Automatic Structural Inference of Binary Protocols

Fredrik Appelros Carl Ekerot

May 19, 2013

Abstract

Communication protocols are the foundation of everyday networking tasks but they are not always open in terms of available documentation. For reasons of interoperability or the collection of statistics some protocols need to be reverse engineered. This thesis presents a method based on density-based clustering and byte distribution classification for automatically inferring the structure of a protocol from a packet dump. We evaluate the precision of the method for finding different packet types and for finding fields, both globally across all packets and within the inferred packet types.

Contents

Chapter 1

Introduction

1.1 Background

Communication protocols are the foundation of everyday networking tasks. Each time a web page is served or an email is sent, a rigorous series of actions is taken to ensure that the requested piece of information is relayed. The information is packaged in a *message* and both the syntax and the semantics of these messages are defined by protocols.

Protocols may be open or closed in terms of available documentation. The reason for them being closed might be that they are proprietary or simply because no one invested in creating a documentation of the protocol in the first place. Despite the lack of a documentation there still sometimes exists a need to understand the inner workings of a certain protocol, e.g. when providing interoperability between services. One infamous example of this is Microsoft's proprietary SMB protocol which spawned the Samba project. This project was created in an attempt to provide access to certain Windows services for Unix-like operating systems. Tridgell (2003) explains that the Samba project reversed the SMB specification using techniques like *fuzzing*, a process where they inspect the response from a SMB server for a wide variety of generated requests. Using this approach, it took many years until most of the functionality of the SMB protocol had been implemented.

Another area where the need for inferring the structures of closed protocols is in *deep packet inspection* (DPI). Messages sent over a network are usually constructed hierarchically. At the bottom there is only raw binary data and at the top there is the application data, e.g. an email or a video. There in between exists several layers that contain metadata which is used to make sure that the data reaches its target and that it does not get corrupted during transmission. As a message is sent from one party to another it traverses a series of nodes in the network. These nodes only needs to look at the lower parts of the hierarchical structure of the message in order to relay it. If a node wants to conduct DPI however, the higher levels of the hierarchy is looked at as well. To be able to do this the structure of the protocol needs to be known.

There are many different reasons why one would want to perform DPI, one of which is to provide *quality of service* (QoS). QoS is needed in order to gather statistics for Internet service providers that manage the networks. One company

that delivers QoS solutions is Procera Networks on whose behalf we are writing this thesis.

In this thesis we have focused on finding the structure of binary protocols as opposed to textual ones. The reason for this being that textual protocols are intrinsically simple to read and understand and therefore do not require advanced methods of analysis. Some of the methods described here are however also applicable to textual protocols.

1.2 Related work

A popular approach to the problem of automatic inference of network protocols has been to draw a parallel to bioinformatics. Byte-streams have been processed using the same methodology as DNA sequence analysis in order to mine patterns in protocol messages. (Beddoe, 2005) explores the possibilities of using bioinformatics methods such as multiple sequence alignment in order to find patterns in both textual and binary protocols. These principles have been applied in e.g. Netzob[1], a tool for protocol analysis.

Using techniques from bioinformatics in protocol analysis seems natural since the analysis of messages, which may be viewed as the analysis of byte-streams, is easily converted to the problem of sequence analysis. Methods such as multiple sequence alignment does provide an insight to the structure of protocols, but further analysis is needed in order to find semantic meaning in segments of bytes.

Discoverer by Cui et al. is an approach which is specifically modeled after the behavior of network protocols. Discoverer tokenizes network dumps into textual and binary tokens. From the tokens Discoverer attempts to group messages based on their true message type, such as requests or responses. After the grouping of messages, Discoverer seeks semantic meaning in the tokens in common for each group of messages, and classifies the tokens into a set of predefined classes.

Caballero and Song (2012) proposes a method where protocol specifications are constructed by monitoring the execution state of the client or server software which utilizes the protocol. This approach has the obvious benefit of mapping data stored in memory at one state to the data subsequently present in a protocol message. This requires access to the actual software which generates the messages. In many cases this may limit the analysis to the messages sent by the client. When the server software is unavailable for execution state monitoring, one must resort to other methods in order to further analyze the protocol.

[1] http://www.netzob.org

Chapter 2

Theory

2.1 Features

In machine learning, a feature is a measurable property of the input data. An example of a feature when working with images could be the intensity of light for a certain pixel. Features do not need to directly correspond to a physically measurable property however, instead they can be constructed by transforming other features. These new high-level features can then be used just like any other feature in order to simplify a complex problem.

One common type of complex problem is working with high dimensional data. There are different ways to deal with such data; one of them being to use a solving method that scales linearly. Another way is to reduce the dimensionality of the problem itself.

2.1.1 Principal component analysis

One method to reduce the dimensionality of a problem is to use principal component analysis (PCA). PCA is defined by Jolliffe (1986) and transforms the input data into another dataset where each dimension accounts for as much variance as possible while requiring it to maintain orthogonality to the other dimensions. The new dataset can be constructed to contain fewer dimensions than the original data and will then capture as much variance as possible in the given number of dimensions. PCA can be accomplished through singular value decomposition of a matrix.

2.2 Simple linear regression

In statistics, a response variable Y_i that is linearly dependent on an explanatory variable X_i can be modeled with a simple linear regression model. The relationship between the two variables are then

$$Y_i = \alpha + \beta X_i + \varepsilon_i \tag{2.1}$$

where α and β are the *regression coefficients* and ε_i is the error term. Given a set of data $\{Y_i, X_i\}_1^n$ the regression coefficients can be estimated through linear regression. This process is often called fitting as it can be seen as trying to find

Algorithm 1: RANSAC

Input: D, M, n, δ, γ, k
Output: P_b

$P_b \leftarrow \varnothing$
$\varepsilon_b \leftarrow \infty$
for k **do**
    $I \leftarrow n$ number of randomly chosen samples
    fit M to I
    $I_m \leftarrow$ samples where distance from $M \leq \delta$
    $I \leftarrow I \cup I_m$
    if $|I| \geq \gamma$ **then**
        fit M to I
        $\varepsilon \leftarrow$ deviation from M for all I
        if $\varepsilon < \varepsilon_b$ **then**
            $P_b \leftarrow$ parameters in M
            $\varepsilon_b \leftarrow \varepsilon$
return P_b

the model with the best fit for the data. There are different methods to do this as there is no consistent measure for what is the best fit.

2.2.1 Ordinary least squares

Ordinary least squares (OLS) is a method for estimating the regression coefficients in equation 2.1. It does this by minimizing the sum of squared residuals of the simple linear regression model. This can be formulated as minimizing the following cost function.

$$C(\alpha, \beta) = \sum_{i=1}^{n} (Y_i - \alpha - \beta X_i)^2 \tag{2.2}$$

The result of this is a model that minimizes the residuals globally across all samples. This intrinsically makes the method *non-robust*, i.e. sensitive to outliers.

2.2.2 RANSAC

Random Sample Consensus (RANSAC) (Fischler and Bolles, 1981) is a *robust* method for fitting a dataset to a model and was originally intended for image analysis. The robustness of the method allows it to avoid the influence of outliers in the model. Given a dataset D, a model M, a set of inliers I and two confidence parameters, δ and γ, it works the following way:

1. Fit M to I
2. Find the samples that are within δ distance from M and add those samples to I
3. Continue if $|I|$ is at least γ

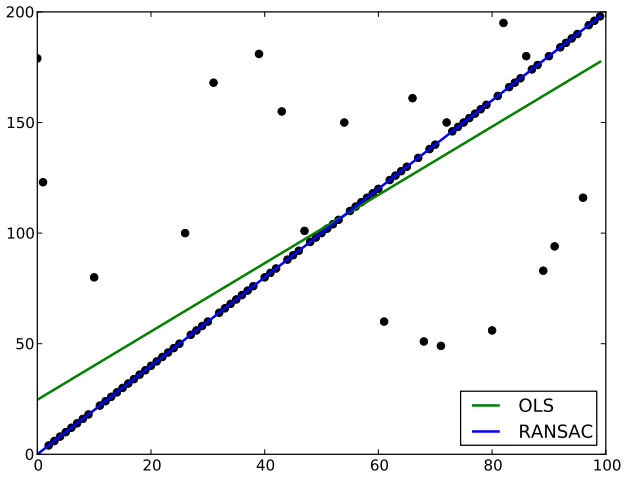


Figure 2.1: A comparison between OLS and RANSAC on a dataset containing outliers.

4. Fit M to I

5. Evaluate M from the residual of I

One problem with this approach is that an initial set of inliers is required. In most cases this is not available so we need to guess which samples might be inliers. A simple solution to this is to take n random samples and assume that they are inliers. This only works if the assumption is actually true and thus the resulting algorithm will be *non-deterministic*, meaning that it only produces a good fit with a certain probability. The algorithm is normally run k number of iterations until the probability of success is sufficiently large. Pseudocode for RANSAC can be seen in algorithm 1.

The model that RANSAC operates on can be any kind of mathematical model, therefore when handling two-dimensional linear data the simple linear regression model given in equation 2.1 can be used. This gives us a method that solves the problem of simple linear regression but that is also insensitive to outliers. A comparison between OLS and RANSAC on a dataset where some samples have been replaced by random noise can be seen in figure 2.1.

2.3 Clustering

A clustering is a grouping of samples based on similarity with respect to their features. The features are predefined to represent distinct properties of the samples. Similarities between samples are normally represented as distances in an n-dimensional space, where n is the number of features. Distances may be calculated using any suitable norm.

In cluster analysis, the goal is normally to find a clustering from a set of samples such that the membership of a cluster represents some true relationship. In

some clustering algorithms such as K-means (MacQueen et al., 1967), the number of clusters are predefined. When the number of clusters are unknown, other algorithms such as DBSCAN, OPTICS and hierarchical clustering methods may be used.

The clustering algorithm we use in our method is OPTICS, which is based on DBSCAN. In the rest of this section we will explain these algorithms and provide examples based on protocol data.

2.3.1 DBSCAN

DBSCAN is a *density-based* clustering algorithm, as opposed to partitioning clustering algorithms such as K-means. Density-based clustering algorithms provides the benefit of not having to provide an estimated number of clusters as a parameter to the algorithm.

DBSCAN was first presented by Ester et al. (1996). The algorithm takes a set of samples D and two parameters, $MinPts$ and ε. $MinPts$ decides how many samples that are needed in order to form a cluster.

The densities are defined by the distances between a set of points. The algorithm introduces the definition *ε-neighborhood* $N_\varepsilon(p)$ for a point p. The samples which are in $N_\varepsilon(p)$ are given by the following condition:

$$N_\varepsilon(p) = \{q \in D \mid dist(p, q) < \varepsilon\} \tag{2.3}$$

The samples in $N_\varepsilon(p)$ are defined as *directly density-reachable* from p. Given that $|N_\varepsilon(p)| \geq MinPts$, the samples in $N_\varepsilon(p)$ forms a cluster, and p is a *core point*.

A cluster is not limited to containing samples which are directly density-reachable. The DBSCAN algorithm also defines that samples which are not directly density-reachable may be *density-reachable*. A sample q is density-reachable from a sample p if there is a set of samples $S = \{s_1, .., s_n\}$ where $p = s_1$, $q = s_n$ and s_{i+1} is directly density-reachable from s_i for $0 < i < n$.

Since the density-reachable relationship is not symmetric, a looser relationship is introduced: *density-connected*. Two samples p and q are density-connected if there exists a third point o from which both p and q are density-reachable. The density-connected relationship is symmetric.

With these relationships, the algorithm defines cluster membership as follows:

Definition. *The following conditions needs to be satisfied for a point q to be a member of a cluster C, given that there is a sample $p \in C$:*

1. *q is density-reachable from p*
2. *q is density-connected to p*

One of the drawbacks of DBSCAN is that it can only find clusters with a density higher than the density decided by the ε-parameter. It is also hard to estimate the ε and $MinPts$-parameters for an unknown dataset. This makes DBSCAN suitable for classifying data into known classes, but not as suitable for finding underlying structures in entirely unknown datasets. In figure 2.2 is an example of DBSCAN running on 5000 packets of DNS data. The parameters are manually selected to $\varepsilon = 0.085$ and $MinPts = 200$.

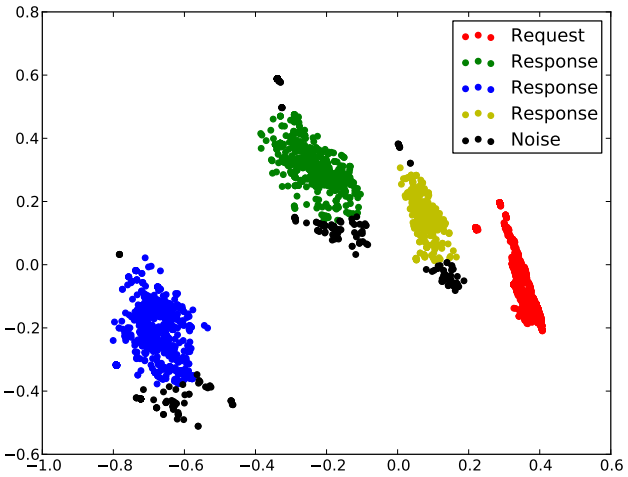


Figure 2.2: An example of clustering with DBSCAN running on samples generated from 5000 packets of DNS data. The samples has been projected down to two dimensions using PCA (see section 2.1.1).

2.3.2 OPTICS

Addressing the difficulties in selecting parameters for DBSCAN, Ankerst et al. (1999) introduced the OPTICS algorithm, which is an extension of the concepts introduced in DBSCAN. OPTICS uses the definitions *directly density-reachable*, *density-reachable* and *density-connected* which were described in the DBSCAN algorithm, but eliminates the need for an explicit ε parameter. Instead the ε parameter is interpreted as the largest distance to consider when clustering. The results is an algorithm which is less sensitive to user-specified parameters, and is able to find clusters with varying densities.

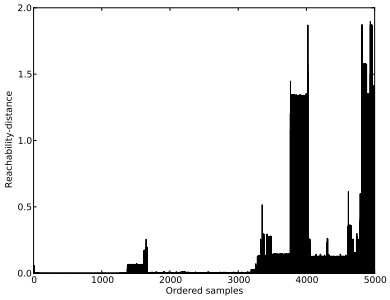
One important aspect of OPTICS is that it does not generate actual clusters. Instead, it generates an *ordering* of samples and a set of corresponding *reachability-distances* which reveals density-based structure in the input data. The OPTICS algorithm extends DBSCAN with the definitions *core-distance* and *reachability-distance*.

Core-distance is a measurement of the distance ε which is required for a sample p to be a core point. That is, a core distance is the distance ε which satisfies $|N_\varepsilon(p)| = MinPts$.

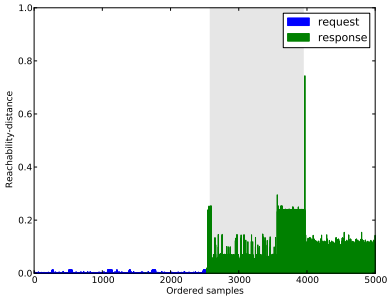
Definition. *The core-distance for a sample p is defined as:*

$$\begin{cases} UNDEFINED, & \text{if } |N_\varepsilon(p)| < MinPts \\ Min\ \varepsilon\ \text{which satisfies } |N_\varepsilon(p)| = MinPts(p), & \text{otherwise} \end{cases}$$

The core-distance is $UNDEFINED$ when an upper limit on ε is given as a parameter to the algorithm. This parameter is not required, although it has an impact on the runtime of the algorithm.



(a) Reachability-plot for 5000 DHCP-packets with $MinPts = 200$.



(b) Hierarchical extraction for 5000 DNS-packets with $MinPts = 200$.

Figure 2.3: Reachability-plots generated from the ordering and reachability-distances given by OPTICS.

The reachability-distance of a sample p is the smallest distance which is needed for p to be directly density-reachable to a core point q for some ε.

Definition. *The reachability-distance of a sample p with respect to some sample q is defined as:*

$$\begin{cases} \textit{UNDEFINED}, & \textit{if } |N_\varepsilon(q)| < MinPts \\ \textit{max(core-distance(q), distance(q, p))}, & \textit{otherwise} \end{cases}$$

where distance(q, p) is the smallest ε for which q is density-reachable from p.

From the ordering and reachability-distances, the cluster structure may be visualized in a *reachability-plot*. The reachability-plot provides an overview of the output from OPTICS as a bar-plot where the bars represent reachability-distances in the ordering generated by OPTICS.

An example of a reachability plot generated from 5000 DHCP-packets with $MinPts = 500$ is shown in figure 2.3a. Valleys in the reachability-plot represents samples which are close to each other. Spikes represents a large difference in distance between the samples to the left and right of the spike.

The pseudocode algorithm is given in algorithm 2.

Algorithm 2: OPTICS

Input: D, $MinPts$, ε
Output: *ordering*, *reachability-distances*

reachability-distances $\leftarrow \varnothing$
ordering $\leftarrow \varnothing$
seeds $\leftarrow 0, 1, ..., |D|$
$i \leftarrow 0$
while $|seeds| > 1$ **do**
    $p \leftarrow seeds.get(i)$
    $seeds.remove(i)$
    $ordering.add(p)$
    if *core-distance*$_{\varepsilon,MinPts}(p) \leq \varepsilon$ **then**
        $neighbors \leftarrow N_\varepsilon(p)$
        for *each neighbor sample* n **do**
            $rdist \leftarrow$ *reachabilty- distance*$_{\varepsilon,MinPts}(n, p)$
            reachability-distances$[n] \leftarrow rdist$
        $i \leftarrow$ seed with least reachability-distance
    else
        $i \leftarrow seeds.first$
`/* Process last remaining seed */`
$ordering.add(seeds.first)$
reachability-distances$[0] \leftarrow 0$
return *ordering, reachability-distances*

Cluster extraction

Ankerst et al. describes the principles of the OPTICS algorithm as running DBSCAN for an infinite number of distance parameters ε_i in the interval $0 \leq \varepsilon_i \leq \varepsilon$. They also explain that extracting clusters from the OPTICS ordering and reachability-distances using a static reachability-distance threshold ε_i gives a clustering which is roughly equivalent to the clustering obtained when running DBSCAN with the same $MinPts$ and $\varepsilon = \varepsilon_i$.

Sander et al. (2003) describes a hierarchical cluster extraction algorithm for OPTICS. As opposed to the DBSCAN-equivalent extraction approach, hierarchical cluster extraction retains many of the properties which comes with OPTICS, such as finding clusters with varying densities and subclusters nested in larger clusters.

The algorithm takes an ordering and reachability-distances from OPTICS and builds a hierarchical representation of the clustering in the form of a tree, where the leaves are clusters.

The pseudocode for hierarchical extraction is given in algorithm 3. Figure 2.3b is the result of the algorithm running on the same DNS dataset as figure 2.2 with $MinPts = 200$. The algorithm yields three clusters, each cluster distinguished by changing background color. The true types *request* and *response* are colored blue and green respectively in the reachability plot. Note that the algorithm is rather insensitive to the $MinPts$-parameter, and that the true types of DNS may be extended to different record types such as A/AAAA/CNAME-records.

2.3.3 Clustering metrics

When measuring the quality of a clustering from a clustering algorithm, a number of metrics are defined. These metrics reflects the quality of the features as well as the clustering algorithm. The metrics requires labels given from the clustering algorithm as well as a *truth*, i.e. the true classes of the samples. The clustering metrics we use are based on information entropy theory and for their exact definition we refer to the paper by Rosenberg and Hirschberg (2007) where they were originally defined. Below are intuitive descriptions of each metric.

Homogenity score

Homogenity is satisfied when every cluster only contains members of a single true class. When completely satisfied, $h = 1$. If every cluster contains samples from different true classes then $h = 0$. When some clusters only contains members from one true class while other contains mixed true classes then $0 < h < 1$.

Completeness score

Completeness is satisfied when all members of a true class are in the same cluster. When completely satisfied, $c = 1$. If all members of a true class are spread out over different clusters then $c = 0$. When some true classes are spread out while other are contained in single clusters then $0 < c < 1$.

V-Measure score

V-Measure is a weighted harmonic mean of homogenity and completeness, and is defined by Rosenberg and Hirschberg as:

$$V_\beta = \frac{(1+\beta) * h * c}{(\beta * h) + c}$$

We weigh homogenity and completeness equally in our application, thus setting $\beta = 1$.

2.4 Sequence alignment

The problem of aligning sequences with one another originally arose in the area of bioinformatics. There, a need to find similarities in long chains of amino acids existed. One of the first methods that solved this problem was the Needleman-Wunsch algorithm (Needleman and Wunsch, 1970). The algorithm finds the maximal number of matching symbols between two sequences, allowing for gaps to be inserted into either sequence. This is done by tracing a path through an alignment matrix where each element represents a possible matching of symbols between the sequences.

Originally, the path was built only with respect to the number of matching identical symbols and not to the number of mismatches or gaps. This has since been improved upon to include pairwise scores for symbol matching and penalties for gaps. The different scores used when matching symbols is typically given in a similarity matrix. An example of such a matrix for a small alphabet of symbols is seen in figure 2.4a.

	A	C	G	T
A	2	1	0	1
C	1	2	1	-1
G	0	1	2	1
T	1	-1	1	2

(a) The similarity matrix for the alphabet.

		A	T	T	C	G	C	T
	0	-1	-2	-3	-4	-5	-6	-7
A	-1	2	1	0	-1	-2	-3	-4
C	-2	1	1	0	2	1	0	-1
T	-3	0	3	3	2	3	2	2
T	-4	-1	2	5	4	3	2	4
G	-5	-2	1	4	6	6	5	4
C	-6	-3	0	3	6	7	8	7

(b) The alignment matrix for the two sequences $S =$ `ATTCGCT` and $T =$ `ACTTGC`.

Figure 2.4: An example of the Needleman-Wunsch algorithm run on two sequences constructed from the alphabet $\{A, C, G, T\}$ with a gap penalty $d = -1$. The similarity matrix used can be seen in (a) and the resulting alignments is given by the gray elements in (b).

Given two sequences, S and T, of length m and n respectively, a scoring matrix C and a gap penalty d the algorithm does the following:

1. Construct an $(n+1) \times (m+1)$ alignment matrix A
2. Fill the first row of A with gap penalties: $a_{0,j} \leftarrow j \cdot d$ where $0 \leq j \leq m$
3. Fill the first column of A with gap penalties: $a_{i,0} \leftarrow i \cdot d$ where $0 \leq i \leq n$
4. Fill the rest of A by doing one of the following actions for each element $a_{i,j}$:
   - Match: $a_{i,j} \leftarrow a_{i-1,j-1} + C_{S_j,T_i}$
   - Delete: $a_{i,j} \leftarrow a_{i-1,j} + d$
   - Insert: $a_{i,j} \leftarrow a_{i,j-1} + d$

   The action that is chosen is the one that results in the largest score where, in the case of a tie, match is given priority. The priority of delete and insert can be chosen arbitrarily.
5. Backtrack through the matrix from $a_{n,m}$ to build the alignments in reverse order. Choose the path that was used to construct the score at the current element. The algorithm finishes when element $a_{0,0}$ is reached.

An example of applying the algorithm can be seen in table 2.4 and the resulting alignments in figure 2.5.

```
A-TTCGCT
ACTT-GC-
```

Figure 2.5: The resulting alignment of the sequences `ATTCGCT` and `ACTTGC`.

Algorithm 3: Hierarchical Cluster Extraction

Input: *ordering*, *reachability-distances*
Output: *clustering*

$R \leftarrow$ *reachability-distances* arranged according to *ordering*
$n \leftarrow$ number of samples
$L \leftarrow$ indices of local maxima in R
Sort L on $R[L_i]$
$R_{max} \leftarrow R[L.last]$
leaves $\leftarrow \varnothing$
function *cluster_tree(node, parent, L)*:
    if L *is empty* **then**
        leaves.add(*node*)
        return
    $s \leftarrow L.pop()$
    node.split $\leftarrow s$
    $sign\_thres \leftarrow significant\text{-}ratio * R[s]$
    Create two new nodes, $N1$ and $N2$
    $N1.samples \leftarrow$ samples left of s
    $N2.samples \leftarrow$ samples right of s
    $L1 \leftarrow$ local maxima left of s
    $L2 \leftarrow$ local maxima right of s
    if $N1$ *and* $N2$ *has average reachability* $<$ *sign_thres* **then**
        if $|N1| > MinPts$ **then**
            children.add($\{N1, L1\}$)
        if $|N2| > MinPts$ **then**
            children.add($\{N2, L2\}$)
        if *children is empty* **then**
            leaves.add(*node*)
            return
        if $R[s] \approx R[parent.split]$ **then**
            for $\{child, L\}$ *in children* **do**
                parent.children.add(*child*)
            parent.children.remove(*node*)
            $p \leftarrow parent$
        else
            for $\{child, L\}$ *in children* **do**
                node.children.add(*child*)
            $p \leftarrow node$
        for $\{child, L\}$ *in children* **do**
            cluster_tree(child, p, L)
    else
        cluster_tree(node, parent, L)

Create node *root* containing all samples
cluster_tree(root, null, L)

Chapter 3

Method

3.1 Approach

A protocol is generally made up of a predefined set of message types. A message type may define if the message is a request or response. If so, it is fairly easy to determine which parts of the message that defines if it is a request or a response. One may simply look for values in a message that depends on the direction of the message. However, most protocols are not limited to being classified as requests and responses. Protocols often define sets of control message types which changes the state of the protocol. Some protocols does not follow the request/response model, which is the case for many peer-to-peer protocols.

In order to thoroughly analyze a protocol, the first step is essentially to distinguish the different types of the protocol. This is a necessary first step which enables us to infer the different states of the protocol and allows for type-specific field analysis. The type is commonly specified by some flag or a combination of flags. A type flag may be a byte, a bit subset of a byte, or spread out over multiple bytes where each byte may define some subtype to a more general type.

Messages that have the same value in their type flags are likely to have a distinguishable structure which differs from messages with other type flags. Some message types may introduce fields which are not present in other message types, and the fields which are global for the protocol are more likely to be identical within a type. A message that is used for binary data transmission is often longer than a control message.

We group messages that are similar using cluster analysis. We then move on to analyzing the clustered messages in an effort to discover the bytes which we rank the most probable to be responsible for the clustering we have found, and label these as *type distinguishers*, i.e. the bytes that distinguishes a specific type. Once these bytes have been discovered, we may regroup the messages based on the value of that the byte takes.

From this new grouping, we are able to analyze the type specific properties of messages of a certain type. We focus on determining the field structure of each type, and make an effort to determine the semantics of each field. From the byte value distribution at a certain byte range, we try to classify each field as one of our primitive field types: constant, flag, uniform, number, incremental or

length. We do this analysis in global scope, cluster scope, stream and connection scope. We define the global scope to cover the fields that all messages have in common e.g. the protocol header. The cluster scope cover fields that are specific for a type. The connection scope is a subscope to the global and cluster scopes, and cover fields that are constant or incremental within a connection, the same goes for the stream scope but only in one direction.

3.2 Type inference

When inferring the message types, we perform a two-pass clustering. In the first pass, we use a clustering algorithm in order to statistically group messages that are similar according to the features we define. In the second pass, we utilize the assumption that there exists one or more bytes which distinguishes a specific type.

3.2.1 Initial clustering

The first step when performing cluster analysis is to define a set of features. The features which we have chosen are based on the byte value distribution at each offset for every message.

We define a matrix P where each element $P_{i,j}$ is the probability that the byte with the decimal value j is present at the offset i for all packets. Using P we create the feature vector $\{f_1, f_2, ..., f_n\}$ for every message m where m is n bytes long. We define each feature as $f_i = P_{i,m_i}$.

Informally, this means that we create a feature vector consisting of n features for each message. Each element f_i in the feature vector is the probability that the byte value m_i occurs at offset i.

The reason for having probability-based features is that density-based clustering algorithms such as OPTICS rely on distance measures. If we were to take the actual byte values at each offset as features, the difference between two byte values would represent a distance between the samples. In a flag byte this would introduce a great distance between a message when the most significant bit is set and when it is not set. Since nominal features are not an option due to difficulties with high dimensionality, the probability measure is a compromise to having nominal features.

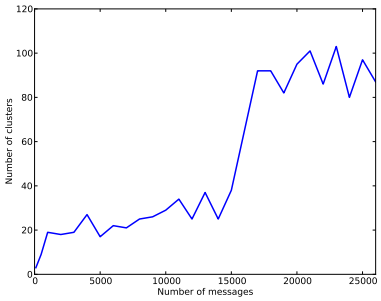
Once the features have been calculated, we apply PCA transformation parametrized to capture 80% of the variance. This reduces the dimensionality of our feature vectors while conserving the most significant variance. We use these new feature vectors as input to OPTICS.

OPTICS works well for our application since it is insensitive to the parameters selected and that it is a suitable clustering algorithm for when the number of classes are unknown. We require an estimate $MinPts$ parameter. The distance parameter has the default value $\varepsilon = \infty$.

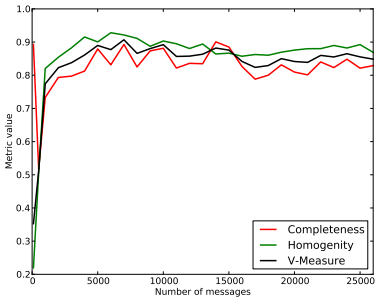
The resulting clustering from OPTICS may have far more clusters than the number of real types in the protocol. This is expected, since messages that are identical on large partitions of the data will induce a low distance when running the OPTICS algorithm, which in its turn will result in high-density areas within the real types. Having more clusters than the number of real types are not a problem however. In the next step of the clustering, we make use of the fact that

the clusters are very homogeneous, i.e. each cluster mainly contains messages of one real type.

Figures 3.1a and 3.1b visualizes how OPTICS performs for various input sizes of SMB messages. The number of messages needed in order to find a good clustering is fairly small. However, a large input size is often needed in order to capture as many message types as possible.



(a) Number of clusters found by OPTICS for different input sizes.



(b) Clustering metrics for our OPTICS clustering for different input sizes.

3.2.2 Type distinguishers

Once we have obtained the OPTICS clustering our goal is to reduce the number of clusters to match the actual number of real types. We do this under the assumption that there is some byte or a number of bytes that define the type of each message. We call these bytes *type distinguishers*.

Many protocols define an explicit type flag in the form of a single byte. The type distinguishers which fit our definition may consist of a number of bytes. When combined, these bytes gives a reasonable indication of what messages with high similarity have in common. In order to find these bytes we use an approach similar to the recursive clustering described by Cui et al. in Discoverer. Our approach is as follows:

1. For every byte offset from the beginning of all messages up to a certain limit we calculate how many values the byte takes. This is equivalent to how many real types that would be present if the chosen byte was a real type flag.

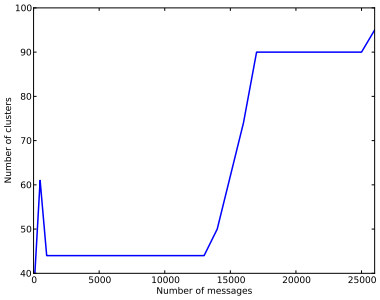
   We introduce two criteria when considering a byte as a type distinguisher. The first criterion is $MaxNumTypes$ which decides the maximum number of types the protocol is assumed to have. The second criterion is $MaxTypeRatio$ which decides the maximum ratio between the number of packets with a certain value for the considered byte and the total number of packets. If $MaxTypeRatio = 0.6$, it prevents a byte from being considered a possible type distinguisher if it assumes the same value in more than 60% of all messages.

2. Now that we have the possible type distinguishers we rank the type distinguishers based on completeness score. For each possible type distinguisher

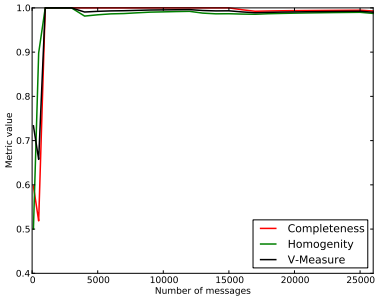
byte we group the messages on every value which the byte takes. This gives us a new clustering. We measure the quality of each type distinguisher byte by comparing its clustering to the OPTICS clustering. This is achieved using completeness score with the OPTICS clustering as our truth.

3. Once our type distinguisher bytes have been scored, we take the bytes with the highest completeness score. For every byte we take, we calculate how many clusters this would yield if we were to cluster all messages on the distinct values that the combination of bytes takes. This process goes on until we reach the point that we can take no more type distinguisher bytes without exceeding $MaxNumTypes$.

In figures 3.1c and 3.1d the performance of our type distinguisher clustering on SMB messages is visualized. This step in our method gives an obvious enhancement to the results from the OPTICS clustering.



(c) Number of clusters found by our type distinguisher clustering for different input sizes.



(d) Clustering metrics for our type distinguisher clustering for different input sizes.

3.3 Field analysis

With our type inference complete we now have a number of clusters that contain similar messages. The next step in inferring the structure of the protocol is to identify field boundaries and their contents. This is a very complex problem as there can be as many fields as there are bits in the messages and the boundaries may lie in between any pair of adjacent bits. The number of possible combinations of fields can therefore easily become too large to handle. Fortunately, most protocols are not designed to utilize each and every bit and instead focus on being extensible, thus there are some common design principles that may be exploited.

3.3.1 Simplifications

First of all we restrict where boundaries may lie. We require fields to be n bytes in length where n is a power of two and also that they must be aligned. We use the definition of *n-byte alignment* which means that the offset at which

		A	T	T	C	G	C	T
	0	-1	-2	-3	-4	-5	-6	-7
A		2	1	0	-1	-2	-3	-4
C			1	0	2	1	0	-1
T				3	2	3	2	2
T					2	3	2	4
G						4	4	3
C							6	5

(e) The alignment matrix for sequences using non-mutual sequence alignment.

```
ATTCGCT
ACTTGC-
```

(f) The resulting alignment of the sequences.

Figure 3.1: An example of our non-mutual sequence alignment algorithm run on the two sequences `ATTCGCT` and `ACTTGC` using the same parameters as in figure 2.4.

the data is located must be a multiple of n. That means that a four byte field may be located at offset 0, 4 or 12 but not at offset 5 or 10. Most protocols follow these principle for optimization reasons as a received packet will reside in a byte buffer in memory and accessing misaligned data can be expensive; see e.g. Assembly/Compiler Coding Rule 46 in Intel® 64 and IA-32 Architectures Optimization Reference Manual for guidelines on memory alignment.

Another principle that is often used is that messages start with a *header*, a collection of fields of fixed length that is located before the data part of the message. Most protocols need a header if they send messages of different types or messages with variable length, or both. In those cases the header will include at least one type flag or length field respectively. A type flag can be a single bit or a collection of bits that indicate what type the current message is so that it can be parsed correctly. A length field is a numerical field that contains metadata on how long a part of the message is, also for parsing purposes. As the header itself usually does not contain any variable length fields no preprocessing is needed to identify them, however for the rest of the message, which is usually type specific, the data needs to be aligned.

3.3.2 Non-mutual sequence alignment

For our purposes we have introduced the term *mutual* sequence alignment to refer to traditional sequence alignment via the Needleman-Wunsch algorithm. Furthermore, we have developed a modified version of the Needleman-Wunsch algorithm that performs *non-mutual* sequence alignment. The difference between the two algorithms is that the non-mutual version only allows gap insertion into one of the two sequences. This restriction only works when the gap insertion is allowed for shorter one of the two sequences; otherwise the algorithm will never be able to reach the top left element during the backtracking. For simplicity let us assume that the longer sequence will always be supplied in S, therefore gaps may only be inserted into T. The non-mutual property is achieved by the following modifications:

1. After the first row has been filled with gap penalties, fill the diagonal with matches: $a_{j,j} \leftarrow a_{j-1,j-1} + C_{S_j,T_j}$ where $1 \leq j \leq n$
2. Do not fill the first column with gap penalties.

3. During the filling phase, only allow the match and insert actions, i.e. remove the delete action, and only fill the upper part of the matrix (above the diagonal).

If we run this algorithm on the same example as in figure 2.4 we get the result shown in figure 3.1..

3.3.3 Preprocessing

Using this method of non-mutual sequence alignment we align each message within each cluster with the longest message in that cluster. This way we get an effective method for doing multiple sequence alignment of all messages in a cluster in a linear number of alignments. To model that variable length data pushes the next field boundary forward we also enforce gap insertion from the right instead of from the left by simply reversing the messages before aligning them and then reversing them back again after the alignment is complete. During the alignment we also use a somewhat complex 256×256 scoring matrix to get a good result. It is constructed to provide the following scores:

- Identical byte values: 2
- Alphanumeric ASCII characters: 2
- Non-alphanumeric ASCII characters: 1
- Numerical distance ≤ 10: 1
- Numerical distance ≤ 20: 0
- Anything else: -1

In addition to aligning the messages for the cluster scope of the analysis we also group messages by stream and connection. These groups are then used when attempting to find fields that are constant in the stream or connection scope. This is accomplished by looking at each message at the IP level and grouping them together depending on their combination of addresses and ports. After this preprocessing of the messages has been done we are now ready to start classifying fields.

3.3.4 Field classification

Now that we have limited our possible field locations and prepared our data for searching in different scopes, we try to classify our data into a predefined set of field types. This is done by iterating through all possible aligned locations of a set of field sizes, typically $\{2^n \mid 0 \leq n \leq 2\}$, and testing if the data at that location may belong to any of the predefined field types. We will then get a number of different possible types for each byte position, possibly of different sizes that might be overlapping. This is our estimate of the field types. To get the most likely candidate we can then give precedence to different sizes and types, favoring larger sizes and more complex types. The field types we have identified are the following:

- Constant

- Flag
- Uniform
- Number
- Incremental
- Length

These field types have been chosen because they are commonly used in the construction of protocols and because they exhibit some degree of identifiability. For the top three types we use simple techniques on the distributions of the individual bytes of all the messages in the current scope while for the bottom three we need to resort to more advanced analytical techniques on the data of each individual message.

3.3.5 Constant fields

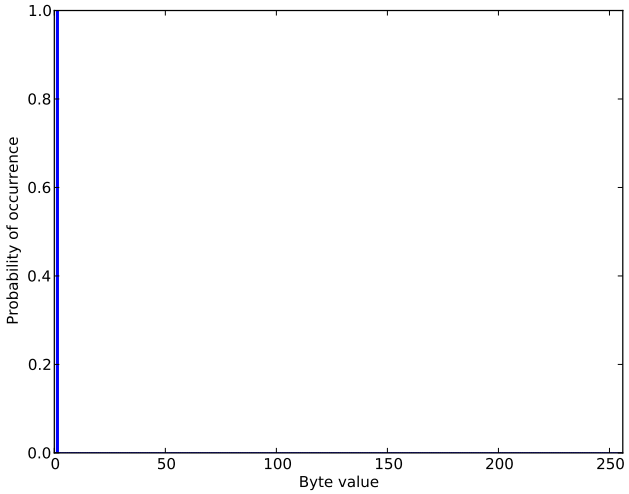


Figure 3.2: Byte value distribution of the 6th byte in the DNS header. This byte is the least significant byte of the two-byte query count field. In our dump its value is constantly one.

We define a constant field as a field that only assumes a single value in all the messages in the current scope. By looking at the byte value probability distributions, we classify an offset i as constant if $P_{i,v} = 1$ holds for some v, $0 \leq v \leq 255$. $P_{i,v}$ is an element in the probability matrix P which is defined in section 3.2.1.

Constant fields may be interpreted as one of the following:

- A true protocol constant. These are fields that are defined to be constant in the protocol specification. An example of such a field is the protocol field in SMB, which is a four-byte constant field with the value `0xff534d42`.

- A reserved field. Reserved fields are specified in case there is need for future additional information in a fixed-sized header. Reserved fields are commonly set to zero.
- A field which is constant in the current scope. For example, if a dump only contains a single connection, then a session ID field may be interpreted as a constant.

3.3.6 Flag fields

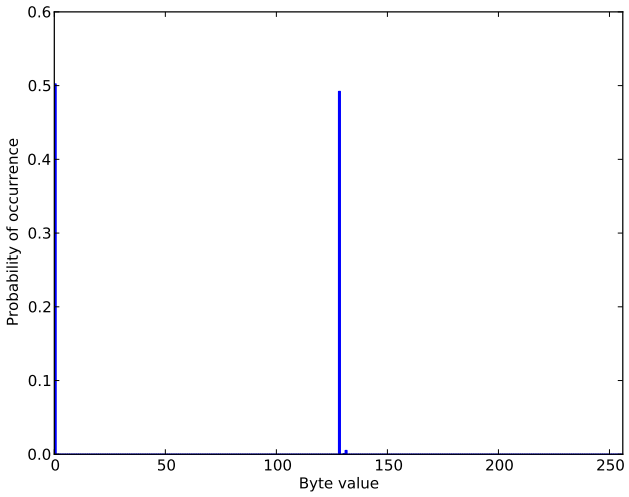


Figure 3.3: Byte value distribution for the 3rd byte in the DNS header. From this byte it may be determined if the message is a query or a response.

A flag field is a field that defines some property of a message. Typically, it assumes a very limited number of values and may be used for representing a message type, an error code, a status code and so on.

Flag fields may be interpreted as real numbers where each number represents a numbered entity, such as a type identifier. It may also represent a set of binary values where each bit is a flag itself, and is interpreted using a bitmask. An example of such a field is the DNS flags field. The flag field in DNS is two bytes long, where the first bit of the field indicates if it is a query (0) or response (1).

The first byte of the DNS flag field is depicted in figure 3.3. The amount of queries and responses are relatively equally distributed. The bar at byte value $0 = \underline{0}0000000$ represent the queries. The bar at byte value $128 = \underline{1}0000000$ represent the responses.

When classifying flag fields, we constrain the amount of values that a certain byte may assume. We classify a field as a flag if the total amount of values the field takes is greater than one single value and less than some upper bound.

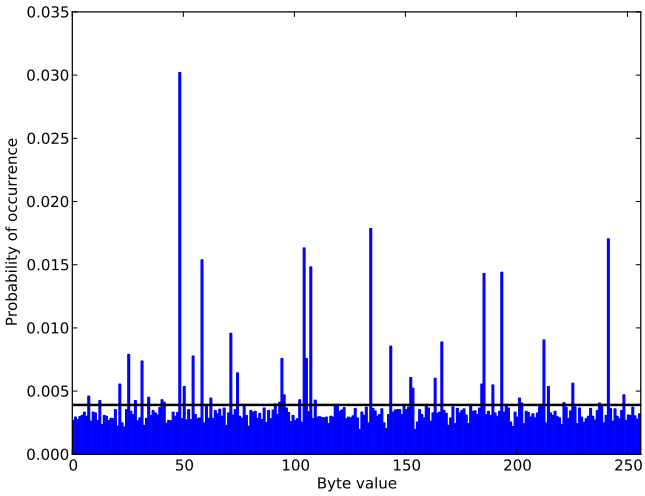


Figure 3.4: Byte value distribution for the first byte of the DNS header. This byte is the first of a two-byte ID field.

3.3.7 Uniform fields

A pattern we observed while inspecting the byte value distribution for various fields was that some fields were uniformly distributed. The real types of these fields are normally some randomly generated value, such as a generated session identifier or a nonce.

When classifying fields as uniform, we compare the distribution of the byte values to the uniform probability distribution for each byte. If we assume the observed byte value is a random variable X and the expected value the scalar x then the uniform probability distribution for a byte is

$$Pr[X = x] = \begin{cases} \frac{1}{256} & 0 \leq x \leq 255 \\ 0 & \text{otherwise} \end{cases}$$

If the sum of the deviations from this distribution is below a certain threshold then the byte is classified as being uniform. If all bytes are classified as uniform then the field is classified as uniform as well.

In figure 3.4 the value distribution for the first byte of the DNS header is displayed. This byte is the first byte of a two-byte identifier. This ID is generated by the client and used by the DNS server in order to identify to which query it issues a response. The black line represents the uniform distribution; a horizontal line at $\frac{1}{256}$.

3.3.8 Number fields

While every field in a binary protocol actually contains numbers, some exhibit a more natural distribution of these numbers than others. We define number fields as those that can be closely fitted to a normal distribution with constraints on the standard deviation. We set these constraints so that the distribution will

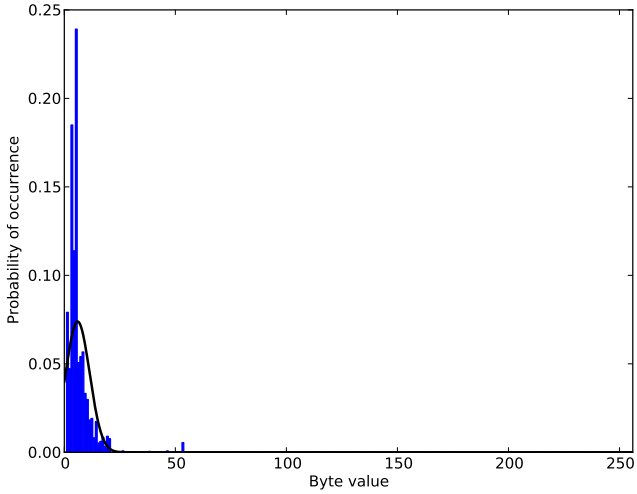


Figure 3.5: A fitted normal distribution to the distribution of values of the 13th byte in DNS.

not collide with our definitions for constant or uniform fields. This captures the behavior of a random variable X with mean μ and standard deviation σ. Typically, values that represent a count behave this way, e.g. the QDCOUNT or ANCOUNT fields of the DNS protocols that refer to how many questions or answers respectively the message contains. This behavior can also been seen in length fields as they represent a byte count. An example of such a byte count can be seen in figure 3.5.

3.3.9 Incremental fields

Some protocols contain fields where the value increases for each message in the scope. It can be a sequence number for each packet or a timestamp of when a certain action was performed. We define these as incremental fields. These can be identified by looking at all the messages within a scope and counting how many of them contain a value that is greater than the previous encountered value. We then compare this count to the total number of messages in the scope and if it is above a certain threshold we accept it as being incremental. The threshold value is used to allow for some amount of wrap-around of the values in the field.

3.3.10 Length fields

Length fields require the most analysis to identify but also gives a very confident estimation because of the depth of the analysis. As mentioned in section 3.3.1, length fields relate to how long a part of the message is. This part can be the entire message, the part following the header or just the next field. For a single message all these types of length fields can be found, as long as there is only one variable length part in the message. For the first case the length of the message

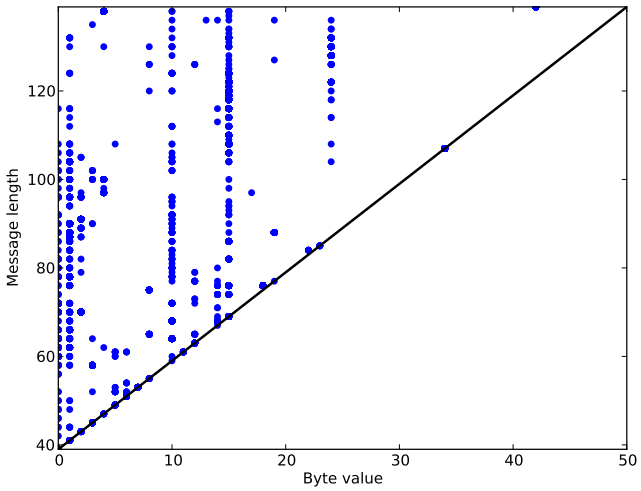


Figure 3.6: RANSAC fit of the WC field in the SMB protocol. The resulting parameters are $\alpha = 39$ and $\beta = 2$.

will be a multiple of the numerical value in the field. Likewise, in the two last cases, the length will be a multiple of the numerical value in the field plus an offset equal to the length of the parts not covered by the field.

Unfortunately, if a message contains multiple parts of variable length the problem becomes more difficult and cannot be solved by looking at individual messages anymore. There is however a better way to approach this problem. We have already seen that the value of the length field multiplied by some scalar plus an offset (which may be zero) is equal to the length of the message. If x is the value of the field, y is the length of the message, α is the offset and β is the multiple then this can be written as

$$y = \alpha + \beta x$$

which means that this can be modeled with a linear model. We can then attempt to solve this with simple linear regression for several messages at once. If the samples are diverse enough there should be a linear relationship between the minimum size of all messages with a specific field value and the value itself. This represents the case of the other variable fields being minimized. The samples that are not of minimal size for the specific field value will then be seen as noise. Because of this noise, which normally is the dominant part of all samples, we need a robust method for fitting our linear model. We therefore use the RANSAC method as described in section 2.2.2 for this purpose. A real world example of this for the SMB protocol can be seen in figure 3.6.

One added advantage of using the RANSAC method is that it also handles other situations where noise appears. This can be if the protocol handles its own fragmentation or some other form of imperfect parsing that affects the length of the messages.

Table 3.1: SMB states.

State	Type
37	Negotiate Protocol Request
15	Negotiate Protocol Response
75	Session Setup AndX Request
48	Session Setup AndX Response
14	Logoff AndX Request
50	Tree Connect AndX Request
20	NT Create AndX Request
24	Delete Request

3.4 Protocol state inference

Many protocols may be described using a state diagram. Some handshake phase between the peers is common when a conversation is initialized. There might also be some teardown phase, and phases for transmitting data etc. Inferring the states of the protocol is an important task when trying to figure out the semantics of the protocol. Modeling the protocol states as a state diagram provides a readable representation of the inner workings of the protocol, and might simplify the analysis of the fields of the different message types.

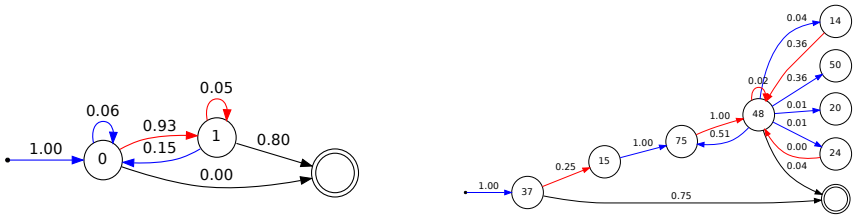
When a reasonable partitioning into types has been achieved, finding the protocol state diagram is trivial. We simply partition the set of messages into connections, where connections are tuples of client and server sockets. From each connection, we define each message in the connection as a change of state. Each state has a set of probabilities for transitioning to all other states we've defined. The cumulative sum of these probabilities sums up to 1.

We represent the states as two $n \times n$-matrices, C and S, where n is the number of types, which will be the same as the number of states. Each entry $C_{i,j}$ or $S_{i,j}$ in the matrix defines the number of times we observe a transition by the client or server from state i to state j. We define the client as the peer initiating the connection. Once we have our two matrices, we may calculate the probabilities for the state transitions by normalizing each row.

A state diagram of all protocol states and transitions may not always be feasible. Some protocols does not follow the state model which we're trying to model with our approach. They may allow some message types to be sent from all states, and may only be represented with a state diagram during some phases such as the handshake phase. We handle this problem by allowing the user to define the depth for which the analysis is performed. If depth is set to 5, we only analyze which states the protocol may be in after five transitions. From this restricted state diagram a handshake procedure will be much more readable.

In figure 3.7a the state diagram generated from a DNS stream capture with $MaxNumTypes = 3$ is shown. State 0 represents queries and state 1 represent responses. Blue edges represent messages sent by the client. Red edges represent messages sent by the server.

Figure 3.7b shows the state diagram for SMB with a depth of five transitions. This shows the handshake used by the protocol. Table 3.1 lists the true types of the messages in a left-to-right order with respect to the state diagram.



(a) State diagram for the DNS protocol. (b) State diagram for the SMB protocol

Figure 3.7: State diagrams generated by our state inference. The left-most dot represents the state before any communication has begun. The right-most double-circle represents the terminal state.

Chapter 4

Results

We measure the performance of our method by comparing the results from the type inference and field inference to annotated data from a variety of network captures for binary protocols.

In common for all the data sets we have analyzed are that they do not explore the entire protocol. If some field in a protocol is constant most of the time, but is set to a certain value in special cases we will have difficulties finding it. Such fields may be flags that denotes some error code or some uncommon state of the protocol. If some type is never present in the dump, we will not be able to predict its existence.

These difficulties emphasize the importance of diverse network captures.

4.1 Datasets

We have chosen five different data sets for our performance analysis. Each data set contains messages from a single protocol. All the protocols vary in complexity, header size and area of usage. The protocols are the following:

- **SMB**: A protocol mainly used by Microsoft Windows for file-sharing, printer-sharing and access control. The protocol has a wide variety of types and a fairly complex header. The protocol messages are generally wrapped in a NetBIOS-message.
- **DNS**: *Domain Name System*, a system for resolving hostnames and addresses to IP-addresses. The DNS header is simple, but its payload contains a variable number of fields. Each of these fields denote a subdomain or domain to some URL.
- **TFTP**: *Trivial File Transfer Protocol*, a very simple file transfer protocol. It defines a very limited number of types and a very simple header.
- **RTP**: *Real-Time Transport Protocol*, a protocol for transporting audio and video. Its header is fairly simple, but in our dump it has no diversity in the payload type. This makes type inference difficult.
- **DHCP**: *Dynamic Host Configuration Protocol*, a protocol for obtaining network configuration information such as IP addresses in a network. The

protocol is an extension to the *Bootstrap Protocol*, BOOTP. Combined with BOOTP, it has a large header and a number of extension headers to BOOTP.

In table 4.1 we list the properties of our different datasets.

Table 4.1: Dataset properties.

Protocol	**# Messages**	**# Connections**	**# Types**
DNS	30628	10935	6
SMB	27906	212	93
TFTP	6492	6	3
RTP	20179	5	1
DHCP	13014	376	11

4.2 Clustering performance

We measure the performance of our clustering using the metrics described in section 2.3.3: homogenity, completeness and V-Measure. The truth we use when calculating the metrics is based on the protocol definition in Wireshark.

In table 4.2 we list the parameters we use on the different data-sets. The initial clustering, which uses OPTICS, uses only the Limit and MinSamples parameters. The type distinguisher clustering uses only the results from the OPTICS clustering and the MaxNumTypes parameter. All of these parameters are rather insensitive to change.

Table 4.2: Clustering parameters.

Protocol	**MinSamples**	**MaxNumTypes**	**Limit**
DNS	200	10	200
SMB	50	100	200
TFTP	50	10	200
RTP	200	10	200
DHCP	50	20	400

Table 4.3 shows the performance of our OPTICS implementation. Like the clustering in Discoverer by Cui et al. we are most concerned with achieving high homogenity. A high homogenity assures that each cluster mainly contains messages from one true type. A high completeness is not as important in our initial clustering. A high completeness gives us a small number of clusters for each true type, ideally a single cluster.

The performance for the RTP and DHCP data sets are inferior to the other protocols. OPTICS has bad performance on RTP since our dump contains messages from a single message type. Instead of clustering on type-specific data, it proceeds to cluster the messages on the protocol payload. Since the payload data is very specific for each connection, each cluster mainly contains messages from a single connection. The DHCP dump is difficult for the clustering due to the size of the header. The BOOTP header is over 200 bytes long. All

DHCP-specific data is located as extensions to BOOTP, and these extensions are located at the end of the header.

Table 4.3: Clustering performance before type distinguisher clustering.

Protocol	**Completeness**	**Homogenity**	**V-measure**
DNS	0.1982	0.9550	0.3282
SMB	0.8480	0.8632	0.8555
TFTP	0.3015	0.9449	0.4571
RTP	0.0000	1.0000	0.0000
DHCP	0.3492	0.1224	0.1813

After the initial clustering, we perform type distinguisher clustering in order to refine the clustering. The results from this process is shown initial table 4.4. Since this method is dependent on some type distinguisher, it does not perform well on the RTP dataset. On the other datasets, the results are close to ideal.

Table 4.4: Clustering performance after type distinguisher clustering.

Protocol	**Completeness**	**Homogenity**	**V-measure**
DNS	0.9534	0.9999	0.9761
SMB	0.9914	0.9880	0.9897
TFTP	1.0000	1.0000	1.0000
RTP	0.0000	1.0000	0.0000
DHCP	1.0000	0.4428	0.6138

In table 4.5 below are the type distinguishers chosen by our clustering and the true type distinguishers for each protocol. Note that we classify both type flags and request/response-flags as type distinguishers, and that the true type distinguishers have byte resolution. The consequence of this is that a byte which contains some bit subset that defines a type are labeled as a true type distinguisher.

Table 4.5: Chosen type distinguishers compared to true type distinguishers.

Protocol	**Chosen bytes**	**Actual bytes**
DNS	2, 3	2, 3
SMB	3, 8	3, 8
TFTP	1	1
RTP	8, 9, 10, 11	1
DHCP	0	0, 242

These results correlate with the results from table 4.4. If the type distinguisher bytes are chosen correctly, the clustering is close to ideal. Failing to find a correct type distinguisher has a heavy impact on the results. This is evident in the results from the RTP data-set. The chosen type distinguishers are actually defined as a stream identifier in the RTP specification. Our data-set contains several streams with variant data for each stream, but only one actual message type.

For DHCP our type distinguisher clustering has found one of the two true type distinguishers. The one it has found is the type byte in the bootstrap protocol BOOTP that wraps the DHCP protocol. The second true type byte is located in an extension header to the BOOTP protocol. Our method fails to find this byte.

4.3 Field inference performance

When measuring the performance of our field inference, we do so by counting the number of bits that match against a definition for the current protocol. To produce a more easily comparable measure we also introduce the term *accuracy* to refer to the percentage of correct bits out of all bits that are being looked at. The definitions for the different protocols were constructed from openly available documentation in RFC 2131, RFC 1350, RFC 1035, RFC 3550 and [MS-SMB]: Server Message Block (SMB) Protocol. We only do these in-depth measurements for the header of each protocol but our analysis yields similar results within clusters as well.

Table 4.6: Symbols used to represent the different field types ordered according to their precedence from highest to lowest.

Symbol	**Description**
C	Global constant
CC	Connection constant
SC	Stream constant
L	Global length
SI	Stream incremental
N	Global number
U	Global uniform
F	Global flag
D	Data
UNK	Unknown

For the definitions we quantified a number of combinations of field types and scopes into symbols. These symbols are given in order of precedence in table 4.6. Worth noting are that the two last field types (data and unknown) are intrinsically non-inferable. The data field type is used in protocol definitions when none of the other field types are applicable. Similarly, the unknown field type is used in our field inference when the analysis cannot reach a conclusion about a type.

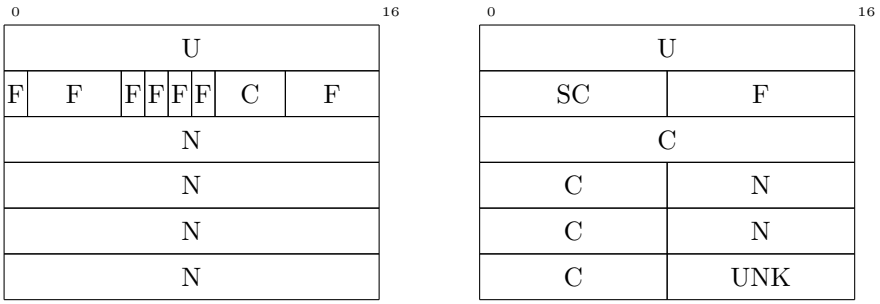
We then annotated each bit of the header with these symbols. The resulting annotations are found in figure 4.1a to figure 4.5a. These annotated definitions of the protocol are then compared, bit by bit, against the result from our field inference algorithm. These resulting inferred protocol definitions are found in figure 4.1b to figure 4.5b.

The resulting number of correctly inferred bits and the corresponding accuracy for the datasets are shown in table 4.7. At a first glance these results seem quite mediocre. This is due to several different factors and we will go through them each on a per dataset basis below.

Table 4.7: The performance of our field inference on the different datasets.

Protocol	# Correct Bits	Accuracy (%)
DNS	37	38.5
SMB	112	38.9
TFTP	0	0.0
RTP	40	41.7
DHCP	64	3.4

4.3.1 DNS performance



(a) Definition of the DNS header.

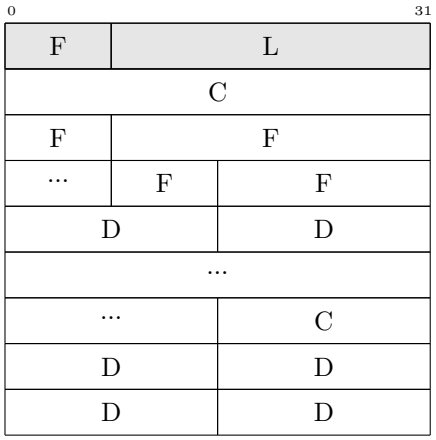
(b) The result from our inference of the DNS header.

Figure 4.1: Comparison of the actual header and the inferred header of DNS.

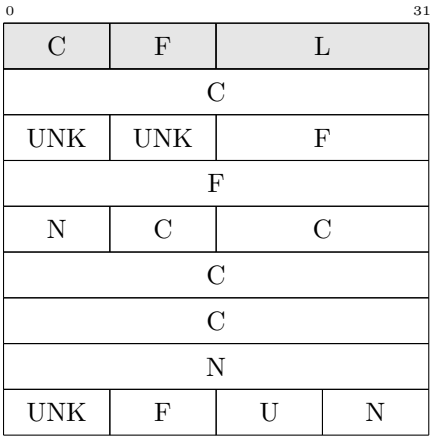
The main reason for the suboptimal performance in the DNS case is the lack of diversity in the packet dump. As we can see in table 4.6 the three types of constant fields has higher precedence than any other type of field. This is the reason for the fields incorrectly classified as C and SC in figure 4.1b as they are constant within their scopes in our dump. Another factor is our precision; we see that the fourth byte has bits of different types, both flag and constant, but our method only looks at them on a byte level. This causes the whole byte to be classified as a flag as the constant bits do not interfere with our classification for flag fields. We can also see that the last byte is not classified as a number because its distribution does not fit well to a normal distribution, again due to low diversity.

4.3.2 SMB performance

Despite similar accuracy, the performance of SMB is actually quite a lot better. The main culprit behind the low performance here lies instead in the protocol definition. The SMB header has a lot of data fields which are, as previously mentioned, non-inferable. This means that actually only 144 of its 288 bits can be found. If we had chosen to define accuracy on these bits instead we would get an accuracy of 77.8%. There is also a slight problem with diversity again, causing some fields to be incorrectly classified as global constant.



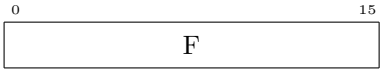
(a) Definition of the SMB header.



(b) The result from our inference of the SMB header.

Figure 4.2: Comparison of the actual header and the inferred header of SMB. Fields which belong to the NetBIOS-header are marked with a gray background. Dots denote continuation of the previous field.

4.3.3 TFTP performance



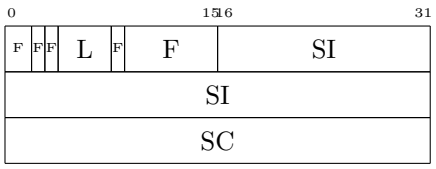
(a) Definition of the TFTP header.

(b) The result from our inference of the TFTP header.

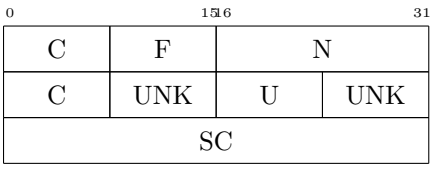
Figure 4.3: Comparison of the actual header and the inferred header of TFTP.

We get the absolute worst performance with TFTP with no correct bits at all. Yet again the problem lies in the diversity of the dump. There are only two bytes in the TFTP header and both are classified as a type of constant. The performance is on the other hand better within the clusters as other fields are found.

4.3.4 RTP performance



(a) Definition of the RTP header.



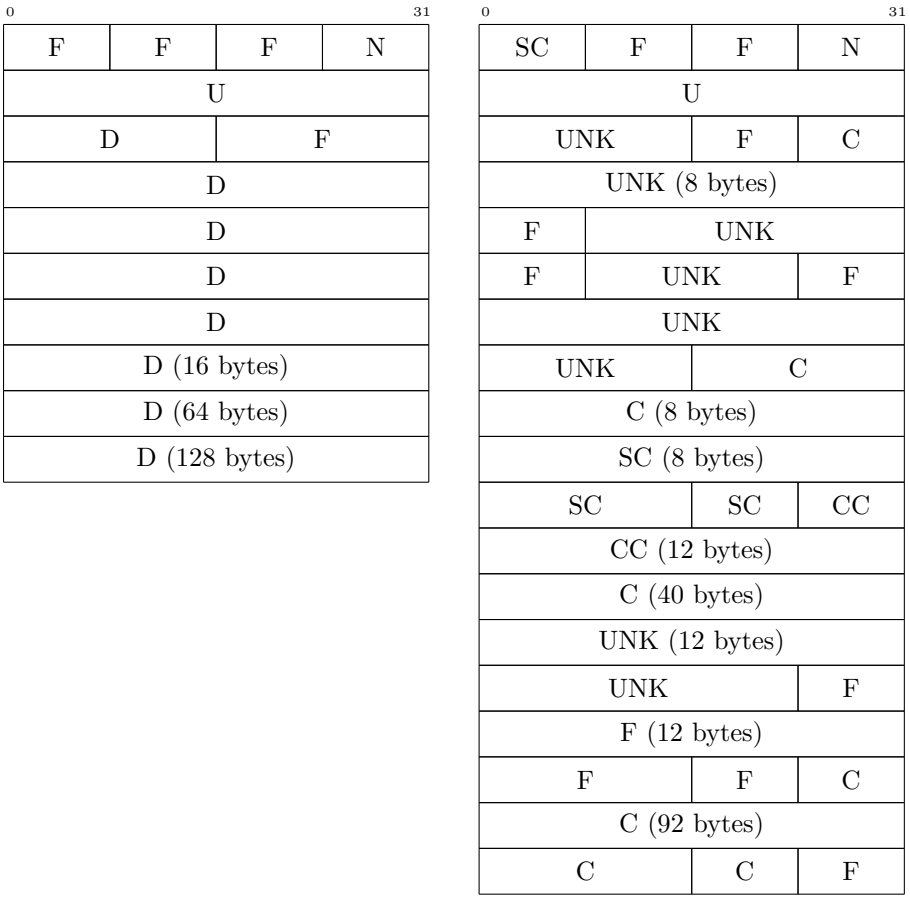
(b) The result from our inference of the RTP header.

Figure 4.4: Comparison of the actual header and the inferred header of RTP.

The best performance for our field inference was found in the RTP dump. When examined more closely it could have been even better as the stream

incremental fields was nearly identified as well. The reason why it was not is that we enforce all streams to be in agreement in order for a field to be classified as something with a stream scope. In this case one of the streams had a lot of retransmitted packets that caused the field to not increase so that it could not be classified as incremental. The condition that all streams need to be in agreement to classify a field might be too strict. A more reasonable approach could be to only require a majority of agreeing streams. Another smaller source of error is that there is yet again some fields that are constant within the dump causing incorrect classifications.

4.3.5 DHCP performance



(a) Definition of the DHCP header.

(b) The result from our inference of the DHCP header.

Figure 4.5: Comparison of the actual header and the inferred header of DHCP. Note that the header is too long to show here in its actual size; the length in bytes for some of the fields are shown in text instead.

The main problem with the classification in the case of DHCP is the extremely large amount of data in the header. Only 80 of the 1888 bits in the header is actually classifiable. If we consider this the accuracy is actually 80.0%.

The fields that are specified as data does not have a clear enough definition in its documentation to fall into one of our categories. They might in reality though have characteristics that does match one of our categories so some of the fields that are classified as data in the definition and something else in our inferred results might actually be correct.

Chapter 5

Discussion

5.1 Conclusions

Draw conclusions about our method and compare it to the related methods.

Is sequence alignment necessary? We could iteratively find length fields to get perfect alignments. Delimiters might be a problem

5.2 Limitations

Give a short introduction to the different limitations of our method.

Mention that the scoring matrix could be improved to use a normal distribution for scoring numerical differences.

Mention the problem with our restricted version of multiple sequence alignment that it the longest message could optimally need gap insertion as well.

5.2.1 Textual protocols

Our method attempts to find structure and semantics in binary protocols. By nature, binary protocols are hard to read and it may be impossible to uncover the exact meaning of every field. Textual protocols, on the other hand, are often self-explanatory. Often they are designed to be read by humans. Textual protocols work well in applications which have no constraint on message size or parsing performance. Notations commonly used in textual protocols such as JSON and XML are more complex to parse than binary protocols, and introduces features where our approach is not feasible.

In Discoverer by Cui et al. an attempt to infer the structure of the HTTP protocol was made. The HTTP header contains a set of unordered key-value pairs delimited by line breaks. Due to the pairs being unordered, the method yielded an immense number of possible types.

For the type inference, our method could be able to find some type distinguisher, perhaps the first three ASCII characters of each HTTP method. Our field inference would however not work, since we're assuming aligned data in a specific ordering.

5.2.2 Variable number of fields

A variable number of fields may be useful in a protocol specification. BOOTP, which is the parent protocol to DHCP, has a variable number of header extensions. DNS has a variable number of queries or answers. How the existence of a variable number of fields is specified is protocol specific, thus hard to determine analytically.

There are byte patterns which could be analyzed in order to infer optional fields, such as length fields or some byte value that decides the number of extended fields. This type of analysis has not been taken into account as we deem it out of the scope for this theses.

5.2.3 Non-aligned data

In our approach, we assume that each field is aligned to one, two, or four bytes. This is somewhat of a standard as explained in section 3.3.1. Although there are obvious benefits to aligning the fields in a protocol, not every protocol follows the field alignment concept. SMB is one of those protocols.

If some field is six bytes long, our method will at best detect it as a two-byte field and a four-byte field. If we were to look for fields with all possible alignments, we would have to use a sliding window approach where we would have to test a six-byte window for every possible offset in order to find a six-byte field. This would increase the complexity of the field inference immensely, since we would have to try $length - n$ possible fields for every length n. With our approach, we try at most $\lfloor \frac{length}{n} \rfloor$ possible fields for every length n.

5.2.4 Bit precision

It's not uncommon for fields to have a higher resolution than one byte. In the RTP header visualized in figure 4.4a there is a four-byte length field between flag-bits in the first byte. In order to find these features, we would have to introduce bit precision in our analysis. Bit precision analysis would introduce the same complexity issues as non-aligned data analysis.

5.3 Future work

Mention the ideas that has come up during the thesis work that we have not had time to investigate further.

5.3.1 Correspondence analysis

In section 2.1.1 we describe how we use PCA in order to reduce the dimensionality of our features while preserving the components that contribute with the most variance. Another method which falls under the same category is correspondence analysis. Correspondence analysis is very similar to PCA, but it is applied to nominal data instead of continous data.

Due to the lack of good correspondence analysis libraries we have been unable to evaluate its performance on our feature vectors. By definition it should be more suitable than PCA in our field of application.

5.3.2 Machine learning for field inference

Our methods use statistical heuristics when inferring fields. Each heuristic is designed for a certain type of field class based on the characteristics we have seen from its byte value distribution. A possible alternative to these heuristics would be to use classification concepts from machine learning such as linear regression or support vector machines.

In our research we considered using these methods when classifying fields. The main complication with this approach was the difficulty in building the training set. Each field in a protocol yields a single sample. In order to build a good training set we would have to annotate and process a large amount of binary protocols. Using these techniques could possibly lead to more robust field classification. Building the training set however is out of the scope for this thesis.

5.3.3 Timestamp identification

From our analysis, we've seen that timestamps have a potentially distinguishable byte distribution. This distribution could perhaps be analyzed in order to classify a field as a timestamp. The distribution reveals a number of spikes we assume to be dependent on the time resolution. This is most visible in the least significant byte of the timestamp field. Using frequency analysis, it should be possible to determine the time resolution. This could be applicable for media streaming protocols.

For general timestamps, we could compare the packet delivery timestamp from the data-set with the value difference in a field. If this difference is consistent enough, we could classify the field as a timestamp.

Chapter 6

Bibliography

Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. OPTICS: ordering points to identify the clustering structure. *ACM SIGMOD Record*, 28(2):49–60, 1999.

Marshall A. Beddoe. Network protocol analysis using bioinformatics algorithms, 2005.

Juan Caballero and Dawn Song. Automatic Protocol Reverse-Engineering: Message Format Extraction and Field Semantics Inference. *Computer Networks*, 2012.

Intel Corporation. Intel® 64 and IA-32 Architectures Optimization Reference Manual, April 2012. URL `http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html`.

Microsoft Corporation. [MS-SMB]: Server Message Block (SMB) Protocol, January 2013. URL `http://msdn.microsoft.com/en-us/library/cc246231.aspx`.

Weidong Cui, Jayanthkumar Kannan, and Helen J. Wang. Discoverer: Automatic Protocol Reverse Engineering from Network Traces. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, pages 1–14, 2007.

Ralph Droms. RFC 2131, March 1997. URL `http://www.ietf.org/rfc/rfc2131.txt`.

Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. University of Munich, München, Germany, 1996.

Martin A. Fischler and Robert C. Bolles. Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography. *Communications of the ACM*, 24(6):381–395, 1981.

Ian T. Jolliffe. *Principal component analysis*, volume 487. Springer-Verlag New York, 1986.

James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, pages 281–297. University of California, Los Angeles, USA, 1967.

Paul V. Mockapetris. RFC 1035, November 1987. URL `http://www.ietf.org/rfc/rfc1035.txt`.

Saul B. Needleman and Christian D. Wunsch. A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970.

Andrew Rosenberg and Julia Hirschberg. V-measure: A conditional entropy-based external cluster evaluation measure. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, volume 410, page 420, 2007.

Jörg Sander, Xuejie Qin, Zhiyong Lu, Nan Niu, and Alex Kovarsky. Automatic Extraction of Clusters from Hierarchical Clustering Representations1. In *Advances in Knowledge Discovery and Data Mining: 7th Pacific-Asia Conference, PAKDD 2003. Seoul, Korea, April 30-May 2, 2003, Proceedings*, volume 7, page 75. Springer, 2003.

Henning Schulzrinne, Steve Casner, Ron Frederick, and Van Jacobson. RFC 3550, July 2003. URL `http://www.ietf.org/rfc/rfc3550.txt`.

Karen R. Sollins. RFC 1350, July 1992. URL `http://www.ietf.org/rfc/rfc1350.txt`.

Andrew Tridgell. How Samba was written, August 2003. URL `http://www.samba.org/ftp/tridge/misc/french_cafe.txt`.