffman tree derived distributions.



4.2 Key Exchange

Some DNS tunneling tools such as Iodine, use a password based challenge-response authentication. The server chooses a password, the client sends an initial request asking to be authenticated. The server then responds with a 32 bit integer which is called the challenge. The client must MD5 hash the password and use the bitwise XOR operation to combine the two values. The combined value is then sent back to the server which, upon satisfactory comparison establishes the tunnel session.

However, in our use case a simple password based authentication handshake is not sufficient. Since the goal is to establish an encrypted tunnel that is not dependent on the tunneled protocol's encryption we must propose a key exchange.

Diffie-Hellman

The most famous and widely adopted cryptographic key exchange protocol is the Diffie-Hellman (DH) key exchange[15]. The protocol can be described as the following process.

The client and server agree on two public numbers p and g . The client picks a secret number a , computes its public key ($A = g^a \bmod p$) and sends it to the server. The server picks a secret number b , computes its public key ($B = g^b \bmod p$) and sends it to the client. Both compute the shared secret K via $K = B^a \bmod p$ or $K = A^b \bmod p$.

However, the public keys in Diffie-Hellman are usually quite large because they are based on very large numbers. A modern public key size is at least 2048 bits and stronger versions use 3072 bits or more.[16] These sizes make it quite difficult for an attacker to reverse the calculation and discover the private keys. In practice however, these public key sizes would be too large to fit in a 255 byte (2040 bit) FQDN, considering it also includes the label separating dots, top level domain names and the attacker controlled domain.

Elliptic-Curve Diffie-Hellman

Fortunately for our DNS tunnel, there are versions of Diffie-Hellman that use elliptic curves.[17] Instead of having to compute powers modulo a large prime, Elliptic-Curve Diffie-Hellman uses scalar multiplication on an elliptic curve E . The private keys are a randomly chosen integer d . The public key is computed by multiplying this integer by a fixed public point G on the curve resulting in $Q = dG$. Here, multiplication means repeated application of the elliptic curve group operation and not the ordinary numerical multiplication. The parties can then combine their own private keys with the other party's public value to derive the same shared secret S .

This elliptic curve variant allow much smaller public keys while providing comparable security. For example, using the elliptic curve X25519[18] results in public keys of the size 32 bytes (256 bits), while the curve P256[19] has a public key size of 65 bytes (520 bits). When encoded to be used in a DNS label, this becomes approximately 43 and 87 characters using unpadded Base64u or 52 and 104 characters using Base32. This fits within the 63 byte per DNS label limit (even though P256 requires fragmentation across labels in the same query), whereas a 2048 bit (256 byte) public key would require fragmentation across multiple DNS queries.

The output of ECDH shall not be used directly as an encryption key since it lacks formal qualities required by a symmetric key such as length or uniformity.[20] Instead, the shared secret is passed through a key derivation function, such as HKDF-SHA256[21] together with context information to create a session key. This binds the derived key to the exchange.[22]

A sequence diagram of this process can be seen in Figure 4.6.

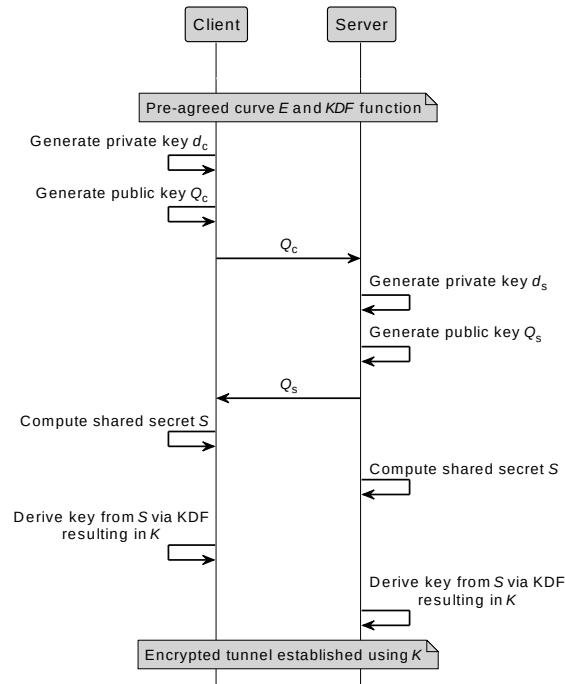


Figure 4.6: Key exchange between client and server.

4.3 Tunneling

This section describes the design of the proposed DNS tunneling tool. The design builds on the DNS tunneling principles introduced in Chapter 2.

Design Scope

The proposed tunnel is designed to transfer files through DNS query names. Unlike general purpose DNS tunnels introduced in previous chapters that transport IP packets, this design focuses on one directional file transfer from a client to an authoritative server. The input file is split into chunks, each chunk is encoded into a DNS compatible query name and the server reconstructs the original file from the received chunks.

System Architecture

The system consists of a client, the DNS resolution infrastructure and an authoritative server controlled by the receiver. The client constructs DNS queries under a controlled domain. Recursive resolvers forward these queries according to normal DNS resolution, causing the encoded query names to reach the authoritative server.

The authoritative server extracts the encoded part of the query name, decodes it and inserts the resulting chunk into a reassembly buffer.

Figure 4.7 shows a sequence diagram of the proposed tool.

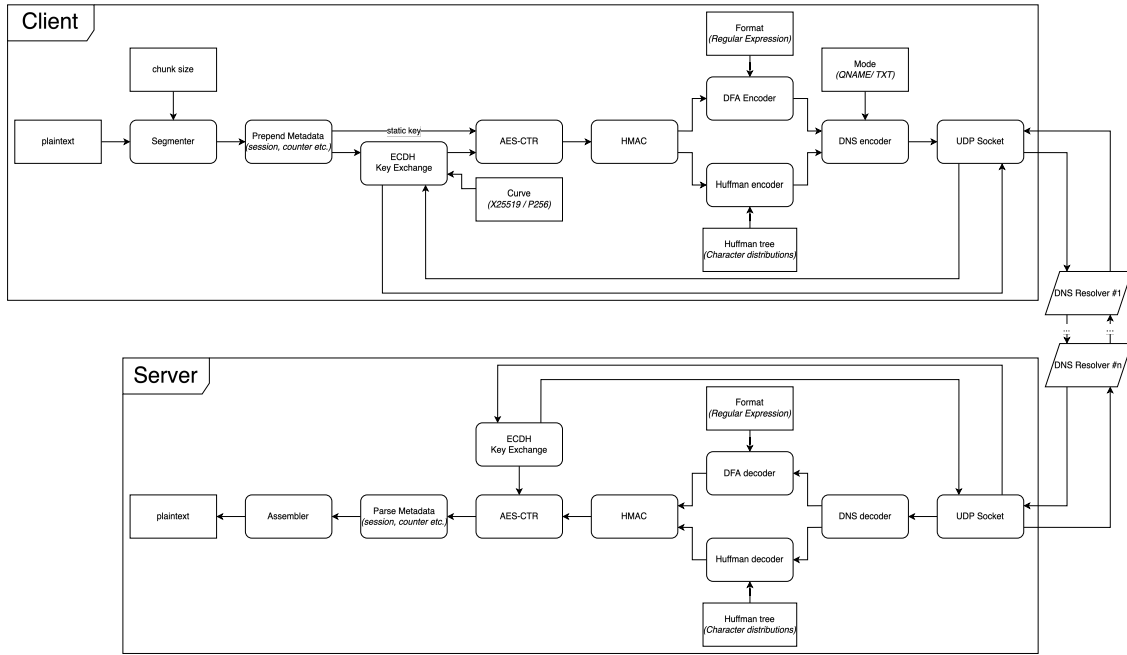


Figure 4.7: Sequence diagram of the proposed DNS tunneling tool

Transfer Pipeline

The transfer process consists of the following stages:

1. A session is established between client and server, with or without key exchange.
2. The input file is read as a byte sequence.
3. The file is divided into payload chunks.
4. Metadata is appended to each chunk.
5. The resulting chunk is encrypted and encoded into a DNS query name, either via FTE or Huffman coding.
6. The client sends the query to a resolver or directly to the authoritative server.
7. The server extracts and decodes the query name.
8. The decoded chunk is inserted into the corresponding transfer session.
9. Once all chunks have been received, the original file is reconstructed.

Establishing a Session

Encryption can operate with either a hard coded key (strictly for debugging purposes as it does not provide meaningful secrecy) or an ECDH key exchange. The client sends a query requesting key exchange with information containing session identifier, the requested key exchange mode and its public key. The server replies in the query response with its own public key. Both parties can then compute the shared secret and derive the session key with a pre agreed key derivation function. Further details of the key exchange scheme is found in subsection 4.2.

Chunking and Metadata

Before encryption, each subdomain consists of a payload fragment and a fixed size metadata section. The metadata allows the receiver to identify the transfer, filter duplicates, observe the progress, place the chunk in the correct order and determine when the complete file has been received.

The metadata section contains the following fields:

Field	Purpose
Version	Identifies the metadata format.
Flags	Provides space for transfer options and future extensions.
Payload length	Indicates how many bytes of the chunk are file payload.
Chunk index	Identifies the position of the chunk in the transfer.
Total chunks	Indicates how many chunks are required to reconstruct the file.
File identifier	Groups chunks belonging to the same file transfer.

Table 4.1: Metadata fields used for chunk reassembly.

Encryption

Encryption is done with the Advanced Encryption Standard in Counter mode of operation (further referred to as AES-CTR).

AES-CTR operates as a stream cipher (created from the AES block cipher), counter mode is an application of the forward cipher (AES) to a set of input blocks, called counters to produce a sequence of output blocks that are then XORed with the plaintext to produce the ciphertext.[23] A visualization of the process can be seen in Figure 4.3. This mode of operation fits our needs perfectly since compared to others, there is no need for padding. Plaintexts can be arbitrary length which matters because each chunk is encrypted as a separate message thus, every chunk would contain a padding overhead in regular block cipher modes.

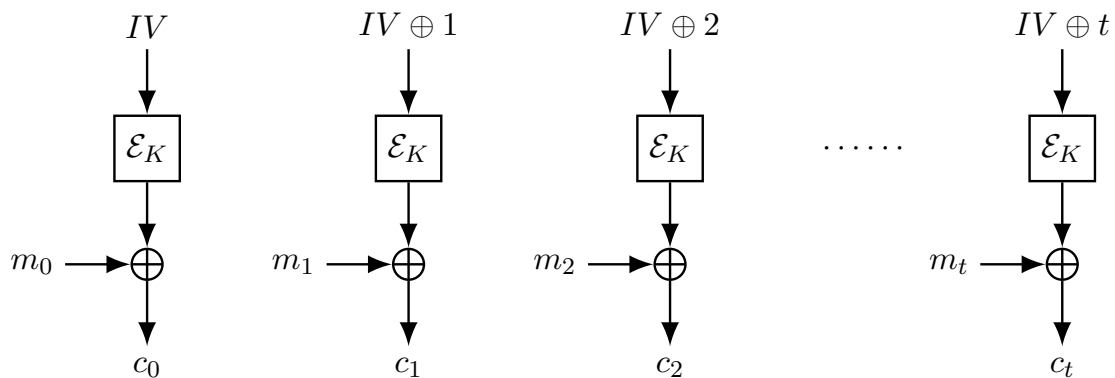


Figure 4.8: AES-CTR mode of operation.[24]

However, if we left it at that our ciphertext would still be vulnerable to tampering. This could occur if a DNS query is intercepted, decoded and then arbitrary bits were

XORed into the ciphertext. To combat this, each ciphertext is authenticated with a Hash-based Message Authentication Code (HMAC). The HMAC is essentially a code computed from the ciphertext with a second key, separate from the one used for encryption and a pre-agreed hash function. The resulting code is appended to the end of the ciphertext. Once a query is received and decoded, the server computes the same authentication codes and compares it with the one appended at the end of the query.

Encoding

The tunneling layer is independent of the encoding method used. After receiving the encrypted bitstream it passes it to the selected encoder. The two encoding methods are detailed in section 4.1.

FQDN Assembly

Each encoded chunk is placed in the first labels of a DNS query name followed by the controlled domain and top level domain name in the last two labels:

`<encoded-chunk>.<controlled-domain>.<TLD>`

This structure ensures that our queries are still routable to our controlled nameserver through normal DNS resolution, while also containing the encoded payload. The resulting FQDN must conform to both the maximum length of individual DNS labels and the maximum full domain name length. Therefore, the available payload capacity depends heavily on the length of the controlled domain and the selected encoding method.

Forwarding Queries

A DNS query is created from the resulting FQDN and forwarded to a specified nameserver. This can be either the controlled authoritative nameserver or a recursive resolver (such as 8.8.8.8).

File Reassembly

While the server is receiving the queries, it extracts, decodes and decrypts the subdomains. The resulting plaintext is parsed for its metadata fields and groups them by file identifiers. This allows the receiver to accept chunks out of order and reconstruct the original file once all expected indexes have been received. Duplicate chunks are ignored rather than appended again.

Limitations

The proposed tool reduces the character frequency and the character entropy payload level indicators however, it does not eliminate the traffic level indicators detailed in Section 2.3 Table 2.3. Most files still require many DNS queries and as a result, high volume of outbound DNS traffic and every transferred chunk produces a distinct, humanly unreadable subdomain.

The design also trades throughput for less uniform encoded output. Frequency shaped encodings can make query names appear less similar to conventional Base32 or Base64 tunnel payloads, but they increase the number of DNS characters required to transfer the same file.

5 Implementation

5.1 Implementation Overview

The proposed tool was implemented in the programming language Rust as a CLI application called `FTExfil`. The application supports two main operational modes, client mode for transmitting files via DNS queries and server mode for receiving those DNS queries and reassembling the original file.

The implementation closely follows the software design and architecture described in Chapter 4.

5.2 Project Structure

The implementation is divided into Rust modules (seen in Table 5.1) to promote loose coupling of software logic.

Module	Responsibility
app	Runtime orchestration for client and server modes.
transfer	Chunk construction, metadata encoding and server side reassembly.
codec	DNS compatible encoding and decoding implementations.
dns	DNS query construction and packet parsing.
crypto	Key exchange and session key derivation.
io	File reading and writing.
errors	Shared error representation used across the application.

Table 5.1: Main implementation Rust modules.

5.3 Command Line Interface

Client Mode

The client mode is used to transmit a file through DNS queries. Its command line form is:

```
ftexfil client <file_path> <size> <domain_suffix> --server <ip:port> \  
  [--encoding <regex|weighted>] [--qtype <A|TXT>] \  
  [--timeout-ms <ms>] [--retries <n>] \  
  [--retry-delay-ms <ms>] [--delay-ms <ms>] \  
  [--kex <none|x25519|p256>] [--resolver-safe]
```

The positional arguments define the input file, total chunk size and controlled authoritative name server domain. The required `-server` argument identifies the DNS server or resolver to which the queries are sent.

Argument	Purpose
<file_path>	Input file.
<size>	Total chunk size in bytes (must exceed the meta-data size).
<domain_suffix>	Controlled authoritative domain name used for query construction.
-server	Resolver IP address and port.
-encoding	Selects the regex based or Huffman (weighted) encoding profile.
-qtype	Selects whether queries use A or TXT DNS query types.
-kex	Selects static key mode or an ECDH key exchange mode.

Table 5.2: Client mode arguments of most importance.

Server Mode

The server mode receives DNS queries and reconstructs completed transfers. Its command line form is:

```
ftexfil server --listen <ip:port> --domain <domain_suffix> --out-dir <dir> \
  [--encoding <regex|weighted>] [--response-ip <ipv4>] \
  [--session-ttl-sec <n>] [--kex <none|x25519|p256>]
```

The server listens on a configured IP address, filters queries by the configured domain, decodes matching names and writes completed files to an output directory.

Argument	Purpose
-listen	IP address and port on which the server listens for incoming DNS traffic.
-domain	Controlled authoritative domain name accepted by the server.
-out-dir	Directory where reconstructed output files are written.
-encoding	Selects the encoding method expected by the server.
-response-ip	IPv4 address used in generated DNS responses.
-session-ttl-sec	Timeout for incomplete receiver sessions.
-kex	Enables or disables key exchange support.

Table 5.3: Server mode arguments of most importance.

5.4 Key Exchange

Key exchange is optional and can be activated through a runtime flag with values x25519 or p256. For mode none a hard coded static key is used, strictly for the purpose of development of the software.

The client begins by generating:

- a random 64 bit transfer identifier

- an ephemeral private key d_C for the selected curve
- the corresponding public key Q_C

The client then initiates a handshake via a DNS TXT query in the format:

`k1.<transfer_id_hex>.<curve>.<base32(Q_C)>.<domain>`

where `curve` is a short curve label. After receiving this request, the server:

1. validates the message structure and mode
2. decodes the client public key Q_C
3. generates its own ephemeral key pair d_S and Q_S
4. computes the shared secret S
5. derives a 32 byte session key using HKDF-SHA256
6. returns its public key in a TXT response:

`k1:<curve>:<base32(Q_S)>`

After the client has received the response, the same shared secret is computed and session key derived via HKDF-SHA256.

The key derivation step is the following:

$K_{\text{session}} = \text{HKDF-SHA256}(S, \text{salt} = \text{transfer_id}, \text{info} = \text{"ftexfil-kex-v1-}<\text{curve}>")$

where S is the raw shared secret and the output length is 32 bytes. The key K_{session} is then halved, resulting in K_1 and K_2 .

The following Rust crates were used in the implementation of the key exchange feature:

- `x25519-dalek` - X25519 operations
- `p256` - P-256 operations
- `hkdf` - HKDF key derivation
- `sha2` - SHA-256 hash for HKDF
- `data-encoding` - Base32 encoding/decoding for public key transport
- `rand` - cryptographic randomness for transfer id and ephemeral key generation

5.5 Encryption and Encoding

Encryption

The solution couples AES-CTR with HMAC for its encryption scheme as specified in the original paper by Dyer et al, which introduced FTE [9] and in Section 4.3.

A header `w1` is created containing a random initial value IV required for encryption, as well as the length of the payload. This header adds up to 16 bytes which are then encrypted in a 128 bit block of AES with the key K_1 .

A counter is built using the IV and K_1 which is then encrypted using AES. The resulting key stream is the XORed with the input data byte for byte, resulting in the encrypted payload w_2 .

An HMAC T is computed over the header and the ciphertext using a separate key K_2 . The chosen hashing function used for HMAC is HMAC-SHA512. Other versions such as HMAC-SHA256 were also considered, however surprisingly the function with a larger output size and more rounds performs 10-80% better on modern 64-bit CPUs.[25] The output size difference matters little in our case since we are truncating it to the first 16 bytes to preserve 48 bytes that can be used for payload data instead. The truncation should not fully compromise the security of the authentication code as NIST (National Institute of Standards and Technology) advises the length of a truncated HMAC output shall be at least 32 bits.[26]

The entire encrypted section is structured as

$$\langle w_1 \rangle || \langle w_2 \rangle || \langle T \rangle$$

with a length of $len(w_2) + 32$ bytes.

As with every implementation that uses cryptography there is a well-known rule to "Do Not Roll Your Own Crypto", meaning if there is no reason to implement a cryptographic primitive, an existing reviewed implementation shall be used. Thus in the encryption section the following Rust crates were used:

- `aes` - AES-128 block cipher primitive
- `ctr` - CTR mode
- `hmac` - HMAC construction
- `sha2` - SHA-512 hash (`Hmac<Sha512>`)
- `subtle` - constant time HMAC comparison
- `rand` - IV randomness
- `zeroize` - key material wipe on drop

FTE Library Integration

The format transforming encryption implementation used by `ftexfil` is provided by the Rust crate `fte_rs`. The FTE portion of this library was developed as a Rust reimplementation of the original Python `libfte`¹ project. Large parts of the translation process were assisted by a large language model, after which the resulting Rust code was reviewed, and modified.

The purpose of this reimplementation was not to introduce a new FTE construction, but to provide a native Rust compatible encoding and encryption layer instead of interfacing with the original Python library. The novel part of the implementation lies in the weighted encoding using Huffman coding.

In the regex based encoder, `fte_rs` compiles a regular expression (`(\d+)`) into a deterministic finite automaton (DFA). For a selected fixed output length (`fixed_slice`) the DFA module computes counts of accepted suffixes and exposes mappings between integers

¹<https://github.com/kpdyer/libfte>

and accepted words. This gives a deterministic bijection between integer values and regex conformant strings of the chosen length.

During encoding, plaintext is first encrypted by the layer described above, resulting in $w1 || w2 || T$. The FTE encoder then packs bytes from the ciphertext into an integer and applies DFA unranking to produce a regex valid string. During decoding, the process is inverted, the fixed regex conformant slice is ranked back into bytes and the ciphertext is reconstructed.

Weighted Encoder

The weighted encoder implements a Huffman coding based transformation from encrypted bytes to lowercase strings. Unlike in conventional Huffman coding used for compression, the encrypted ciphertext is treated as a uniformly distributed bitstream and this bitstream is decoded through a Huffman tree to output characters.

The implementation builds a codebook from the alphabet a-z using the `ENGLISH_FREQ` and `DNS_FREQ` frequency tables. More frequent letters receive shorter bit codes, while less frequent letters receive longer codes. During encoding, ciphertext bits are consumed one at a time by traversing the Huffman tree. While walking the tree, whenever it reaches a leaf node the corresponding character in that node is added to a buffer and the traversal restarts at the root. This causes characters with shorter codes to appear more often in the output.

The decoding path reverses this process. Each received character is mapped back to its assigned bit code using the previously constructed codebook. These bits are then turned back into bytes and interpreted as the ciphertext produced by the encryption layer.

The implementation also provides an estimated capacity function. It estimates the plaintext capacity from the expected number of bits represented per output character. Since the frequency shaped representation is variable length this estimate is an approximation.

5.6 DNS Tunneling

The tunnel is implemented as a staged processing pipeline. At runtime the client performs chunk construction, metadata creation and DNS transport. While the server performs DNS parsing, decoding and decrypting, metadata parsing and file reconstruction.

Client-side processing

The sender reads the full input file as bytes and constructs transfer chunks. Each chunk is represented as `payload || metadata` where metadata is of fixed size and appended to the payload.

After chunk creation, each chunk is encrypted and transformed into a DNS name through either the regex based or the weighted encoding modes. The resulting encoded subdomain is prepended in front of the configured domain to form a routable FQDN. For transmission, one DNS request is generated per encoded chunk. The client can apply a timeout and retry policy per query and tracks the delivery success. Optional delays can also be inserted between queries to reduce the pace of generated network traffic.

Metadata handling

Metadata is 16 bytes long and generated once per chunk to carry values for server side reconstruction:

- version - 1 byte (u8)
- flags - 1 byte (u8)
- payload_len - 2 bytes (u16)
- chunk_index - 4 bytes (u32)
- total_chunks - 4 bytes (u32)
- file_id - 4 bytes (u32)

This allows the server to accept reordering, identify duplicates and detect transfer completion without relying on the chunks arriving in order.

Server-side Processing

The receiver runs a UDP DNS loop and processes each incoming query as follows:

1. Parse the incoming DNS packet and extract the subdomain.
2. Check whether the query belongs to the controlled domain. If not, treat it as unrelated DNS traffic.
3. Take the encoded part of the name, reverse the selected encoding and decrypt, resulting in a raw chunk byte sequence.
4. Split that chunk into payload and metadata, validate all metadata fields and place the payload into the correct file and chunk slot in memory.
5. Return a DNS response code indicating success or error.

Chunks are inserted by `file_id` and `chunk_index` into a reassembly buffer. Duplicate indexes are counted but not reinserted. Completion is triggered when all expected indexes for a transfer are present in the buffer.

Reassembly

When a transfer reaches completeness the chunk payloads are concatenated in index order to reconstruct the original byte stream. The output is written to a file with `.bin` file extension with a name of the following format:

```
recv_<file_id>_<total_chunks>_<timestamp>.bin
```

6 Evaluation

6.1 Benchmarking Setup

The benchmarking harness is implemented as an isolated Docker environment. Thus, all benchmark logic executes inside containers. The current setup uses two containers, one of them being the `tool-server` which is running the DNS tunneling tool in server mode. The other container `orchestrator` executes the client transfer and captures all DNS traffic. A Docker bridge network (`bench_net`) is used for internal communication and a volume (`bench_results_vol`) stores all of the artifacts generated during the benchmark. Each configuration ran 5 times and the results below are the mean values across 5 runs. A test file of 1MiB size was used for the benchmarks. In all the benchmarks the `bench.local` domain was used.

6.2 Metrics

The following metrics were used to benchmark the tool:

- `duration_seconds` - duration of the file transfer
- `packet_count` - total number of packets
- `throughput_bps` - throughput in bits per second
- `entropy` - measured Shannon entropy (bits/character) of domain names

6.3 Results

Example packet captures in the network traffic analyser Wireshark can be seen of the benchmark runs in Appendix Figures A.1 and A.2.

Base32 Encoding

For the sake of benchmarking, a simple Base32 encoder was implemented. The encoder prioritises maximum throughput by taking the encrypted output, encoding it in Base32 and separating the domain name labels with a period (.) ASCII character at every 63 bytes.

The Base32 encoded runs used a chunk size of 115 bytes and a delay of 10 ms between each query. A chunk size of 115 bytes were selected according to an upper bound estimated chunk length resulting from the encoding

$$|S|_{Base32} \approx \left\lceil \frac{5}{8}(252 - d) \right\rceil - 32 = \left\lceil \frac{5}{8}(252 - 11) \right\rceil - 32 \approx 118$$

where $d = 11$ is the length of the domain controlled by the authoritative server and 32 bytes of encryption overhead consisting of the initial value IV and authentication code T .

Key Exchange	Duration (s)	Throughput (bps)	DNS packets	Entropy (bits/char)
none	134.530	62 357	21 186	~4.9489
x25519	134.045	62 580	21 178	~4.9492
p256	134.735	62 264	21 177	~4.9492

Table 6.1: Base32 configuration results

Regex Encoding

The regex runs used a chunk size of 90 bytes and a delay of 10 ms between queries.

Similarly to Base32, the maximum chunk size for regex encoding is derived from an expansion formula that uses the theoretical entropy $\log_2 26$ bits/char introduced in subsection 4.1 for representing the capacity of the chosen regular expression.

$$|S|_{regex} \approx \left\lfloor \frac{\log_2(26)}{8} (252 - d) \right\rfloor - 32 = \left\lfloor \frac{\log_2(26)}{8} (252 - 11) \right\rfloor - 32 \approx 109$$

The theoretical regex capacity estimate gives approximately 109 bytes however, the FTE implementation also stores an additional 16 byte header. Therefore, the benchmark used a conservative 90 byte chunk size.

Key Exchange	Duration (s)	Throughput (bps)	DNS packets	Entropy (bits/char)
none	185.956	45 111	28 342	~4.6739
x25519	188.601	44 482	28 337	~4.6739
p256	186.904	44 881	28 330	~4.6740

Table 6.2: Regex configuration results

Weighted Encoding

Weighted encoded runs used a chunk size of 80 bytes and a delay of 10 ms between each query.

As discussed in Section 4.1, weighted/Huffman based encoding has a probabilistic output length. Thus, the chunk size was selected empirically by generating Huffman encoded subdomains and checking if they fit in the maximum length requirement. For the domain name `bench.local`, the chunk size of 80 bytes resulted in 0 failures in 20000 trials.

Key Exchange	Duration (s)	Throughput (bps)	DNS packets	Entropy (bits/char)
none	207.044	40 516	32 767	~4.1798
x25519	207.188	40 488	32 764	~4.1806
p256	206.839	40 556	32 768	~4.1797

Table 6.3: Weighted configuration results

6.4 Evaluation Summary

The results show that base32 based encoding achieves the best overall transfer performance, followed by regex/FTE based encoding, while weighted/Huffman based encoding is slower.

This can be explained by payload capacity per packet and the resulting number of DNS packets needed for transfer. The performance differences between the key exchange modes (none, x25519 and p256) were small across all three encoding types. This suggests that key exchange introduces little overhead and does not substantially reduce transfer performance. This was expected since the key exchange only affects the first couple packets. Entropy values remained stable within each encoding type and showed no major variation across key exchange configurations. Overall, this evaluation indicates that as expected, the choice of encoding has the strongest impact on performance. Choosing Base32 or regex based encoding results in higher throughput with near theoretical maximum levels of measured entropy, while weighted encoding sacrifices throughput for the sake of stealth against entropy based detection methods, resulting in significantly lower measured Shannon entropy .

7 Conclusion

This thesis presented FTExfil, a proof of concept DNS tunneling tool designed to transfer files through DNS query names while reducing the detectability produced by conventional tunnel encoding. The project investigated whether Format Transforming Encryption and a frequency shaped Huffman encoding could be used to produce DNS compatible query names that avoid some common payload level indicators.

The resulting implementation satisfies the main requirements defined for the project. FTExfil can split an input file into chunks, encrypt and authenticate those chunks, encode the result into query name, transmit the data through DNS queries and reconstruct the original file on the receiving server. The implementation also supports configurable encoding methods and ECDH based key exchange using the curves X25519 or P-256.

Future work could evaluate the tool against real DNS tunnel detection systems. Additional work could also explore more accurate DNS specific frequency models such as evasion of bi- and trigram based detection by constructing additional Huffman trees based on previously emitted characters.

Overall, the project demonstrated that DNS tunneling traffic can be transformed with a novel application of Huffman coding to reduce certain payload level detection features. However, the tool does not eliminate traffic level indicators. Large transfers still require many unique DNS queries to the same controlled domain. Therefore, FTExfil should be understood as a successful proof of concept for encoding based evasion, rather than as a complete solution for avoiding DNS tunnel detection.

A Appendix

No.	Protocol	Length	Info
1	DNS	100	Standard query response 0xb73b A tool-server A 172.18.0.2
2	DNS	73	Standard query response 0x603a AAAA tool-server
3	DNS	148	Standard query 0x5ab3 TXT k1.bab29ceb5c537ef5.x.mcdsz7vt45fgcgwvpgyptj5d6yeaqrucs4zndx2cwjgncxbr2a.bench.local
4	DNS	304	Standard query response 0x5ab3 TXT k1.bab29ceb5c537ef5.x.mcdsz7vt45fgcgwvpgyptj5d6yeaqrucs4zndx2cwjgncxbr2a.bench.local TXT
5	DNS	315	Standard query 0x0001 A adbbkaahkitublpowsobeazvohndtbgae1fcytohetlebufrwefecivdhzscvc.impcgknvxxxzlsdbbvk1fyxijpbgkwohtbhlcgckqvjrrffwfltxtylofwrzsy
6	DNS	584	Standard query response 0x0001 A adbbkaahkitublpowsobeazvohndtbgae1fcytohetlebufrwefecivdhzscvc.impcgknvxxxzlsdbbvk1fyxijpbgkwohtbhlcgckqvjrrffwfltxtylofwrzsy
7	DNS	315	Standard query 0x0002 A aehenfnssihqzfuenshzhvjfwgmblmshpmdqpggtxzhjqqdieglfkhvlye.wiayrjkypegpxuebvkseejajhgvayvzavodhlylmatnmj1xesoiilxqnoha
8	DNS	584	Standard query response 0x0002 A aehenfnssihqzfuenshzhvjfwgmblmshpmdqpggtxzhjqqdieglfkhvlye.wiayrjkypegpxuebvkseejajhgvayvzavodhlylmatnmj1xesoiilxqnoha
9	DNS	315	Standard query 0x0003 A acabyfkuinugvzjfcvqqlwmpzrrmuiprjljumpywztoxiyvhfdhriaq.wykhpnjxvvgafvsvunthxhefjzjvnmkznaiqdbwt1pyetcfztzjybzcy
10	DNS	584	Standard query response 0x0003 A acabyfkuinugvzjfcvqqlwmpzrrmuiprjljumpywztoxiyvhfdhriaq.wykhpnjxvvgafvsvunthxhefjzjvnmkznaiqdbwt1pyetcfztzjybzcy
11	DNS	315	Standard query 0x0004 A afuikgbvvcslrvhdmzokhpxfrsjhbozwosuiqegac1awgnabhw1axvtkrpbg.fycvbhqobdjxryizosgaqymob1smbadvakmlycwa1xdsjgchakuvuqezl
12	DNS	584	Standard query response 0x0004 A afuikgbvvcslrvhdmzokhpxfrsjhbozwosuiqegac1awgnabhw1axvtkrpbg.fycvbhqobdjxryizosgaqymob1smbadvakmlycwa1xdsjgchakuvuqezl
13	DNS	315	Standard query 0x0005 A adoxbierzbtotpilgcnkwdjxeauieyvcvkujg1qcpriucwxyhtgmfvuknsiepwshdwsophupvrj1jiegzhkwjrukezeomj1uwpvcgsszmpzauj1x
14	DNS	584	Standard query response 0x0005 A adoxbierzbtotpilgcnkwdjxeauieyvcvkujg1qcpriucwxyhtgmfvuknsiepwshdwsophupvrj1jiegzhkwjrukezeomj1uwpvcgsszmpzauj1x
15	DNS	315	Standard query 0x0006 A acmhlgujieqmgaeaktwozdzkzmvptwz2cdwraywffjetupsqtdutpbhts.shdgmizdeskhgssyqzdkdeheukiczxrhjbyarmthfj1aloirfctppalrsmjd
16	DNS	584	Standard query response 0x0006 A acmhlgujieqmgaeaktwozdzkzmvptwz2cdwraywffjetupsqtdutpbhts.shdgmizdeskhgssyqzdkdeheukiczxrhjbyarmthfj1aloirfctppalrsmjd
17	DNS	315	Standard query 0x0007 A abkzryvgsjveznapaoyvowpt1bjymldsqzpbewfndvuoichbhvbxieutb.zqcnvbxekzugvfabgmtnu1nsczkbf1fnipwz1fzesbzazary1odntvrsfgt
18	DNS	584	Standard query response 0x0007 A abkzryvgsjveznapaoyvowpt1bjymldsqzpbewfndvuoichbhvbxieutb.zqcnvbxekzugvfabgmtnu1nsczkbf1fnipwz1fzesbzazary1odntvrsfgt
19	DNS	315	Standard query 0x0008 A afygrxqjueavzoxgutf1fdyktaknagfrysbenenjvewtwkzcfowaeylagabw.jfvugssy1iregchexenc1bqadhtncf1fzrphkxcorragucckrg1ahzdcfe
20	DNS	584	Standard query response 0x0008 A afygrxqjueavzoxgutf1fdyktaknagfrysbenenjvewtwkzcfowaeylagabw.jfvugssy1iregchexenc1bqadhtncf1fzrphkxcorragucckrg1ahzdcfe
21	DNS	315	Standard query 0x0009 A aficbvzcdntphsspfk1rxqqlumnyh1vguykiyq1gqxdntiagnr1nqkdaw.noipsufkt1zdwgrzudyawewj1bdghonx1preeauahz1xzjutasescsgyl
22	DNS	584	Standard query response 0x0009 A aficbvzcdntphsspfk1rxqqlumnyh1vguykiyq1gqxdntiagnr1nqkdaw.noipsufkt1zdwgrzudyawewj1bdghonx1preeauahz1xzjutasescsgyl
23	DNS	315	Standard query 0x000a A aaw1lhvtocjhrkdwseinsm1lcmpldsro1faldwgu1lgppzyt1wjd1duroy.wqtsefobrpdoekawz1ieqscotphmdwaesocbpcpsxl1xumbugum1fzleupamol
24	DNS	584	Standard query response 0x000a A aaw1lhvtocjhrkdwseinsm1lcmpldsro1faldwgu1lgppzyt1wjd1duroy.wqtsefobrpdoekawz1ieqscotphmdwaesocbpcpsxl1xumbugum1fzleupamol
25	DNS	315	Standard query 0x000b A afrjbmkgzcfosfuc1fozqkvkpbuar1winharnj1ntdkrhgdkshw1bwlpux.tgfcfnyh1legvt1wjo1yharagcdm1roezdfq1h1gdi1eionncmyob1tnopsaj
26	DNS	584	Standard query response 0x000b A afrjbmkgzcfosfuc1fozqkvkpbuar1winharnj1ntdkrhgdkshw1bwlpux.tgfcfnyh1legvt1wjo1yharagcdm1roezdfq1h1gdi1eionncmyob1tnopsaj
27	DNS	315	Standard query 0x000c A aetculnmwob1cngldkzsswzhkgem1t1ujrydvfyk1jatr1fzulncgofopp.albacebcmwz1rsu1jerypbwkdqysvdk1lpmfosyvdsk1ibpmwzwa1k
28	DNS	584	Standard query response 0x000c A aetculnmwob1cngldkzsswzhkgem1t1ujrydvfyk1jatr1fzulncgofopp.albacebcmwz1rsu1jerypbwkdqysvdk1lpmfosyvdsk1ibpmwzwa1k
29	DNS	315	Standard query 0x000d A acnzmoguhj1gmplnphkwo1zrrmfmp1chay1ir1jkt1vdm1bpx1lkwqotcc1cm1jeo1q1f1j1cy1zdh1kcyut1exheba1rbhd1f1jgesof1bt1uhsnqkngz1hmsi
30	DNS	584	Standard query response 0x000d A acnzmoguhj1gmplnphkwo1zrrmfmp1chay1ir1jkt1vdm1bpx1lkwqotcc1cm1jeo1q1f1j1cy1zdh1kcyut1exheba1rbhd1f1jgesof1bt1uhsnqkngz1hmsi
31	DNS	315	Standard query 0x000e A aejoyakafy1ldpfvsazalahwca1jhfbvuergzrbhtyoksn1faqofqpschf1s1x1sbqaxoy1lwcxfmwzuy1to1apertocs1zdw1qfmg1lctme1f1aics1lha1sc1
32	DNS	584	Standard query response 0x000e A aejoyakafy1ldpfvsazalahwca1jhfbvuergzrbhtyoksn1faqofqpschf1s1x1sbqaxoy1lwcxfmwzuy1to1apertocs1zdw1qfmg1lctme1f1aics1lha1sc1
33	DNS	315	Standard query 0x000f A afbnrfadzyt1fmszhvboqxab1qpsdi1ogpnvndy1ih1jkytxs1x1dal1j1tqm1s1ihc1fgd1ptw1aj1zey1baw1tblbno1cbby1ghbht1vz1y1shjof1wq1d1cl1r1l1kr
34	DNS	584	Standard query response 0x000f A afbnrfadzyt1fmszhvboqxab1qpsdi1ogpnvndy1ih1jkytxs1x1dal1j1tqm1s1ihc1fgd1ptw1aj1zey1baw1tblbno1cbby1ghbht1vz1y1shjof1wq1d1cl1r1l1kr
35	DNS	315	Standard query 0x0010 A adgx1vbf1gnfkyohydrbn1lzeuxn1krzssk1lnqnmzsk1lmkmo1vcgg1ldo.x1p1lwumr1ymgd1kzvd1at1e1kues1yew1cbzj1rfgr1f1rat1bs1bt1v1p1z1ig1n1j
36	DNS	584	Standard query response 0x0010 A adgx1vbf1gnfkyohydrbn1lzeuxn1krzssk1lnqnmzsk1lmkmo1vcgg1ldo.x1p1lwumr1ymgd1kzvd1at1e1kues1yew1cbzj1rfgr1f1rat1bs1bt1v1p1z1ig1n1j
37	DNS	315	Standard query 0x0011 A aaeq1htz1lhmvmvcd1j1ykw1xqnhg1lgtov1rs1n1lqm1c1w1fay1sxu1emkhu.aj1dxt1zcd1k1f1m1w1yo1c1p1d1p1w1b1j1o1u1s1d1k1l1og1w1ad1k1q1h1h1f1n1e1w1d1
38	DNS	584	Standard query response 0x0011 A aaeq1htz1lhmvmvcd1j1ykw1xqnhg1lgtov1rs1n1lqm1c1w1fay1sxu1emkhu.aj1dxt1zcd1k1f1m1w1yo1c1p1d1p1w1b1j1o1u1s1d1k1l1og1w1ad1k1q1h1h1f1n1e1w1d1
39	DNS	315	Standard query 0x0012 A abhh1ev1fxyg1vsc1yhr1xrd1muotr1f1wxi1b1h1sp1q1s1f1c1o1xm1j1n1w1l1b1gu1e.wp1d1xw1b1k1mp1d1y1oz1d1f1kr1j1u1x1o1f1k1w1d1k1w1o1y1e1t1l1q1h1r1y1h1n1v1e1f1l1s1v1
40	DNS	584	Standard query response 0x0012 A abhh1ev1fxyg1vsc1yhr1xrd1muotr1f1wxi1b1h1sp1q1s1f1c1o1xm1j1n1w1l1b1gu1e.wp1d1xw1b1k1mp1d1y1oz1d1f1kr1j1u1x1o1f1k1w1d1k1w1o1y1e1t1l1q1h1r1y1h1n1v1e1f1l1s1v1

Figure A.1: Screen capture of packets generated by regex encoder based file transfer with key exchange on curve x25519.

No.	Protocol	Length	Info
1	DNS	73	Standard query response 0xc5bd AAAA tool-server
2	DNS	100	Standard query response 0x755c A tool-server A 172.18.0.2
3	DNS	148	Standard query 0x9963 TXT k1.f96255c4214bfdf1.x.abjbu4rn1r7qkhezsa415mgzuxwudp7ozr1i4exkd1n3yq4dq.bench.local
4	DNS	304	Standard query response 0x9963 TXT k1.f96255c4214bfdf1.x.abjbu4rn1r7qkhezsa415mgzuxwudp7ozr1i4exkd1n3yq4dq.bench.local TXT
5	DNS	315	Standard query 0x0001 A acfcgzb1v1g1w1x1u1r1j1f1t1e1r1n1f1t1x1g1y1a1k1d1o1h1y1a1z1x1v1j1d1t1y1s1e1s1g1v1a.n1r1e1t1s1x1y1s1t1z1s1t1h1p1u1l1a1e1a1x1b1j1k1t1q1b1t1h1p1z1d1w1b1j1h1d1g1y1a
6	DNS	584	Standard query response 0x0001 A acfcgzb1v1g1w1x1u1r1j1f1t1e1r1n1f1t1x1g1y1a1k1d1o1h1y1a1z1x1v1j1d1t1y1s1e1s1g1v1a.n1r1e1t1s1x1y1s1t1z1s1t1h1p1u1l1a1e1a1x1b1j1k1t1q1b1t1h1p1z1d1w1b1j1h1d1g1y1a
7	DNS	315	Standard query 0x0002 A adbx1csut1srwungnrc1vqm1jah1f1j1y1v1cc1namb1j1ey1r1v1b1j1e1q1t1r1a1v1s1j1q1e1b.tu1k1f1g1t1f1a1y1d1u1x1q1c1y1a1k1o1z1a1h1a1f1j1k1t1w1p1c1t1i1a1s1b1c1t1h1j1h1x1n1z1j1h1
8	DNS	584	Standard query response 0x0002 A adbx1csut1srwungnrc1vqm1jah1f1j1y1v1cc1namb1j1ey1r1v1b1j1e1q1t1r1a1v1s1j1q1e1b.tu1k1f1g1t1f1a1y1d1u1x1q1c1y1a1k1o1z1a1h1a1f1j1k1t1w1p1c1t1i1a1s1b1c1t1h1j1h1x1n1z1j1h1
9	DNS	315	Standard query 0x0003 A acfedkgnxkngatcqvpbuhr1loqr1lmvdmq1veosdpo1incguedfvej1ajddpw.aexf1yohnoxgekcpr1x1z1s1ry1dor1lyw1j1eb1c1i1q1t1z1x1u1p1a1x1r1w1g1z1p1a1z1
10	DNS	584	Standard query response 0x0003 A acfedkgnxkngatcqvpbuhr1loqr1lmvdmq1veosdpo1incguedfvej1ajddpw.aexf1yohnoxgekcpr1x1z1s1ry1dor1lyw1j1eb1c1i1q1t1z1x1u1p1a1x1r1w1g1z1p1a1z1
11	DNS	315	Standard query 0x0004 A actbv1x1j1zd1l1o1r1i1q1b1o1a1y1re1q1e1t1am1h1l1k1v1k1b1p1y1n1g1c1p1h1j1a1k1z1l1a1q1.oq1y1h1f1m1p1u1q1r1s1w1y1d1x1y1h1f1m1l1f1m1a1s1f1c1y1a1h1y1w1b1z1x1p1n1r1f1b1a1y1b1v1n1w1d1
12	DNS	584	Standard query response 0x0004 A actbv1x1j1zd1l1o1r1i1q1b1o1a1y1re1q1e1t1am1h1l1k1v1k1b1p1y1n1g1c1p1h1j1a1k1z1l1a1q1.oq1y1h1f1m1p1u1q1r1s1w1y1d1x1y1h1f1m1l1f1m1a1s1f1c1y1a1h1y1w1b1z1x1p1n1r1f1b1a1y1b1v1n1w1d1
13	DNS	315	Standard query 0x0005 A aavb1q1h1nt1c1k1j1n1a1k1i1w1j1v1t1e1p1g1f1z1d1d1g1t1q1b1y1p1v1n1l1a1p1v1s1g1l1p1y1e1l1w1a.naw1l1q1d1p1om1y1z1u1d1f1v1o1s1x1h1g1e1m1o1y1a1d1e1a1y1l1l1c1u1v1c1r1z1j1p1
14	DNS	584	Standard query response 0x0005 A aavb1q1h1nt1c1k1j1n1a1k1i1w1j1v1t1e1p1g1f1z1d1d1g1t1q1b1y1p1v1n1l1a1p1v1s1g1l1p1y1e1l1w1a.naw1l1q1d1p1om1y1z1u1d1f1v1o1s1x1h1g1e1m1o1y1a1d1e1a1y1l1l1c1u1v1c1r1z1j1p1
15	DNS	315	Standard query 0x0006 A agcb1d1a1c1p1j1h1m1y1o1d1y1ra1l1k1j1p1s1e1z1o1w1l1m1z1u1p1l1f1e1s1t1p1y1m1w1r1f1u1s1j1.p1c1k1z1e1f1f1r1p1w1c1e1v1n1z1b1a1t1s1k1e1w1y1l1z1f1d1b1j1u1g1w1z1n1t1y1j1v1h1d1
16	DNS	584	Standard query response 0x0006 A agcb1d1a1c1p1j1h1m1y1o1d1y1ra1l1k1j1p1s1e1z1o1w1l1m1z1u1p1l1f1e1s1t1p1y1m1w1r1f1u1s1j1.p1c1k1z1e1f1f1r1p1w1c1e1v1n1z1b1a1t1s1k1e1w1y1l1z1f1d1b1j1u1g1w1z1n1t1y1j1v1h1d1
17	DNS	315	Standard query 0x0007 A aby1p1z1n1a1m1f1e1v1n1p1n1x1f1v1d1c1z1l1x1e1s1v1f1q1b1c1q1z1h1v1t1s1q1w1a1b1e1r1m1z1d1x1u1a1n1d1.t1c1y1n1j1x1c1k1f1e1o1j1b1l1p1t1e1m1h1l1n1a1k1h1t1r1f1w1j1n1w1c1m1o1e1z1v1l1q1d1e1r1d1d1k1
18	DNS	584	Standard query response 0x0007 A aby1p1z1n1a1m1f1e1v1n1p1n1x1f1v1d1c1z1l1x1e1s1v1f1q1b1c1q1z1h1v1t1s1q1w1a1b1e1r1m1z1d1x1u1a1n1d1.t1c1y1n1j1x1c1k1f1e1o1j1b1l1p1t1e1m1h1l1n1a1k1h1t1r1f1w1j1n1w1c1m1o1e1z1v1l1q1d1e1r1d1d1k1
19	DNS	315	Standard query 0x0008 A aad1w1p1f1k1z1p1y1a1c1o1e1s1j1f1d1o1f1g1s1c1t1y1u1w1r1j1d1a1m1v1o1k1z1y1o1x1a1q1a1b1u1v1.r1v1j1w1y1t1s1l1m1r1x1o1s1v1m1t1g1a1p1m1k1n1s1b1f1u1z1m1g1c1p1h1o1d1n1q1w1j1y1r1v1y1u1d1k1
20	DNS	584	Standard query response 0x0008 A aad1w1p1f1k1z1p1y1a1c1o1e1s1j1f1d1o1f1g1s1c1t1y1u1w1r1j1d1a1m1v1o1k1z1y1o1x1a1q1a1b1u1v1.r1v1j1w1y1t1s1l1m1r1x1o1s1v1m1t1g1a1p1m1k1n1s1b1f1u1z1m1g1c1p1h1o1d1n1q1w1j1y1r1v1y1u1d1k1
21	DNS	315	Standard query 0x0009 A aew1g1v1p1a1z1d1f1w1k1t1m1f1w1c1v1s1y1f1t1d1c1s1r1x1u1o1w1z1l1b1r1u1c1r1d1f1u1m1y1a1v1u1y1.1y1q1d1b1l1g1u1b1z1a1m1r1h1g1t1o1n1y1z1o1c1n1w1y1c1u1g1m1l1o1w1d1o1i1d1t1y1z1n1p1k1b1k1z1j1y1
22	DNS	584	Standard query response 0x0009 A aew1g1v1p1a1z1d1f1w1k1t1m1f1w1c1v1s1y1f1t1d1c1s1r1x1u1o1w1z1l1b1r1u1c1r1d1f1u1m1y1a1v1u1y1.1y1q1d1b1l1g1u1b1z1a1m1r1h1g1t1o1n1y1z1o1c1n1w1y1c1u1g1m1l1o1w1d1o1i1d1t1y1z1n1p1k1b1k1z1j1y1
23	DNS	315	Standard query 0x000a A ab1v1c1m1p1z1n1r1x1r1v1t1q1d1u1p1z1u1m1d1p1u1z1j1m1y1f1h1c1l1m1a1p1c1m1o1e1n1r1h1u1d1b1l1.p1k1b1w1h1g1d1e1b1o1f1g1r1g1t1b1e1h1f1n1v1b1z1u1p1a1y1a1f1t1u1p1z1y1m1n1j1r1d1a1l1p1c1m1e1k1o1y1y1
24	DNS	584	Standard query response 0x000a A ab1v1c1m1p1z1n1r1x1r1v1t1q1d1u1p1z1u1m1d1p1u1z1j1m1y1f1h1c1l1m1a1p1c1m1o1e1n1r1h1u1d1b1l1.p1k1b1w1h1g1d1e1b1o1f1g1r1g1t1b1e1h1f1n1v1b1z1u1p1a1y1a1f1t1u1p1z1y1m1n1j1r1d1a1l1p1c1m1e1k1o1y1y1
25	DNS	315	Standard query 0x000b A ad1y1n1g1z1u1a1u1c1u1y1e1j1y1k1z1g1h1g1v1r1b1u1c1m1h1f1j1k1s1j1a1d1r1u1e1o1h1p1k1n1z1y1b1t1m1g1.i1p1e1d1l1s1h1f1e1s1c1o1s1e1d1z1y1c1e1o1h1w1k1t1o1p1d1o1f1w1g1a1y1z1l1d1w1d1r1f1o1
26	DNS	584	Standard query response 0x000b A ad1y1n1g1z1u1a1u1c1u1y1e1j1y1k1z1g1h1g1v1r1b1u1c1m1h1f1j1k1s1j1a1d1r1u1e1o1h1p1k1n1z1y1b1t1m1g1.i1p1e1d1l1s1h1f1e1s1c1o1s1e1d1z1y1c1e1o1h1w1k1t1o1p1d1o1f1w1g1a1y1z1l1d1w1d1r1f1o1
27	DNS	315	Standard query 0x000c A a1c1e1s1u1d1z1i1n1t1r1e1d1w1o1j1m1j1o1d1u1w1a1k1a1d1e1w1s1f1q1v1r1e1y1e1r1n1g1o1m1a1l1p1d1v1o1q1.b1h1c1r1b1x1c1l1b1n1a1c1o1d1k1z1a1b1p1f1u1s1u1g1r1a1x1g1y1b1j1m1f1d1d1c1s1p1d1u1
28	DNS	584	Standard query response 0x000c A a1c1e1s1u1d1z1i1n1t1r1e1d1w1o1j1m1j1o1d1u1w1a1k1a1d1e1w1s1f1q1v1r1e1y1e1r1n1g1o1m1a1l1p1d1v1o1q1.b1h1c1r1b1x1c1l1b1n1a1c1o1d1k1z1a1b1p1f1u1s1u1g1r1a1x1g1y1b1j1m1f1d1d1c1s1p1d1u1
29	DNS	315	Standard query 0x000d A af1x1o1m1s1t1n1x1i1y1d1v1t1s1n1f1y1g1a1v1y1s1a1g1o1z1k1u1y1g1p1n1v1m1t1z1t1g1a1h1x1b1h1n1a1z1f1a1.p1r1d1h1n1p1n1z1n1l1o1a1j1i1a1o1k1r1n1m1e1y1z1o1g1h1m1y1k1p1h1t1p1o1n1k1
30	DNS	584	Standard query response 0x000d A af1x1o1m1s1t1n1x1i1y1d1v1t1s1n1f1y1g1a1v1y1s1a1g1o1z1k1u1y1g1p1n1v1m1t1z1t1g1a1h1x1b1h1n1a1z1f1a1.p1r1d1h1n1p1n1z1n1l1o1a1j1i1a1o1k1r1n1m1e1y1z1o1g1h1m1y1k1p1h1t1p1o1n1k1
31	DNS	315	Standard query 0x000e A a1h1b1l1u1a1j1r1q1t1r1n1j1v1g1h1m1e1t1f1c1j1b1z1y1b1f1x1s1g1m1f1p1w1g1v1f1r1k1a1b1o1q1o1r1i1u1a1g1.obb1w1k1a1b1o1c1n1q1w1i1n1t1a1d1c1t1e1z1i1d1g1m1e1a1y1p1h1n1s1o1y1o1q1v1b1x1s1i1y1x1e1l1d1o1
32	DNS	584	Standard query response 0x000e A a1h1b1l1u1a1j1r1q1t1r1n1j1v1g1h1m1e1t1f1c1j1b1z2y1b1f1x1s1g1m1f1p1w1g1v1f1r1k1a1b1o1q1o1r1i1u1a1g1.obb1w1k1a1b1o1c1n1q1w1i1n1t1a1d1c1t1e1z1i1d1g1m1e1a1y1p1h1n1s1o1y1o1q1v1b1x1s1i1y1x1e1l1d1o1
33	DNS	315	Standard query 0x000f A ab1d1v1m1l1u1h1g1v1c1l1t1p1c1v1f1e1a1c1n1c1y1m1d1g1v1z1g1a1a1t1a1v1z1p1o1a1c1e1r1m1.o1g1d1o1b1y1w1g1e1c1t1b1m1h1m1g1b1c1y1l1k1a1k1o1y1l1x1f1n1s1f1j1q1b1k1j1f1d1b1x1d1z1k1b1g1
34	DNS	584	Standard query response 0x000f A ab1d1v1m1l1u1h1g1v1c1l1t1p1c1v1f1e1a1c1n1c1y1m1d1g1v1z1g1a1a1t1a1v1z1p1o1a1c1e1r1m1.o1g1d1o1b1y1w1g1e1c1t1b1m1h1m1g1b1c1y1l1k1a1k1o1y1l1x1f1n1s1f1j1q1b1k1j1f1d1b1x1d1z1k1b1g1
35	DNS	315	Standard query 0x0010 A af1j1j1x1k1v1d1e1t1s1g1t1m1b1j1z1u1o1h1j1z1e1s1p1l1z1r1m1t1k1l1r1u1k1b1w1q1o1s1c1n1i1h1o1e1.unx1k1g1t1d1n1f1v1h1m1r1z1d1p1h1r1a1u1n1y1p1l1c1x1b1h1n1g1e1c1w1a1f1b1y1b1u1
36	DNS	584	Standard query response 0x0010 A af1j1j1x1k1v1d1e1t1s1g1t1m1b1j1z1u1o1h1j1z1e1s1p1l1z1r1m1t1k1l1r1u1k1b1w1q1o1s1c1n1i1h1o1e1.unx1k1g1t1d1n1f1v1h1m1r1z1d1p1h1r1a1u1n1y1p1l1c1x1b1h1n1g1e1c1w1a1f1b1y1b1u1
37	DNS	315	Standard query 0x0011 A a1p1j1p1z1l1y1m1f1c1f1e1n1b1d1r1n1j1z1x1p1y1k1s1m1q1z1f1w1c1r1s1z1s1g1d1a1z1m1c1o1d1s1t1a1q1.c1m1z1v1r1c1o1h1d1v1l1c1p1j1n1f1n1d1o1z1k1y1u1s1i1n1q1e1r1l1o1a1k1d1f1w1e1b1t1a1x1o1v1r1j1
38	DNS	584	Standard query response 0x0011 A a1p1j1p1z1l1y1m1f1c1f1e1n1b1d1r1n1j1z1x1p1y1k1s1m1q1z1f1w1c1r1s1z1s1g1d1a1z1m1c1o1d1s1t1a1q1.c1m1z1v1r1c1o1h1d1v1l1c1p1j1n1f1n1d1o1z1k1y1u1s1i1n1q1e1r1l1o1a1k1d1f1w1e1b1t1a1x1o1v1r1j1
39	DNS	315	Standard query 0x0012 A af1l1f1c1s1z1p1t1k1b1l1t1f1u1t1c1h1w1u1b1d1r1h1n1k1l1p1h1s1z1v1k1h1l1t1q1z1y1f1h1u1z1a1l1m1o1i1.nb1d1g1y1s1q1m1k1f1

Bibliography

- [1] Oscar Bonastre, Andreu Veà, and David Walden. Origins of the domain name system. *IEEE Annals of the History of Computing*, 41:48–60, 04 2019.
- [2] Paul E. Hoffman and Kazunori Fujiwara. DNS Terminology. RFC 9499, March 2024.
- [3] Fabian Strachanski, Denis Petrov, Tobias Schmidbauer, and Steffen Wendzel. A comprehensive pattern-based overview of stegomalware. In *Proceedings of the 19th International Conference on Availability, Reliability and Security*, ARES '24, New York, NY, USA, 2024. Association for Computing Machinery.
- [4] Domain names - implementation and specification. RFC 1035, November 1987.
- [5] Sanjay Adiwai, Balaji Rajendran, Pushparaj Shetty D., and Sithu D. Sudarsan. Dns intrusion detection (did) — a snort-based solution to detect dns amplification and dns tunneling attacks. *Franklin Open*, 2:100010, 2023.
- [6] Greg Farnham. Detecting dns tunneling. Giac gold certification paper, SANS Institute, February 2013. Accepted February 25, 2013; GIAC GCIA Gold Certification.
- [7] Kenton Born and David Gustafson. Detecting DNS tunnels using character frequency analysis. *CoRR*, abs/1004.4358, 2010.
- [8] C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27(3):379–423, 1948.
- [9] Kevin P. Dyer, Scott E. Coull, Thomas Ristenpart, and Thomas Shrimpton. Protocol misidentification made easy with format-transforming encryption. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, page 61–72, New York, NY, USA, 2013. Association for Computing Machinery.
- [10] Jonathan Oakley, Lu Yu, Xingsi Zhong, Ganesh Kumar Venayagamoorthy, and Richard Brooks. Protocol proxy: An fte-based covert channel. *Computers Security*, 92:101777, 2020.
- [11] Simon Josefsson. The Base16, Base32, and Base64 Data Encodings. RFC 4648, October 2006.
- [12] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.
- [13] Oxford College Math Center, Emory University. Letter frequencies in english. <https://mathcenter.oxford.emory.edu/site/math125/englishLetterFreqs/>, n/a.
- [14] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, 2 edition, 1996.
- [15] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

- [16] Daniel Kahn Gillmor. Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS). RFC 7919, August 2016.
- [17] Victor S. Miller. Use of elliptic curves in cryptography. In Hugh C. Williams, editor, *Advances in Cryptology — CRYPTO '85 Proceedings*, pages 417–426, Berlin, Heidelberg, 1986. Springer Berlin Heidelberg.
- [18] Daniel J. Bernstein. Curve25519: New diffie-hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography - PKC 2006*, pages 207–228, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
- [19] Elaine Barker, Lily Chen, Allen Roginsky, Apostol Vassilev, and Richard Davis. Recommendation for pair-wise key-establishment schemes using discrete logarithm cryptography. 2018-04.
- [20] Lily Chen. Recommendation for key derivation using pseudorandom functions. 2024-02-02.
- [21] Hugo Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme. Cryptology ePrint Archive, Paper 2010/264, 2010.
- [22] Hugo Krawczyk and Pasi Eronen. HMAC-based Extract-and-Expand Key Derivation Function (HKDF). RFC 5869, May 2010.
- [23] Morris Dworkin. Recommendation for block cipher modes of operation: Methods and techniques. NIST Special Publication 800-38A, National Institute of Standards and Technology, Gaithersburg, MD, 2001.
- [24] Jérémy Jean. TikZ for Cryptographers. <https://www.iacr.org/authors/tikz/>, February 2015. Figure used: “CTR mode of operation”.
- [25] Quynh Dang. Changes in federal information processing standard (fips) 180-4, secure hash standard. *Cryptologia*, 37(1):69–73, 2013.
- [26] Meltem Sonmez Turan. Keyed-hash message authentication code (hmac):. 2024.