Institute of Technology Blanchardstown

Masters Thesis

Gisele - Automated Data Model and State Machine Inference of Previously Unobserved Network Protocols

Author:
Hugh Pearse

Supervisor:
Arnold Hensman

A thesis submitted in fulfilment of the requirements for the degree of Master of Science in Computing

in the

Institute of Technology Blanchardstown
School of Informatics and Engineering

May 2014

Declaration of Authorship

I, Hugh Pearse, declare that this thesis titled, 'Gisele - Automated Data Model and State Machine Inference of Previously Unobserved Network Protocols' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a degree at this Institute.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: ____________________________________

Date: ____________________________________

“Most hackers are young because young people tend to be adaptable. As long as you remain adaptable, you can always be a good hacker”

Emmanuel Goldstein, Dear Hacker: Letters to the Editor of 2600

INSTITUTE OF TECHNOLOGY BLANCHARDSTOWN

Abstract

Department of Informatics
School of Informatics and Engineering

Master of Science in Computing

Gisele - Automated Data Model and State Machine Inference of Previously Unobserved Network Protocols

by Hugh Pearse

Network protocol reverse engineering is a time consuming and tedious task. Protocol vocabulary and grammar models have applications in malware analysis as well as in software testing. Without documentation it took the open-source SAMBA project 12 years to manually reverse engineer the Microsoft SMB protocol. Previous work in this area has focused on hierarchical clustering based on the seminal work by Marshall Beddoe to infer generic message sequences.

This paper presents a method for clustering network packets using a perfect sequence alignment algorithm and then projecting the packets onto a 2 dimensional space so that it can be clustered using more widely tested clustering algorithms as opposed to hierarchical clustering. By using a 2 dimensional space a quality criterion such as the Calinski-Harabasz criterion can be used to find the optimal number of clusters as opposed to choosing an arbitrary branch length. After clustering is complete a generic message sequence is generated representing all infix delimiters. Prefix and postfix delimiters are then inferred for both intra-packet and inter-packet field relationships. The data model is then output to the Peach Fuzzer XML format. For inferring state transition diagram topologies, some have suggested custom methodologies when existing solutions exist.

Once a network protocol specification has been defined, software implementing the protocol must be tested. A complete suite of fuzzer test cases for a file or network specification can range in the hundreds of millions, leading to test cycles of weeks to months. Test cases with a high risk should have a high probability of finding security vulnerabilities and test cases with a low risk should have a low probability of finding a vulnerability. By attempting to analyse the security risk the test cases pose to the program being tested, low risk test cases can be discarded to shorten the test cycle.

Acknowledgements

I would like to thank my supervisor Arnold Hensman for guiding me while I completed my project.

Contents

List of Figures

List of Tables

Abbreviations

MDS	**M**ulti **D**imensional **S**caling
BIC	**B**ayesian **I**nformation **C**riterion
AIC	**A**kaike **I**nformation **C**riterion
GMS	**G**eneric **M**essage **S**equence
FSM	**F**inite **S**tate **M**achine
HMM	**H**idden **M**arkov **M**odels
MM	**M**arkov **M**odels
LCS	**L**ongest **C**ommon **S**ubstring
SM	**S**tate **M**achine

Chapter 1

Background

1.1 Introduction

Network protocol reverse engineering is a time consuming and tedious task. Protocol vocabulary and grammar models have applications in malware analysis as well as in software testing. Without documentation it took the open-source SAMBA project 12 years to manually reverse engineer the Microsoft SMB protocol. Previous work in this area has focused on hierarchical clustering based on the seminal work by Marshall Beddoe to infer generic message sequences.

This paper presents a method for clustering network packets using a perfect sequence alignment algorithm and then projecting the packets onto a 2 dimensional space so that it can be clustered using more widely tested clustering algorithms as opposed to hierarchical clustering. By using a 2 dimensional space a quality criterion such as the Calinski-Harabasz criterion can be used to find the optimal number of clusters as opposed to choosing an arbitrary branch length. After clustering is complete a generic message sequence is generated representing all infix delimiters. Prefix and postfix delimiters are then inferred for both intra-packet and inter-packet field relationships. The data model is then output to the Peach Fuzzer XML format. For inferring state transition diagram topologies, some have suggested custom methodologies when existing solutions exist.

Once a network protocol specification has been defined, software implementing the protocol must be tested. A complete suite of fuzzer test cases for a file or network specification can range in the hundreds of millions, leading to test cycles of weeks to months. Test cases with a high risk should have a high probability of finding security vulnerabilities and test cases with a low risk should have a low probability of finding a vulnerability. By attempting to analyse the security risk the test cases pose to the program being tested, low risk test cases can be discarded to shorten the test cycle.

1.2 Related Work

Automated protocol reverse engineering projects[4][5][6][7][8][9] can be categorised into three primary categories: Network Based Methods, Program Based Methods and Hybrid Methods[10]. Works that use purley network based methods include PI, ScriptGen, RolePlayer, Discoverer[11], AutoSig, AutoFuzz[12], ReverX[13] and Ptext[14]. These use network packets as the input and reverse engineer a protocol specification from the network data. The advantage of network based methods over program based methods is that the program binary will not always be available to an analyst. A disadvantage of network based methods is that the network traffic can be encrypted which can hinder analysis.

Works that use purely program based methods include Rewards and Tunpi. These use compiled program binaries as an input and use a combination of dynamic or static disassembly and runtime binary instrumentation to tokenize that data being transmitted across the network. Most malware samples use code obfuscation techniques to hinder analysis such as polymorphic encryption algorithms with a decryption stub appended to the end of the file, and metamorphic mutation algorithms to mutate the code of the file every time it is executed or every time it replicates itself. An advantage of program based methods over network based methods is that the data structures used to create the network packet can be helpful in tokenizing the fields in the network traffic.

Works that use hybrid methods compose of a combination of both network based methods and program based methods. Works that use hybrid methods include Polygot, AutoFormat, Prospex, Reformat and Dispatcher. These methods attempt to leverage the advantages of both techniques to create more robust analysis systems.

1.3 Comparing Packets - OSI Layer 7

Data transmitted across the network relies on higher layer protocols to encapsulate the data in order for the devices on the network to know the where to send it. These protocols reduce the complexity for a software developer to transport data from one place to another. However known protocols already have their specification documented and it is trivial enough for someone to implement them. In order to analyse an unknown protocol that relies on TCP/UDP, the higher layers of the network packet can be removed. What is left is an unknown dataset that contains messages destined for a piece of software listening on the network. These messages form the protocol which is to be analyzed. Sets of message types can usually be discerned from one another. If different types of messages are separated from each other into subsets during an analysis, this can reduce noise when deriving statistical information about the subset.

1.4 Topic Modelling

Topic modelling is a process performed on text used to identify the themes of the text by using the word frequency and identifying which part of an ontology the words come from[15]. Topic modelling relies on delimiters to separate words, sentences and paragraphs using characters such as spaces, full stops, semicolons and new lines. When these delimiters are known it is trivial to tokenize the information at varying levels of granularity[16]. However if these delimiters are unknown as they are with network packet data more work is required.

1.4.1 Keywords

One method of tokenizing information with unknown delimiters is to split the data into variable size n-grams. When the data is tokenized into 1-grams, each individual character represents a token and can be compared against an ontology or some other method can be used to identify it as a valid token. When the data is tokenized into 2-grams every character is paired once with the character directly to its left and once with the character directly to its right[17]. These 2-gram tokens can once again be checked to identify any valid ones. When this method is brought from analysing text to analysing network packet data there is no ontology to compare a packets tokens against. However this method can still be used within a subset of network packets to identify tokens that occur frequently.

1.5 Topic Flow

In text analysis transitions between collections of topics can be measured by the overlap between parts of the language ontology. In network packets where there is no ontology to reference, the packets must be categorized into separate types and the order in which the packet types are transmitted represents transitions between states in the application. When using network based methods the application is not monitored making it more difficult to detect different states. This means that Hidden Markov models must be used to model transitions between states which are not known.

1.6 Identifying Fields and Delimiters

Identifying delimiters which specify attributes relating to intra-packet semantics of tokens such as length of a token or number of tokens, and delimiters which specify length attributes relating to inter-packet semantics of packet types transitions such as number of packets of a certain type. In addition to delimiting different aspects of the data there are three basic locations where a delimiter can be placed: Prefix, Infix and Postfix. Prefix notation (also called polish notation) is when a delimiter is placed before a token specifying the length. Infix notation is when the delimiter is placed between tokens and the delimiter is filtered or encoded from the alphabet of the tokens. Lastly postfix notation (also called reverse polish notation) is placed after a token, such as an end of file character which was used to separate files on tape storage.

Chapter 2

Genetic Algorithms

2.1 Perfect Sequence Alignment

Perfect sequence alignment algorithms were originally designed for nucleotide sequences to align two separate sequences of data[18]. The two most well known sequence alignment algorithms are the Needleman-Wunsch algorithm and the Smith-Waterman[19] algorithm, both of which operate in a similar way. These algorithms identify all sub-sequences common to both sequences and aligns the sequences in such a way that they share the maximum possible matching pairs of entries. For sequences which have varying numbers of entries between common sub-sequences, gaps are inserted by the algorithm into the sequence in order to make the sub-sequences align.

		G	E	T		/	i	n	d	e	x	.	h	t	m	l		H	T	T	P	/	1	.	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
G	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
E	0	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
T	0	1	2	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
	0	1	2	3	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
/	0	1	2	3	4	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
	0	1	2	3	4	5	5	5	5	5	5	5	5	5	5	5	6	6	6	6	6	6	6	6	6
H	0	1	2	3	4	5	5	5	5	5	5	5	5	5	5	5	6	7	7	7	7	7	7	7	7
T	0	1	2	3	4	5	5	5	5	5	5	5	5	5	5	5	6	7	8	8	8	8	8	8	8
T	0	1	2	3	4	5	5	5	5	5	5	5	5	5	5	5	6	7	8	9	9	9	9	9	9
P	0	1	2	3	4	5	5	5	5	5	5	5	5	5	5	5	6	7	8	9	A	A	A	A	A
/	0	1	2	3	4	5	5	5	5	5	5	5	5	5	5	5	6	7	8	9	A	B	B	B	B
1	0	1	2	3	4	5	5	5	5	5	5	5	5	5	5	5	6	7	8	9	A	B	C	C	C
.	0	1	2	3	4	5	5	5	5	5	5	6	6	6	6	6	6	7	8	9	A	B	C	D	D
0	0	1	2	3	4	5	5	5	5	5	5	6	6	6	6	6	6	7	8	9	A	B	C	D	E

FIGURE 2.1: Needleman-Wunsch Traceback

The Needleman-Wunsch algorithm consists of three steps: similarity scoring, summing and back-tracing. the matrix is initially filled with a zero value in every cell. Similarity scoring is as simple as iterating across the entire matrix assigning a value of 1 to elements that match and a value of 0 to a mismatch. Summing searches across the matrix for the maximum value in the sub rows and columns to each cell. This maximum value is added to the value of the current cell.

$$M_{ij} = \max \begin{cases} M_{i-1,j-1} + S_{ij} \\ M_{i,j-1} + w \\ M_{i-1,j} + w \end{cases}$$

FIGURE 2.2: Needleman-Wunsch Equation

Back-tracing across to matrix is a process of finding the optimum pathway across the matrix starting from the bottom right hand corner. This can be done by simply checking the values of the cells diagonally to the top left, the cell to the left, and the cell above and moving in the direction of the largest value. Moving directly left or up indicates the insertion of a gap, whereas moving diagonally left and up indicates a match.

2.2 Approximate Matching Distance Metrics

In order to prevent aligning two unrelated sequences, a metric must be used measure the similarity between sequences[20][21][1]. When selecting a metric the one should be aware of the properties of a metric including: symmetry, non-negativity and triangle-inequality. Some metrics for measuring distance between sequences include: Euclidean, Manhattan, Jaccard[22], Cosine, Edit, Hamming, Cavalli-Sforza, Masatoshi Nei, Sanghvi, Rogers, Reynolds, Nei Standard, Nei Minimum, Latter, Goldstein, Slatkin, difference between arithmetic means (used in UPGMA), Shriver distance and Longest Common Subsequence Length[23][24]. Of these distance metrics, some of them satisfy the desirable properties of a metric and some of them do not.

Nei's standard distance is a measure of distance that results in values that are statistically derived from data. Nei's standard distance does not satisfy the symmetry property. This means that if string A is compared to B, and then are compared in the opposite order where B is compared to A, a different distance value will be calculated. This symmetry property is especially important when comparing highly dimensional data such as images or large portions of text. This means that depending on the order that the data is being compared, when the data is being projected into a multi-dimensional space it should be projected relative to the most significant aspect of the data (the principal component)[25][26]. Once the principal components have been identified the frame of reference can be selected to put the data in euclidean space. An example of this would be completing a lower-triangle matrix for every element being compared. Using a symmetric distance metric the upper-triangle and lower-triangle distance matrix contain the same values. These distance matrices facilitate the plotting of the elements into N multi-dimensional space using a multidimensional scaling algorithm. However as Nei's standard distance metric is a-symmetric this means that all distances will be relative to the first point plotted on the N space.

	A	B	C
A	0	1	2
B	3	0	6
C	4	5	0

Table 2.1: Distance Matrix

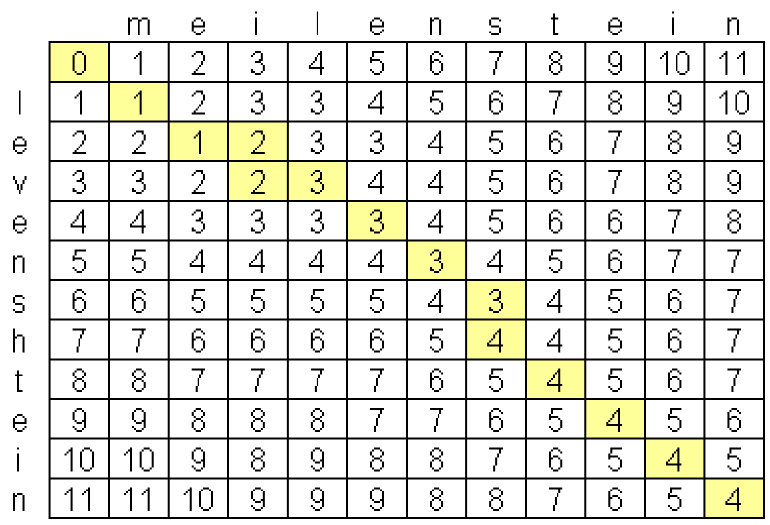
The Smith-Waterman algorithm finds the perfect alignment using a similarity matrix also used in the Needleman-Wunsch algorithm. These algorithms are capable of extracting information such as the number of matching characters in an aligned string, the number of non-matching characters in an aligned string and the length of the longest common substring. The number of matching characters in two aligned sequences satisfies the symmetry property that is desirable for creating similarity matrices for clustering. The number of non-matching characters between two aligned sequences is desirable for populating a distance matrix for multidimensional scaling.

	A	B	C
A	0		
B	3	0	
C	4	5	0

Table 2.2: Lower Triangular Matrix

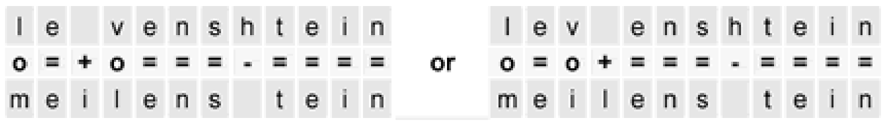
2.3 Levenshtein distance

The Levenshtein algorithm is designed to find the optimal number of edit operations to modify one string to another string[27]. A matrix is populated with the values for the first string arranged across the top and the other across the side. The matrix is then populated with initial values corresponding to its offset. After the matrix is populated the comparison begins by travelling diagonally across the matrix and scoring the fields. Different scores are assigned to different observations. If both corresponding characters are the same in both strings are the same at the same offset, then usually a cost of 0 is assigned to that cell. If the character from one string is incorrect this can be assigned three possible costs: insert, delete or update. Insert and delete both have a cost of 1. Update is considered to be a combination of both delete and insert so it is assigned a cost of 2. Travelling across the matrix diagonally means the characters match. Travelling horizontally or vertically means an insert or delete operation. The following characters signify different operations. "=" Match; "o" Substitution; "+" Insertion; "-" Deletion.



		m	e	i	l	e	n	s	t	e	i	n
	0	1	2	3	4	5	6	7	8	9	10	11
l	1	1	2	3	3	4	5	6	7	8	9	10
e	2	2	1	2	3	3	4	5	6	7	8	9
v	3	3	2	2	3	4	4	5	6	7	8	9
e	4	4	3	3	3	3	4	5	6	6	7	8
n	5	5	4	4	4	4	3	4	5	6	7	7
s	6	6	5	5	5	5	4	3	4	5	6	7
h	7	7	6	6	6	6	5	4	4	5	6	7
t	8	8	7	7	7	7	6	5	4	5	6	7
e	9	9	8	8	8	7	7	6	5	4	5	6
i	10	10	9	8	9	8	8	7	6	5	4	5
n	11	11	10	9	9	9	8	8	7	6	5	4

Figure 2.3: Levenshtein Matrix



l	e		v	e	n	s	h	t	e	i	n		l	e	v		e	n	s	h	t	e	i	n
o	=	+	o	=	=	=	-	=	=	=	=	or	o	=	o	+	=	=	=	-	=	=	=	=
m	e	i	l	e	n	s		t	e	i	n		m	e	i	l	e	n	s		t	e	i	n

Figure 2.4: String Similarities and Differences

2.4 Similarity vs Dissimilarity Metrics

There is an important difference between a measure of similarity and a measure of dissimilarity. Similarity in the Levenshtein algorithm is represented as the number of matching characters in two sequences. Dissimilarity is represented as the sum of the costs applied to the Levenshtein matrix. Similarity and dissimilarity have two distinct applications. Similarity is useful for clustering algorithms, whereas dissimilarity is useful for multidimensional scaling. If two packets are compared and their similarity is 100% then they can be considered to be the same and placed in the same cluster. This implies that their dissimilarity is 0% but this is implicit reasoning instead of explicit. Multidimensional scaling functions are functions which take an input of a distance matrix and return an output of coordinates in N dimensional space. Thus dissimilarity metrics have direct applications for creating relative coordinates and similarity metrics have direct applications in clustering.

Chapter 3

Clustering

3.1 Clustering Algorithms

There are two principal clustering strategies: agglomerative clustering and divisive clustering[28]. Agglomerative clustering is where each data entry is considered to be a unique cluster when the algorithm starts, and for each iteration of analysis clusters are paired and merged recursively. Divisive clustering is where all the elements of the data together are considered as one single cluster from the beginning, then for each iteration of the clustering algorithm the clusters are split recursively. In addition to these categories of algorithms, there are two principle problems commonly faced when using these algorithms: supervised learning and unsupervised learning. Supervised learning is where the classes are known from the beginning. Unsupervised learning is where the classes are unknown from the beginning.

3.1.1 Hierarchical Clustering

Hierarchical clustering algorithms are where all elements are organised into a dendrogram with a root node representing a superclass which encompasses a representation of all classes in the dataset. Each leaf vertex represents a single data element in the dataset and the intermediate graph vertices which lie between the root and the leaves are abstract classes that do not exist in the dataset but represent classes of paired elements. The distance between two elements in the dataset is represented by the branch length which is the value of the graphs edges[29]. Similar vertices have a low value assigned to their edges, while distantly related vertices have high values assigned to their edges.

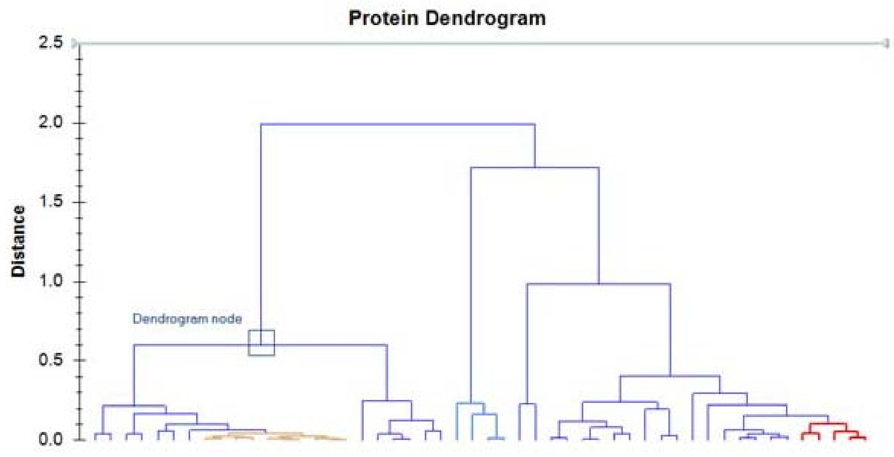


FIGURE 3.1: Dendrogram

3.1.2 Hierarchical Linkage Criteria

Linkage is a process in hierarchical clustering to connect abstract intermediate vertices. The linkage criteria for a hierarchical graph consists of three main distance methods: single linkage (nearest neighbour), complete linkage (furthest neighbour) and average linkage (UPGMA)[30][31]. In single linkage two intermediate vertices are compared by comparing all the leaf nodes that each vertex represents and finding the two leaf nodes with the smallest distance. In complete linkage two intermediate vertices are compared by comparing all the leaf nodes that each vertex represents and finding the two leaf nodes with the largest distance. In average linkage two intermediate vertices are compared by calculating the average value of all the leaf nodes that each vertex represents and calculating the distance between the average value for the intermediate vertices.

3.1.3 Divisive and Agglomerative Hierarchical Clustering

Divisive clustering algorithms differ from agglomerative algorithms in both the steps used to create them and the final output. Divisive algorithms begin with a single superclass and recursively split the clusters. The final graph does not necessarily have a large number of classes. With a hierarchical clustering algorithm that uses a pairwise binary tree for its dendrogram all vertices are connected to the tree even if they are unrelated. This means that with hierarchical clustering, distinct classes must be derived from the dendrograms branch lengths.

3.1.4 Centroid-based clustering

In the case of the k-means centroid-based clustering algorithms the number of classes must be specified from the beginning. When comparing this to a hierarchical clustering algorithm, with k-means the number of classes must be calculated before the clustering begins and with hierarchical clustering the number of classes must be calculated afterwards using the branch lengths.

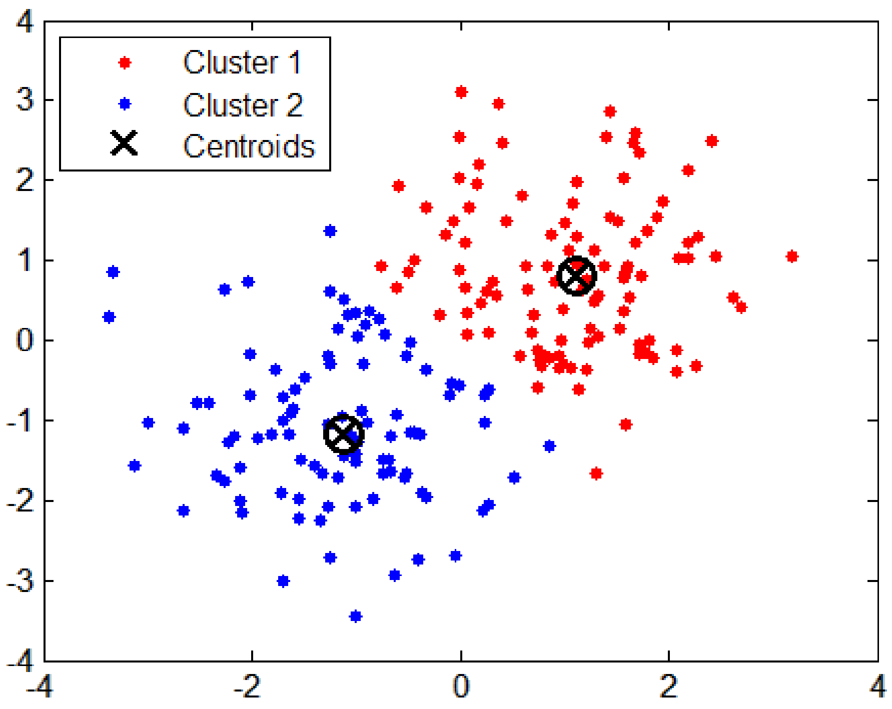


FIGURE 3.2: Centroid Based Clustering

3.1.5 Selecting a Clustering Algorithm

In various comparative studies of the precision, performance and recall of clustering algorithms, the k-means clustering algorithm has proven repeatedly to be the best[32][33]. The problem however is that the k-means algorithm can only cluster data in euclidean space which means the distance metric should be symmetric. Additionally with k-means the number of classes must be known beforehand. K-means performs poorly on high dimensional data. This means that large volumes of text data needs to go through a dimension reduction process before it can be clustered using k-means.

3.2 Multidimensional Scaling

Suppose that you had a difference matrix where each cell represented the distance between two geographic locations such as Moscow, London, Paris and Tokyo. You have the distances between every city but not their geographic coordinates and you want to represent their locations spatially[34]. It is possible to reconstruct a spatial location from non spatial distances. This process is called multidimensional scaling. Multidimensional scaling is means of converting N-dimensional data to any number of dimensions less than N. This is done using a distance matrix for every element in the set[35]. If a dataset of elements are compared using a distance function and their distance values are put into the distance matrix, then the data can be converted into 2-dimensional x and y coordinates placing the first element at point 0,0 and placing all subsequent elements relative to it. It is also possible to perform MDS using asymmetric metrics however graphing this requires adding or subtracting dimensions depending on the data[36][37][38].

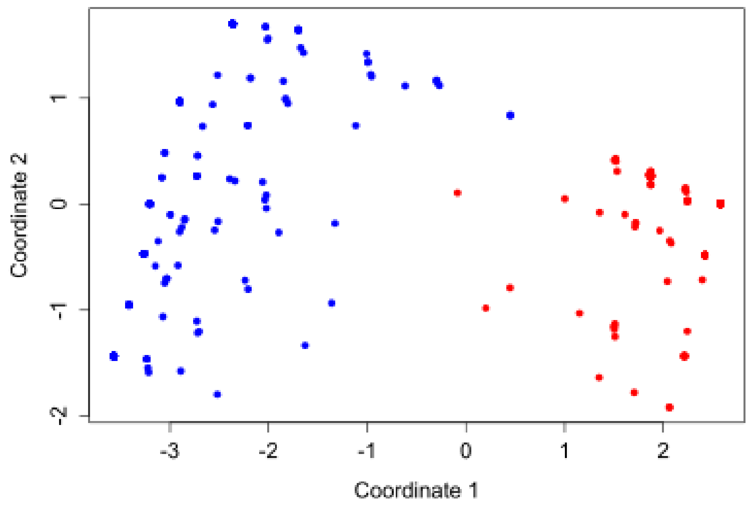


Figure 3.3: Multidimensional Scaling

3.3 Estimating K Clusters Correctly

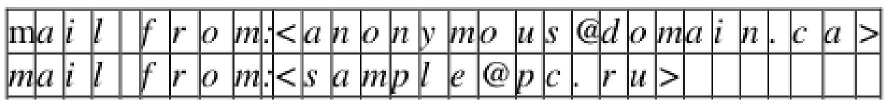
There are a number of algorithms for estimating the correct number of distinct classes in a dataset including the following[39][40][41][42]: Calinski and Harabasz, Je(2)/Je(1), C-Index, Gamma, Beale, Cubic Clustering Criterion, Point-Biserial, Mojena, Davies and Bouldin, Stepsize, Likelihood Ratio, abs(log p), Sneath, Frey and Van Groenewoud, Tau and Bayesian Autoclass. The most modern of these is the Bayesian Autoclass algorithm which has not been subject to any comparative studies unlike the others which have all been comparatively reviewed in terms of precision. The Calinski and Harabasz class estimation algorithm has been found to be the most precise in terms of estimating the correct number of classes for a set of points at varying distances.

Chapter 4

Constructing the Data Model

4.1 Merging Graph Vertices - Generic Message Sequences

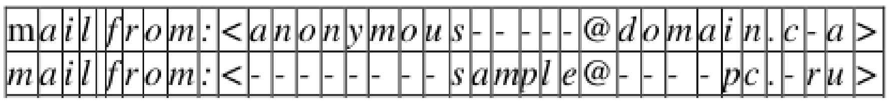
A generic message sequence is an abstracted representation of a collection of vertices, each of which represent an entry in the dataset. Assuming the data has been clustered already the following messages need to be converted into more generalised forms where their variable parts are distinct from their infix delimiters.



m	*a*	*i*	*l*		*f*	*r*	*o*	*m*	:	<	*a*	*n*	*o*	*n*	*y*	*m*	*o*	*u*	*s*	@	*d*	*o*	*m*	*a*	*i*	*n*	.	*c*	*a*	>
m	*a*	*i*	*l*		*f*	*r*	*o*	*m*	:	<	*s*	*a*	*m*	*p*	*l*	*e*	@	*p*	*c*	.	*r*	*u*	>							

Figure 4.1: Two Adjacent Sequences In The Same Cluster

Creating generic messages sequences using a hypercube of Needleman-Wunsch similarity matrices is memory inefficient. Instead of attempting to process every element in the cluster simultaneously the messages in a cluster of message can simply be added to a job queue and compared two at a time. Each message can be compared to its least distant message within the cluster. A generic message sequence can be created from each pairwise similarity matrix and gaps can be inserted to create a generic sequence for that pair. Once message X is compared to another message Y, both messages can be removed from the queue and a GMS can be created for that pair. Once all messages have been removed from the queue, a new queue can be created for the generic message sequences and the process can be repeated on the generic sequences within the cluster.



m	*a*	*i*	*l*		*f*	*r*	*o*	*m*	:	<	*a*	*n*	*o*	*n*	*y*	*m*	*o*	*u*	*s*	-	-	-	-	-	@	*d*	*o*	*m*	*a*	*i*	*n*	.	*c*	-	*a*	>
m	*a*	*i*	*l*		*f*	*r*	*o*	*m*	:	<	-	-	-	-	-	-	-	-	*s*	*a*	*m*	*p*	*l*	*e*	@	-	-	-	-	*p*	*c*	.	-	*r*	*u*	>

Figure 4.2: Aligned Sequences In A Cluster

Once all generic message sequences within a cluster have been merged recursively, a single generic message sequence will remain and should represent the format of all the messages within that cluster.

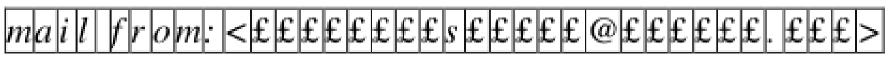


Figure 4.3: Generic Message Sequence For A Cluster

4.2 Measuring Inter Cluster Distance and Merging

The most abstracted generic message sequences representing distinct clusters may be similar when compared to the generic message sequences of other clusters on the same graph. This could occur if the distance metric used to compare the sequences is influenced by substrings in individual elements of the data which could be common to several elements but does affect the structure of the data such as a string "create(car, truck)" and "create(bat, lion)". Both of these share the substring "create" which defines their functionality but may be accidentally clustered separately as their parameters differ. This problem can be solved by overestimating too many classes initially and merging duplicate clusters once the generic message sequences have been created. In the opposite case where the number of classes has been underestimated, this may lead to strings such as "create(car, truck)" and "destroy(car, truck)" being clustered into the same class as they share a common substring "car, truck".

If a cluster has a tree structure of generic message sequences, then clusters can be compared at various levels of granularity. The most abstracted root of the tree is the generic sequence which represents all entries in the cluster. The generic message sequence merging process can be accelerated by extracting the principal components of the elements in the cluster. Each message can then in turn be selected based on its similarity to each distinct principal component. This means that instead of processing every message in the cluster, only the most distinct messages are processed.

4.3 Over Simplification Problem

Once the variable fields and their infix delimiters have been identified, the observed values of the fields can be extracted. The default values can be useful to a fuzzing application when the field is not being fuzzed. An example of a situation where this would be required is with a network fuzzer where a session cookie inside the network packet authorises network traffic to communicate with REST services. If the session cookie was being altered the packet would be refused by the server. Alternatively with a file fuzzer, the magic number of the file could be "MZ" if the file is a compiled program, and altering the magic number of the file would usually prevent any applications from opening it, or could limit the code coverage of the program opening the file to only testing to the code that does the file header precheck.

4.4 Optional Data Fields

A case where generic message sequences overlook the semantics of the data is with optional fields. XML data is an example of data which elements of the data can be optionally included or excluded. An example of this problem is with a java ellipsis for defining a variable number of parameters. Given the function "create(String... animal)" two sample use cases would be "create(dog,cat)" and "create(cat,dog,bat)". During the clustering phase by using the longest common substring metric as the only metric for clustering, these items may be clustered together and generalised into a generic message sequence "create(—,——-)". This results in the ",bat" substring being substituted with gaps in the generic message sequence. A solution to this problem is to enumerate all substrings common to all the messages within a cluster at the variable offsets within each message. The substrings that would be found in the example would be "cat", "dog", "dog,bat". The first two values would be found in the first two fields across the two messages. The last value would be due to the generic message sequence inserting gaps into the message with the optional field as the delimiter separating it from the other fields is not found in the other messages. To prevent this false token enumeration, only tokens common to two or more messages should be enumerated and tokens that occur only once should be discarded.

4.5 Enumerating Observed N-Grams

By dividing a message into a set of variable length n-grams starting from the maximum length and then repeating the process for smaller values of n, the other messages can be queried with the values of the n-grams from the first message to find common substrings. The initial results for maximum length n-grams should return no query results for common message substrings. However for each iteration of reducing the size of the n-grams and querying the other messages, the number of returned results should increase as the n-gram length gets shorter. Once the n-grams approach length 1, the number of results reaches a saturation point.

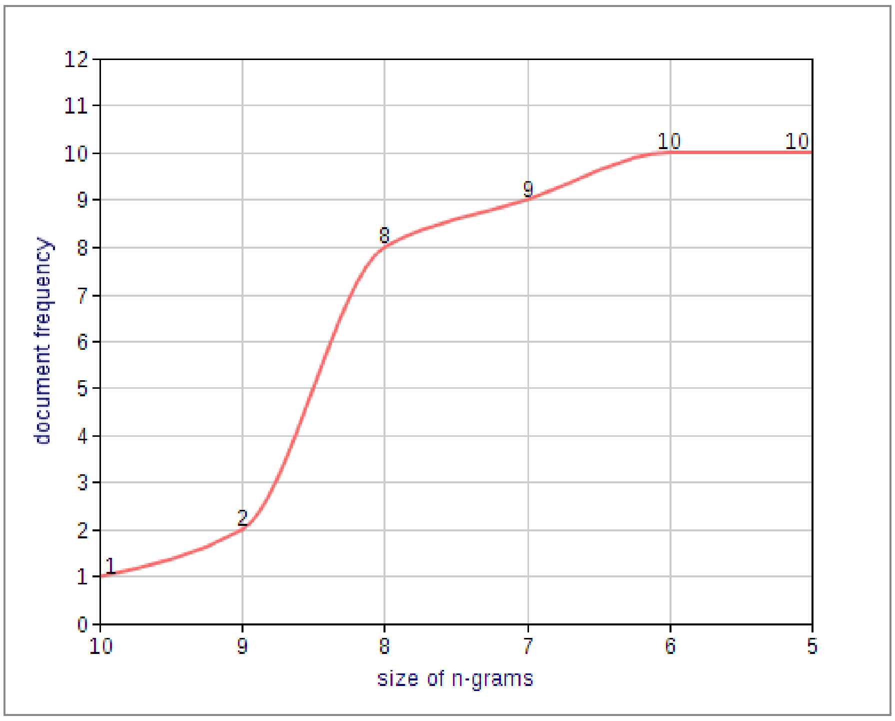


FIGURE 4.4: Graph Of N-Gram Frequency

Ideally the longest n-grams with the maximum number of shared substrings across the whole set of messages are the most desirable as they represent something specific (due to its length) that is frequently used and is not too close to the saturation point. Performing a search across the set of messages for a single substring token with variable n-gram lengths will return results that vary in number.

For example, given a set of messages: "cat-in-the-hat","cat-in-the-hut" and "cat-in-de-house" by tokenizing the first message from an offset of zero (the beginning of the message) into an n-gram of length seven "cat-in-the-hat" may only return one result, the message itself. Re-tokenizing the message from the same offset into an n-gram of length ten "cat-in-the" might return two results, and searching for "cat" would return three results. Deciding on which n-gram length is most appropriate for a given offset within a message is a separate problem. There exists a metric to decide which offset is the best given the length of the n-gram and the number of results. Metrics used for model selection in state transition diagrams and clustering algorithms can also be used to select the appropriate length for n-grams. A metric called the elbow criterion, also known as the percentage of variance, or sometimes called an F-Test is commonly used for model selection. This can be explained as a ratio of the between-group variance to the total variance. By using this metric to identify the elbow in the graph, a suitable n-gram length can be selected and the offset can be incremented within the message to the next variable token. Once the offset has reached the end of the first message, the process can be repeated again from the beginning of the next message. This process repeats itself until all messages in the set have been tokenized.

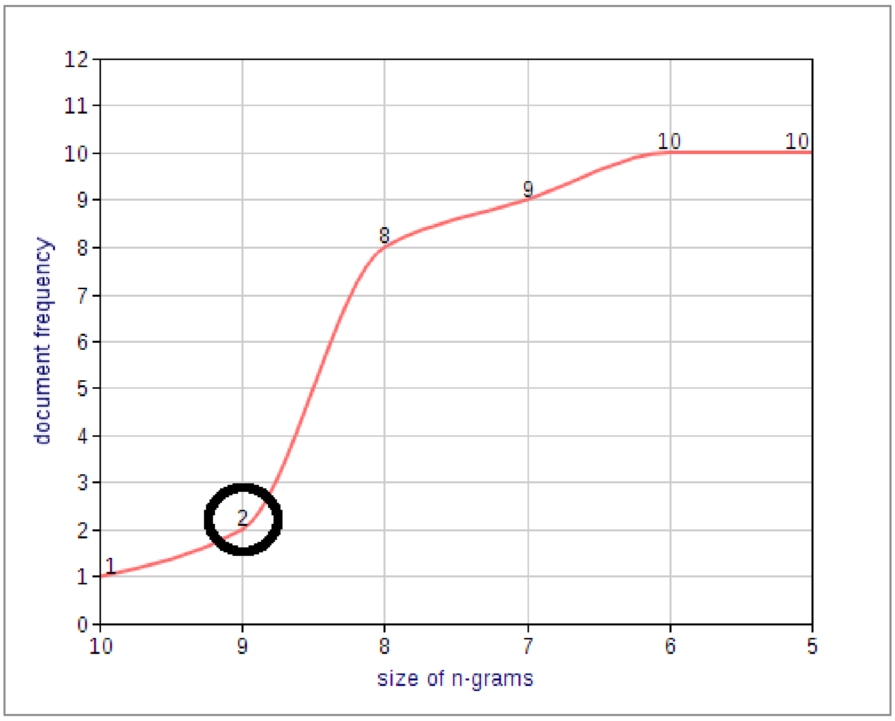


FIGURE 4.5: Percentage Of Variance For N-Grams

Enumerating the n-grams in the dataset is a computationally expensive operation. This problem can be accelerated by reducing the amount of work required. By performing an exhaustive search, we begin at offset zero of the first message and select the maximum length for the n-gram. By selecting the offsets located at the beginning of each variable token in the generic message sequence, each variable part of the generic message sequence can be the beginning offset for an n-gram and non-variable parts of the generic message sequence do not need to be included in the n-gram. For example, given the generic message sequence "cat-in-XXe-hX" where the "X" represents the variable data, the creation of n-grams can begin at an offset of seven characters with a maximum length of two characters. N-grams that begin at an offset of zero for the first token begin with a length equal to that of the variable tokens length. For each iteration of queries made with a tokens n-grams across the dataset, the length is reduced until it reaches a saturation point. Once the saturation point is reached, the offset is incremented by one so that the beginning of the n-grams is operating on the same token but not including the first character, and the length of the n-gram is set to the length of the token reduced by one. An example of this starting with the words "cat-bus" would be tokenized as: "cat", "ca", "c". Then the offset within the same token is incremented so the n-grams would be: "at", "t". And the final n-gram for the first token would be "t". Then the second token would be split as ”bus”, ”bu”, ”b”, ”us”, ”s”, ”s”. This method can enumerate numeric values used in variable alphanumeric fields.

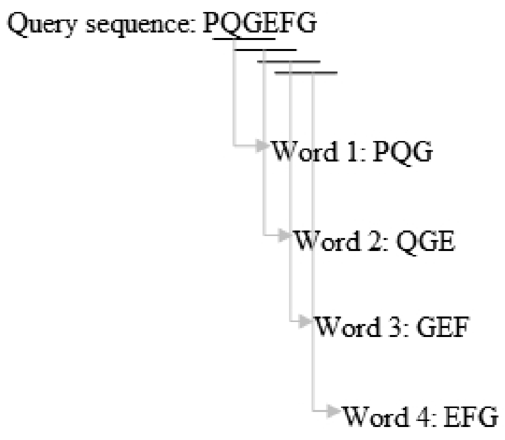


Figure 4.6: Incrementing Starting Point Of Ngrams[1][2]

Each n-gram is then searched for across the entire dataset. To reduce the inter-message search space, the n-gram only needs to search within its assigned cluster from the clustering process. Given five distinct clusters of generic message sequences, the search space can be reduced by one fifth. Additionally with the intra-message search space, the search within a single message can begin at the same offset that the variable data begins in the generic message sequence.

Lastly, as each n-gram iteratively searches across the dataset, its length is decremented until a saturation point is reached. If the frequency of a collection of n-grams starting from the same single offset is recorded during the search and the saturation point is calculated afterwards, then a lot of time will be spent calculating the frequency of small n-grams of length one and two. To reduce the time spent calculating the frequency of less useful n-grams, the saturation point should be calculated for every iteration of n-gram length at a single offset. This means that once there is a large jump in frequency, the n-gram length no longer needs to be decremented and the offset can be changed to the next variable portion of the generic message sequence.

4.6 Field Data Type Granularity

Once all the tokens have been identified, they can begin to be classified into distinct data types such as: booleans, small integers, integers, doubles, floats, hashes, ascii characters and binary. This is as simple as attempting to cast each value and catching the error if one is returned. The casting process should begin with the most specific and likely to fail. For each failure to cast a token, a less specific cast is attempted until a cast to

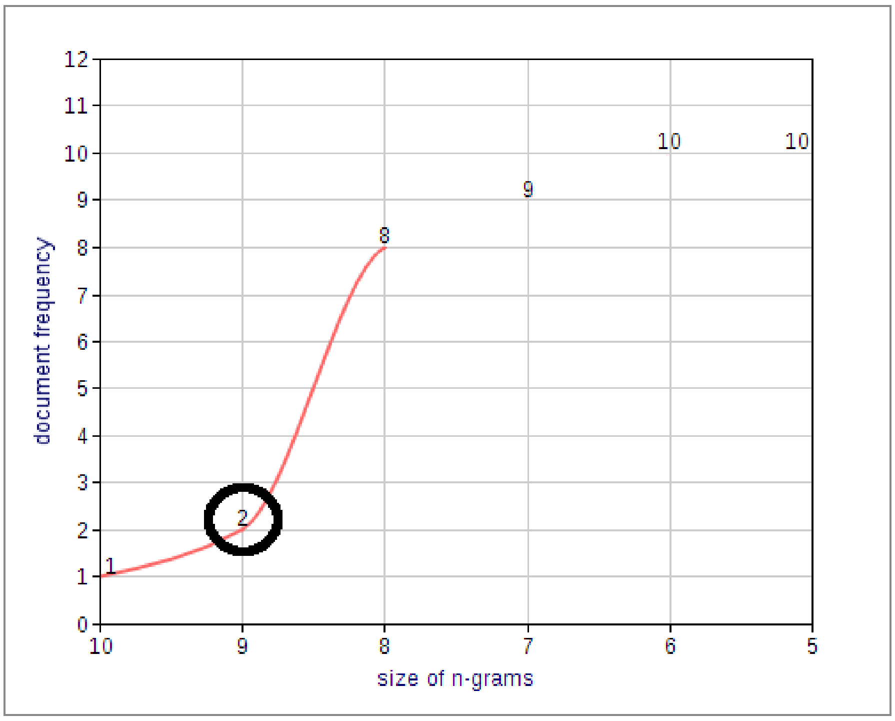


Figure 4.7: Restricting Ngram Lookups To Variable Fields

binary data is reached, which will never fail. During this casting process, certain types may be similar making it difficult to distinguish between them such as a hash value and ascii.

4.7 Casting High Entropy Fields

There are certain approaches which attempt to solve this problem. The first is a probabilistic approach, by processing a data value such as an MD5 hash of the string ”test” would give the value ”098f6bcd4621d373cade4e832627b4f6”. It is not immediately obvious that this is a hash of a piece of data. A probabilistic approach to identifying a hash would be to calculate its Shannon Entropy value 0.11814. This is a measure of how difficult the value is to predict, and as hashes are designed to be difficult to predict, it is a suitable metric. Another approach would be to use a hash identifying tool which uses several metrics to differentiate between types of hashes. These metrics include factors such as whether or not the string is in lowercase or uppercase, it is numeric or alphabetic or alphanumeric and finally the length of the string. If the hashing algorithm that created the string can not be identified, then the casting process should continue until a suitable cast is reached.

4.8 Casting Basic Fields

Assigning data types to variable offsets within a generic message sequence is trivial when the data is separated using predictable infix delimiters. However when fields are composed of compound data types such as alphanumeric fields, then the complexity increases. Compound fields can be handled by attempting to cast each of the enumerated n-grams for a variable field in a generic message sequence and catching the error if one is returned. If a single n-gram fails to cast, then the casting datatype should be changed to a more general one and the process should start again from the beginning. For example given the n-grams "cat", "bat", "dog12" each n-gram should be attempted to be cast as numeric which will fail on the first token. Then the data type should be changed to something more general such as alphabetic, which will fail on the last token as it contains numbers. The casting datatype should be changed again to something more general such as alphanumeric, which should pass for every token.

4.9 Field Relationships (Prefix and Postfix Delimiters)

Once the message tokens have been identified additional information such as intra-message and inter-message semantics is perceived to be a problem with a high Kolmogorov complexity making it computationally expensive to infer automatically. This is true given a set of messages which have not been tokenized and had their field data types labelled. However by first creating the generic message sequence, identifying the n-grams, then classifying the field data types by considering all n-grams at each offset within the generic message sequence, the complexity of the problem has already been reduced in the previous steps. Finding a field referring to intra-message or inter-message length only requires checking fields previously labelled as integers as other data types such as hashes can be ruled out. The remaining problem is now trying to identify which previously identified integer field refers to the length in question.

Once the n-grams for variable fields have been enumerated, identifying which fields relate to intra-message properties and which fields relate to inter message properties is simply a case of addition of metrics and selecting whichever result has complete agreement across all packets within the same cluster. For example, given two generic message sequence classes A and B, a field has been identified in class A to contain the values 5, 6 and 7 across 3 messages. By arranging the messages in order of occurrence "ABBBBBABBBBBBABBBBBBB", measurements can be made on both the inter-message and intra-message properties. Given that the field occurs in class A, the intra-message properties could be: number of parameters, time message was sent, checksum of data, hash of data etc. The inter-message properties could include: number of messages of the same class or different class that succeeds the message, The size of the data that succeeds the message, a checksum of the data that succeeds the message, a hash of the data that succeeds the message. Given that a large amount of data is collected, messages sequences can be observed acting under different circumstances, as in the example the message classes are repeated three times with three different parameters. This means that if all intra-message and inter-message measurements are considered, an analysis will find that the "number of messages of a different class that succeeds the message" metric will have the maximum agreement as B succeeds A 5, 6 and 7 times respectively, and the other metrics will probably have less agreement or even no agreement at all.

Chapter 5

State Machines

5.1 State Machine Inference

A state machine is a state transition diagram where each vertex represents a state and each edge represents a transition between states[43]. Finite state machines resemble finite markov chains however the transitions in a finite markov chain are probabilistic in contrast to a deterministic finite state machine. Given a set of transitions between unknown states it is possible to reverse engineer the number and location of distinct states using hidden markov models[44][45]. This means that given transitions A1, B2, C3, B4, C5, D6 the class A can be said to be a transition from an entry state, class D can be identified as a transition to an exit state, and classes C and D are two transitions that can iterate back and forth between three states[46]. This assumes each combination of adjacent transitions such as A1B2, B2C3 and C3B4 etc each represent a distinct state[47].

5.1.1 Forward Algorithm

However one of the disadvantages of using hidden markov models is the need for a prior knowledge of the state topology. Domain specific knowledge is usually required to create the hidden markov model transition probability matrix and observation probability matrix. Once these matrixes have been constructed using domain specific knowledge, new observations can be made and the probability of being in particular states can be calculated based on specific observations. The algorithm for performing this operation is called the Forward Algorithm[48].

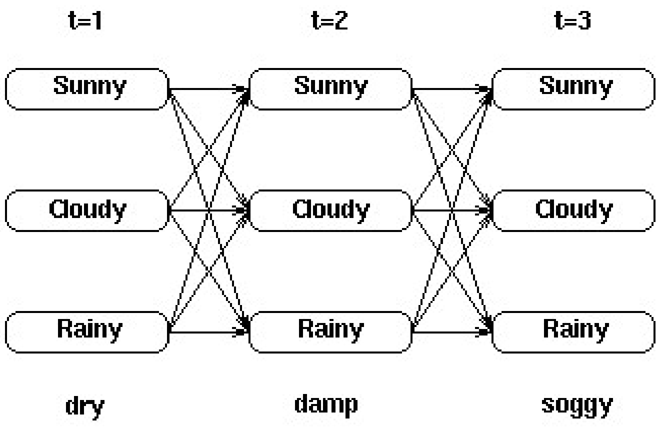


FIGURE 5.1: Full Trellis Topology

5.1.2 Viterbi Algorithm

In addition to inferring a state sequence based on a dataset of new transitions, the most probable path can also be calculated using the training data. The algorithm for performing this operation is called the Viterbi Algorithm and it removes the probability of all possible paths[49]. A graph generated by the Forward Algorithm would be a fully connected topology, however a graph generated by the Viterbi Algorithm would simply be a subgraph of only the most probable states and their transitions[48][50].

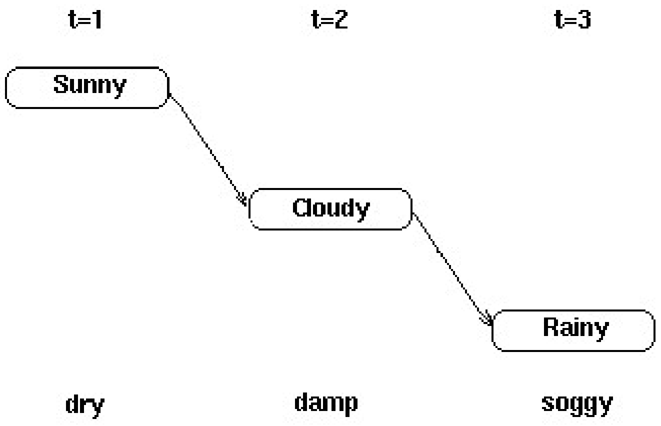


FIGURE 5.2: Trellis Subgraph Topology

A better solution to constructing the transition probability and observation probability matrices would be to infer the matrices based on the data. Calculating the probabilities automatically is an NP-complete problem as all possible sets of probabilities must be searched for every possible sequence of transitions. By using an expectation maximization algorithm to calculate probabilities using missing data, the amount of work that needs to be done can be reduced. One such algorithm is the Baum-Welch Algorithm, also known as the Forward-Backward Algorithm.

5.1.3 Baum-Welch Algorithm

The Baum-Welch Algorithm begins with a HMM trellis consisting of a number of states which is either initialized with a random number[51], initialized to one state, or initialized with many states[52]. Observed transitions between hidden states are processed in order until all probabilities have been calculated. The process then begins again by re-initializing with different parameters, such as a different number of states[53]. There are two distinct approaches to initializing the Baum-Welch algorithm: state splitting and state merging[49]. State splitting is when the algorithm is initialized with one state on the first iteration and subsequently adds states on every iteration until a threshold has been reached. State merging is when the algorithm is initialized with a complex model on the first iteration, and subsequently removes states with every iteration until a threshold is reached.

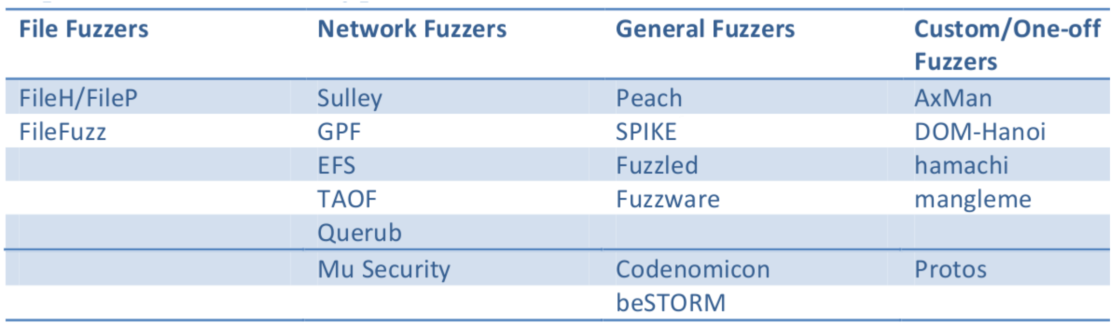
State splitting and merging strategies each use two metrics in their operation. Each strategy needs to be able to identify which state to split or merge, and each strategy needs to know when a threshold has been reached so it knows when to stop splitting or merging. A popular solution is to perform an exhaustive search by generating all possible solutions and use model selection metrics such as the Bayesian Information Criterion (BIC)[50][54], the Akaike Information Criterion (AIC), the Minimum Description Length (MDL) or the percentage of variance to identify the most accurate model.

Chapter 6

Output Format

6.1 Representing Protocol Specification

Fuzzers can be grouped into four generic categories: file fuzzers, network fuzzers, general fuzzers and one-off fuzzers. All of these fuzzer types are designed to manipulate data which is destined for a computer program. Some of these programs allow data to be input from a file, or from packets on a network, or from the keyboard, or other forms of input such as a microphone.

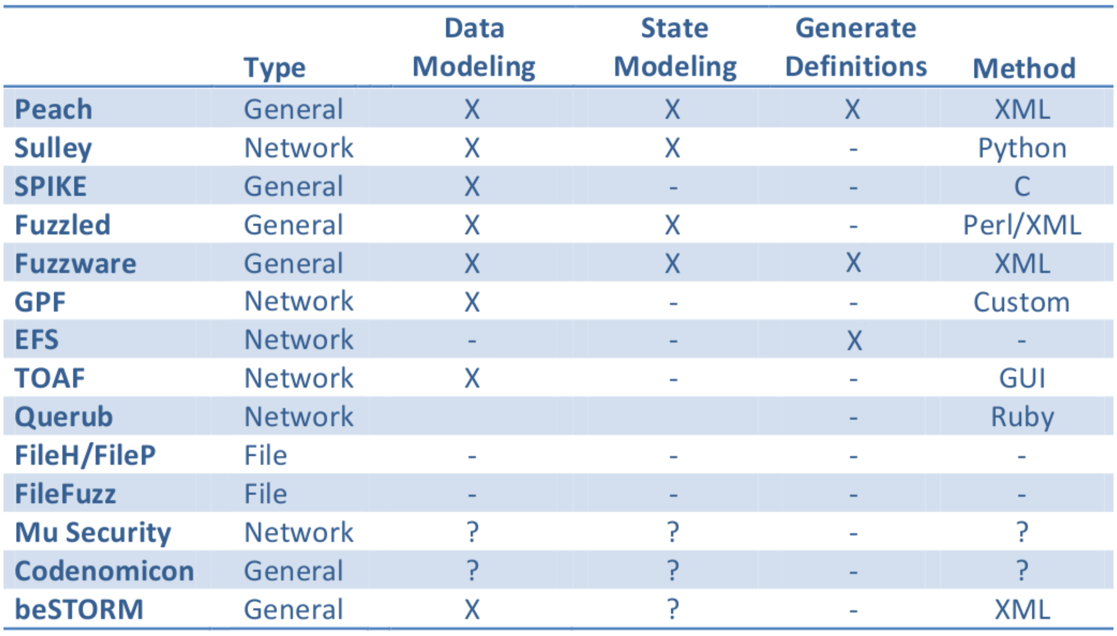


File Fuzzers	Network Fuzzers	General Fuzzers	Custom/One-off Fuzzers
FileH/FileP	Sulley	Peach	AxMan
FileFuzz	GPF	SPIKE	DOM-Hanoi
	EFS	Fuzzled	hamachi
	TAOF	Fuzzware	mangleme
	Querub		
	Mu Security	Codenomicon	Protos
		beSTORM	

Figure 6.1: Fuzzer Features[3]

When a fuzzer is running, it is designed to alter the data being input in a computer program in an attempt to find security vulnerabilities. A simple HTML page can have hundreds of elements each of which a fuzzer should test. If a file fuzzer is given one HTML file as an input, a file fuzzer might generate thousands of altered HTML files. Each sample of manipulated data represents a single test case designed to investigate a particular vulnerability. More complex test cases may include compound changes to a file by creating multiple tests per test case. For a file fuzzer to perform a test with a HTML file, it must open the file using the application being tested for security vulnerabilities.

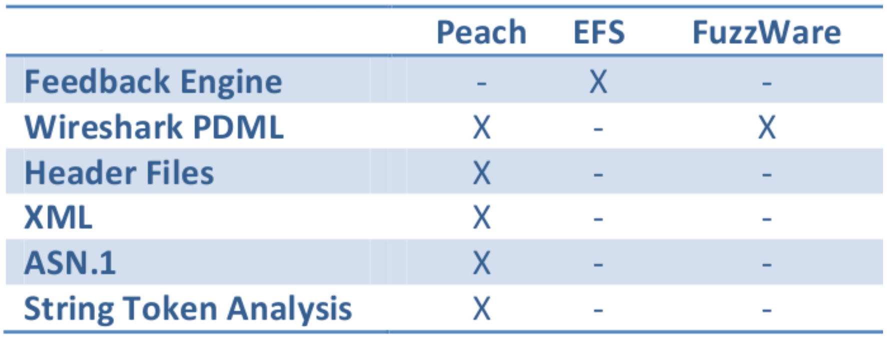
Each test case may contain valid data which follows a specification, or alternatively some test cases may contain invalid data purposely designed to break the specification in an attempt to find a vulnerability. In order to either follow or break a specification, the rules must be defined. There exist various tools which have already implemented methods of defining a specification.



	Type	Data Modeling	State Modeling	Generate Definitions	Method
Peach	General	X	X	X	XML
Sulley	Network	X	X	-	Python
SPIKE	General	X	-	-	C
Fuzzled	General	X	X	-	Perl/XML
Fuzzware	General	X	X	X	XML
GPF	Network	X	-	-	Custom
EFS	Network	-	-	X	-
TOAF	Network	X	-	-	GUI
Querub	Network			-	Ruby
FileH/FileP	File	-	-	-	-
FileFuzz	File	-	-	-	-
Mu Security	Network	?	?	-	?
Codenomicon	General	?	?	-	?
beSTORM	General	X	?	-	XML

FIGURE 6.2: Fuzzer Features[3]

There currently exists three open source fuzzers capable of analysing data and representing the structure of the data using data models and state machines. Of these three projects, two of them have a clearly defined data model: Peach and FuzzWare. Both Peach and FuzzWare use XML to represent the structure of the data, making their data models a desirable choice for outputting the results of the analysis outlined above.



	Peach	EFS	FuzzWare
Feedback Engine	-	X	-
Wireshark PDML	X	-	X
Header Files	X	-	-
XML	X	-	-
ASN.1	X	-	-
String Token Analysis	X	-	-

FIGURE 6.3: Fuzzer Supported Data Models[3]

6.2 Peach Pit XML Files

Peach data model files support primitive data types of various sizes as well as complex data types. These fields are represented as an XML document. Each field can have a number of attributes some of which are mandatory and some of which are optional such as name, size and default value. Other attributes such as transformers and fixups perform operations on data so that it can be digested properly by the application being tested, such as base 64 encoding or hashing the data.

6.2.1 Primitives

Peach supports elements for representing primitives including Blob, String, Flag, Number, Padding. These elements support a number of optional attributes. The padding element allows to align its parent element to any bit or byte boundary specified.

```
<DataModel name="Example1">
<Blob name="Unknown1" valueType="hex" value="01 06 22 03"/>
<String name="Key" value="Content-Length" />
<Flag name="compression" position="0" size="1" />
<Number name="Hi5" value="5" size="32"/>
<Padding aligned="true"/>
</DataModel>
```

6.2.2 Fixups and Transformers

Fixups and transformers are simple data alterations that allow data to be correctly formatted. Transformers operate on the parent elements whereas Fixups alter another element's data.

```
<Block name="base64Block">
<Transformer class="Base64Encode" />
<Fixup class="MD5Fixup">
<Blob name="Data" />
</Fixup>
</Block>
```

6.2.3 Relations

Relations enable values of data to effect the values of other data, such as sizes and offsets. If the value of the data changes in the block element named"base64Block", its metadata is set as the value for the number element titled "Length".

```
<DataModel name="Base64TLV">
<Number name="Type" size="8" signed="false" value="1" token="true" />
<Number name="Length" size="16" signed="false">
<Relation type="size" of="base64Block" />
</Number>

<Block name="base64Block">
<Transformer class="Base64Encode" />
<Blob name="Data" />
</Block>
</DataModel>
```

Chapter 7

Risk Inference

7.1 Determining Risk

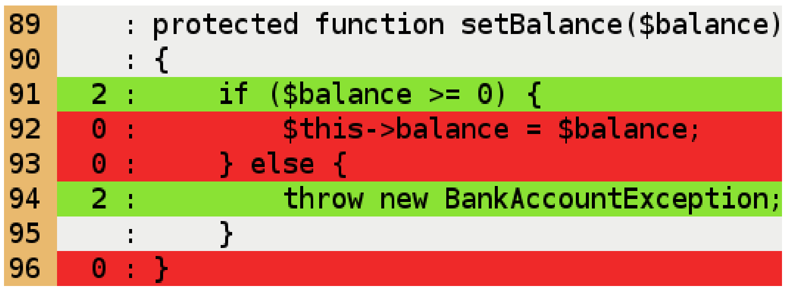
Creating a data model of a file or protocol manually is a relatively time consuming task and there is a lot of time to be saved by doing it automatically using tools. In addition to creating the data model, a small amount of time is spent generating test cases but significantly more time is spent executing the test cases in an attempt to find vulnerabilities.

The Microsoft Security Development Lifecycle (SDL) requires a minimum of 250,000 iterations of a fuzzer without any bugs being found to consider that being compliant with the SDL. This means that if a test is on the 100,000th iteration and a bug is found, the counter has to be reset and the fuzzing process must begin again from the beginning before the 250,000th iteration is reached. This means that a fuzzing process could easily go through between 500,000 and a million iterations before it is considered being compliant with the SDL.

If a test case takes 30 seconds and there are 500,000 iterations, this means that the fuzzing process will take approximately 174 days before it will complete. This is longer than most software development sprints. Parallelization is a common solution to this problem, by simply dividing the test cases across several machines and using more resources, the workload can be shared and completed at the same time by several computers at once. By dividing the workload between 10 computers the time to complete a fuzzing test cycle can be reduced to 17 days.

7.2 Code Coverage

A method of reducing the time and resources required, is to be more selective with which test cases are being executed. This method requires process monitoring tools or code coverage tools[55]. If file fuzzing is being performed, the test cases that are going to be generated are usually based on a sample file of which copies of it are then modified and sent to the program being tested. However depending on the contents of the file, different functionality of the program being will be tested. For example with a program such as Microsoft word, its source code is compiled into different dynamically linked libraries. One library may contain the code relating to displaying various typefaces and fonts, while another library may provide functionality for tables and figures. This means that by opening the sample file in the program being tested and monitoring that program using code coverage tools, the comprehensiveness of a test can be measured in terms of number of lines of code being executed[56]. In terms of reducing the time to complete a test and measuring the comprehensiveness of a test, there is a clear tradeoff: more comprehensive tests require more time to execute.



```
  : protected function setBalance($balance)
  : {
2 :     if ($balance >= 0) {
0 :         $this->balance = $balance;
0 :     } else {
2 :         throw new BankAccountException;
  :     }
0 : }
```

Figure 7.1: Code Coverage

Certain code coverage tools provide detailed information relating to each line of code that is executed, such as how long each line of code took to execute and the number of times the line was executed[57]. This line count can however can provide misleading information as a program can be executed differently depending on the options provided to the compiler. Regarding the flow control of a program, loops can be executed differently depending if the compiler is optimizing the code for the program to have the smallest possible size or if it is optimizing the code to have the fastest possible speed. An example of this is with loop unwinding optimizations. This process neglects the output size of the program in an attempt to reduce the latency introduced by the assertion of a loop. In the following example a for loop iterates 10 times to display the value of the variable. At the beginning of each iteration the value of the variable is asserted and at the end of the iteration its value is incremented.

```
for(int x=0; x<10; x++){
printf(x);
}
```

In the following example the loop has been unwound using a compiler optimization to reduce the time to execute the instructions at the expense of the size of the program. Executing the unwound loop requires it to be iterated twice only meaning the assertions are only executed twice.

```
for(int x=0; x<10; x+=5) {
printf(x);
printf(x+1);
printf(x+2);
printf(x+3);
printf(x+4);
}
```

This means that if the code coverage of a test case is being measured the program being tested should be compiled with optimizations for size enabled, allowing for an accurate determination of the number of times an instruction is being executed. If loop unwinding is enabled for performance, code coverage tools will need to be selected depending if they support monitoring of both the line numbers of the source code and virtual memory addresses of the binary program.

7.3 Topical Vulnerabilities

Best case, worst case and average case are three of the potential outcomes for a test cycle. Assume a product has 4 vulnerabilities with a maximum CVSS score of 10 which will be discovered during the fuzzing test cycle. This means the product has 5 vulnerabilities that make it completely insecure thus compromising the entire system and must be fixed before the product is released and can not be fixed after release in a fixpack or update. In a worst case scenario each of these 4 vulnerabilities will be discovered between the 249,996th and 250,000th iteration of the test cycle. This means the test cycle will complete 5 times before it is considered to be SDL compliant totaling 1,250,000 tests. However in reality this is unlikely and most users will observe the average during their test cycle.

A desirable quality of a fuzzer is to achieve best case scenario results where vulnerabilities are found very early on during the test cycle and the SDL compliance is achieved with as little redundant testing as possible. By ordering test cases in order of the most topical vulnerabilities first and least topical vulnerabilities first, a tester can increase the chances of encountering vulnerabilities at an early stage of the test cycle which will reduce the number of iterations required for a clean run of 250,000 test cases required for SDL compliance. For example if a product contains a third party library such as libssl and 250,000 test cases have been generated with the buffer over read test cases at the end of the test cycle, this means that 249,999 test cases will need to be executed before the iteration count is reset and the tests are started from the beginning. By enumerating a list of the most recent vulnerabilities disclosed and their respective vulnerability types, test for these vulnerabilities can be performed at the start of the test cycle in an attempt to achieve the best-case scenario for the number of iterations required for the SDL.

The Common Vulnerabilities and Exposures (CVE) website provides xml feeds of their latest vulnerabilities and each vulnerability is accompanied by its unique vulnerability code to identify it. Each vulnerability is composed of its description, vendor, the product name, the product version and another unique code called a CWE referring to the type of vulnerability found. The Common Weakness Enumeration (CWE) lists vulnerability types. Each vulnerability type documented with the CWE is assigned its own unique code. In order to enumerate the most topical vulnerabilities, the CWE identification numbers must be extracted from the most recently reported CVEs. The CWE description can be matched to an appropriate element in the data model. For example if a CWE description contains the word buffer, then data model elements such as blob and block can be identified as likely places to start testing for topical vulnerabilities.

Whilst predicting certain data model fields as being more likely to contain vulnerabilities that are currently trending as more probable for finding vulnerabilities than the other fields, this probability must take into account the size of the data and the size of the source alphabet containing n unique symbols. The larger the attack surface an application has the larger chance there is of finding a vulnerability. Thus if an application supports the 26 characters of the english alphabet there is a 1/26 chance of finding a character that is not supported, this implies that a larger alphabet results in a larger attack surface. In addition to the supported alphabet, the encoding scheme must also be considered. The same character in a single alphabet can be encoded in several different ways. If an application supports multiple encoding schemes, this increases the attack surface by an even larger extent.

Appendix A

Smith Waterman algorithm

The following code demonstrates the Smith-Waterman algorithm and is taken from the open source Puriney GitHub repository[58].

```
#!/usr/bin/python
from math import *
strG = str(raw_input("Genome sequence: "))
strR = str(raw_input("Read sequence: "))
strG = strG.upper()
strR = strR.upper()
print "Your genome ref input : " +strG
print "Your read ref input   : " +strR

matrix = []
path = []
row = len(strR)
col = len(strG)
strR = "^"+strR
strG = "^"+strG
for i in range(row+1):
        matrix.append([0]*(col+1))
        path.append(["N"]*(col+1))

def print_matrix(matrix):
        print '\t'+('\t'.join(map(str,list(strG))))
        i = 0
        for line in matrix:
                print strR[i]+"\t"+('\t'.join(map(str,line)))
                i +=1

# print_matrix(matrix)
indelValue = -1
matchValue = 2
for i in range(1,row+1):
        for j in range(1,col+1):
                # penalty map
                from_left = matrix[i][j-1] + indelValue
                from_top = matrix[i-1][j] + indelValue
                if strR[i]==strG[j]:
                        from_diag = matrix[i-1][j-1] + matchValue
                else:
                        from_diag = matrix[i-1][j-1] + indelValue

                matrix[i][j]= max(from_left,from_top,from_diag)
                # path map
                if matrix[i][j]==from_left:
                        path[i][j]="-"
                elif matrix[i][j]==from_top:
                        path[i][j] = "|"
                elif matrix[i][j] == from_diag:
                        path[i][j] = "M"
                else:
                        pass

                if matrix[i][j]<0:
                        matrix[i][j]=0
pass
print_matrix(matrix)
print
print_matrix(path)

iRow = len(matrix)-1
jCol = len(matrix[0])-1

while iRow>=0:
        maxPnltyVaue = max(matrix[iRow])
        while jCol>=0:
                if matrix[iRow][jCol]==maxPnltyVaue:
                        ipair = iRow
                        jpair = jCol
                        reportR=[]
                        reportG=[]
                        while (1):
                                if ipair ==0 and jpair==0:
                                        break
                                # else:
                                if path[ipair][jpair]=="M":
                                        reportR.append(strR[ipair])
                                        reportG.append(strG[jpair])
                                        ipair -= 1
                                        jpair -= 1
                                elif path[ipair][jpair]=="-":
                                        # reportR.append(strR[ipair])
                                        reportR.append("-")
                                        reportG.append(strG[jpair])
                                        # ipair -= 1
                                        jpair -= 1
                                elif path[ipair][jpair]=="|":
                                        reportR.append(strR[ipair])
                                        reportG.append("-")
                                        ipair -= 1
                                        # jpair -= 1
                                elif path[ipair][jpair]=="N":
                                        if ipair > 0:
                                                reportR.append(strR[ipair])
                                                ipair -=1

                                        if jpair > 0:
                                                reportG.append(strG[jpair])
                                                jpair -= 1

                                print "Your alignment would be followed: "
                                print "R:"+ "".join(reversed(reportR))
                                print "G:"+ "".join(reversed(reportG))
                                print
                        jCol -=1
                iRow -=1
```

Appendix B

State Machine Inference Algorithms

The following code is an excerpt from Machine Learning: An Algorithmic Perspective, and demonstrates the Forward, Viterbi and Baum-Welch algorithms[59].

```
# Code from Chapter 15 of Machine Learning: An Algorithmic Perspective
# by Stephen Marsland (http://seat.massey.ac.nz/personal/s.r.marsland/MLBook.html
    )

# You are free to use, change, or redistribute the code in any way you wish for
# non-commercial purposes, but please maintain the name of the original author.
# This code comes with no warranty of any kind.

# Stephen Marsland, 2008

# A basic Hidden Markov Model
from numpy import *

def HMMfwd(a,b,obs):

        nStates = shape(b)[0]
        T = shape(obs)[0]

        alpha = zeros((nStates,T))

        alpha[:,0] = aFirst*b[:,obs[0]]

        for t in range(1,T):
                for s in range(nStates):
                        alpha[s,t] = b[s,obs[t]] * sum(alpha[:,t-1] * a[:,s])

        #alpha[q,T] = sum(alpha[:,T-1]*aLast[:,q])
        #print max(alpha[:,T-1])
        return alpha
```

```
def HMMbwd(a,b,obs):

        nStates = shape(b)[0]
        T = shape(obs)[0]

        beta = zeros((nStates,T))

        beta[:,T-1] = aLast

        for t in range(T-2,0,-1):
                for s in range(nStates):
                        beta[s,t] = b[s,obs[t+1]] * sum(beta[:,t+1] * a[:,s])

        beta[:,0] = b[:,obs[0]] * sum(beta[:,1] * aFirst)
        return beta

def BaumWelch(obs,nStates):

        T = shape(obs)[0]
        a = random.rand(nStates,nStates)
        b = random.rand(nStates,T)
        olda = zeros((nStates,nStates))
        oldb = zeros((nStates,T))
        maxCount = 50
        tolerance = 1e-5

        count = 0
        while (abs(a-olda)).max() > tolerance and (abs(b-oldb)).max() > tolerance
     and count < maxCount:
                # E-step

                alpha = HMMfwd(a,b,obs)
                beta = HMMbwd(a,b,obs)
                gamma = zeros((nStates,nStates,T))

                for t in range(T-1):
                        for s in range(nStates):
                                gamma[:,s,t] = alpha[:,t] * a[:,s] * b[s,obs[t
    +1]] * beta[s,t+1] / max(alpha[:,T-1])

                # M-step
                olda = a.copy()
                oldb = b.copy()

                for i in range(nStates):
                        for j in range(nStates):
                                a[i,j] = sum(gamma[i,j,:])/sum(sum(gamma[i,:,:]))

                for o in range(max(obs)):
                        for j in range(nStates):
                                places = (obs==o).nonzero()
                                tally = sum(gamma[j,:,:],axis=0)
                                b[j,o] = sum(tally[places])/sum(sum(gamma[j,:,:])
    )
```

```
                                        #print b[j,o], sum(gamma[j,places])/sum(gamma[j
    ,:])

                        count += 1
                print count
                return a,b

def ViterbiSimple(a,b,obs):

        nStates = shape(b)[0]
        T = shape(obs)[0]

        path = zeros(T)
        viterbi = zeros((nStates,T))

        viterbi[:,0] = aFirst * b[:,obs[0]]
        path[0] = argmax(viterbi[:,0])

        for t in range(1,T):
                for s in range(nStates):
                        viterbi[s,t] = max(viterbi[:,t-1] * a[:,s] * b[s,obs[t]])
                path[t] = argmax(viterbi[:,t])

        print "Path: ", path
        #print viterbi[path[T-1]]
        return path,viterbi

def Viterbi(a,b,obs):

        nStates = shape(b)[0]
        T = shape(obs)[0]

        path = zeros(T)
        backwards = zeros((nStates,T))
        viterbi = zeros((nStates,T))

        viterbi[:,0] = aFirst * b[:,obs[0]]
        backwards[:,1] = 0

        for t in range(1,T):
                for s in range(nStates):
                        tally = viterbi[:,t-1] * a[:,s] * b[s,obs[t]]
                        backwards[s,t] = argmax(tally)
                        viterbi[s,t] = tally[backwards[s,t]]
                path[t] = argmax(viterbi[:,t])

        print path
        return path,viterbi,backwards


aFirst = array([0.25,0.25,0.25,0.25])
aLast = array([0.25,0.25,0.25,0.25])
#a = array([[.7,.3],[.4,.6]] )
a = array([[.4,.3,.1,.2],[.6,.05,.1,.25],[.7,.05,.05,.2],[.3,.4,.25,.05]])
#a = a.transpose()
```

```
#b = array([[.2,.4,.4],[.5,.4,.1]] )
b = array([[.2,.1,.2,.5],[.4,.2,.1,.3],[.3,.4,.2,.1],[.3,.05,.3,.35]])
obs = array([0,0,3,1,1,2,1,3])
#obs = array([2,0,2])
HMMfwd(a,b,obs)
Viterbi(a,b,obs)
ViterbiSimple(a,b,obs)
BaumWelch(obs,4)
ViterbiSimple(a,b,obs)
```

Bibliography

[1] Ziqi Wang, Gu Xu, Hang Li, and Ming Zhang. A fast and accurate method for approximate string search. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies - Volume 1*, HLT '11, pages 52–61, Stroudsburg, PA, USA, 2011. Association for Computational Linguistics. ISBN 978-1-932432-87-9. URL http://dl.acm.org/citation.cfm?id=2002472.2002480.

[2] Michael Krauthammer, Andrey Rzhetsky, Pavel Morozov, and Carol Friedman. Using {BLAST} for identifying gene and protein names in journal articles. *Gene*, 259(1–2):245 – 252, 2000. ISSN 0378-1119. doi: http://dx.doi.org/10.1016/S0378-1119(00)00431-5. URL http://www.sciencedirect.com/science/article/pii/S0378111900004315.

[3] Michael Eddington. Demystifying fuzzers. *Black Hat Europe*, 2009.

[4] Marshall A Beddoe. Network protocol analysis using bioinformatics algorithms, 2004.

[5] Patrick LaRoche, Aimee Burrows, and A. Nur Zincir-Heywood. How far an evolutionary approach can go for protocol state analysis and discovery. In *IEEE Congress on Evolutionary Computation*, pages 3228–3235. IEEE, 2013. ISBN 978-1-4799-0453-2. URL http://dblp.uni-trier.de/db/conf/cec/cec2013.html#LaRocheBZ13.

[6] Antonio Trifilo, Stefan Burschka, and Ernst W Biersack. Traffic to protocol reverse engineering. In *CISDA 2009, 2nd IEEE Symposium on Computational Intelligence for Security and Defence Applications, July 8-10, 2009, Ottawa, Canada*, Ottawa, CANADA, 07 2009. doi: http://dx.doi.org/10.1109/CISDA.2009.5356565. URL https://www.eurecom.fr/publication/2850.

[7] Justin Ma, Kirill Levchenko, Christian Kreibich, Stefan Savage, and Geoffrey M. Voelker. Unexpected means of protocol inference. In *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement*, IMC '06, pages 313–326, New

York, NY, USA, 2006. ACM. ISBN 1-59593-561-4. doi: 10.1145/1177080.1177123. URL http://doi.acm.org/10.1145/1177080.1177123.

[8] Gilbert Wondracek, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, and Scuola Superiore S Anna. Automatic network protocol analysis. In *NDSS*, volume 8, pages 1–14, 2008.

[9] Jian-Zhen Luo and Shun-Zheng Yu. Position-based automatic reverse engineering of network protocols. *Journal of Network and Computer Applications*, 36(3):1070 – 1077, 2013. ISSN 1084-8045. doi: http://dx.doi.org/10.1016/j.jnca.2013.01.013. URL http://www.sciencedirect.com/science/article/pii/S1084804513000222.

[10] Xiangdong Li and Li Chen. A survey on methods of automatic protocol reverse engineering. In Yuping Wang, Yiu ming Cheung, Ping Guo, and Yingbin Wei, editors, *CIS*, pages 685–689. IEEE, 2011. ISBN 978-1-4577-2008-6. URL http://dblp.uni-trier.de/db/conf/cis/cis2011.html#LiC11.

[11] 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*, SS'07, pages 14:1–14:14, Berkeley, CA, USA, 2007. USENIX Association. ISBN 111-333-5555-77-9. URL http://dl.acm.org/citation.cfm?id=1362903.1362917.

[12] S. Gorbunov and A. Rosenbloom. Autofuzz: Automated network protocol fuzzing framework. *IJCSNS*, 10(8):239, 2010. URL http://paper.ijcsns.org/07_book/201008/20100836.pdf.

[13] Maxim Shevertalov and Spiros Mancoridis. A reverse engineering tool for extracting protocols of networked applications. In *Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on*, pages 229–238. IEEE, 2007.

[14] João Antunes, Nuno Neves, and Paulo Verissimo. Reverse engineering of protocols from network traces. In *Reverse Engineering (WCRE), 2011 18th Working Conference on*, pages 169–178. IEEE, 2011.

[15] Young-Woo Seo and Katia Sycara. Text clustering for topic detection. Technical Report CMU-RI-TR-04-03, Robotics Institute, Pittsburgh, PA, January 2004.

[16] Pengtao Xie and Eric P. Xing. Integrating document clustering and topic modeling. *CoRR*, abs/1309.6874, 2013.

[17] Christian Wartena and Rogier Brussee. Topic detection by clustering keywords. In *Proceedings of the 2008 19th International Conference on Database and Expert*

Systems Application, DEXA '08, pages 54–58, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3299-8. doi: 10.1109/DEXA.2008.120. URL http://dx.doi.org/10.1109/DEXA.2008.120.

[18] Tom Van Court and Martin C. Herbordt. Families of fpga-based accelerators for approximate string matching. *Microprocess. Microsyst.*, 31(2):135–145, March 2007. ISSN 0141-9331. doi: 10.1016/j.micpro.2006.04.001. URL http://dx.doi.org/10.1016/j.micpro.2006.04.001.

[19] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *J. Mol. Biol.*, 147(1):195–197, Mar 1981.

[20] T. Lassmann and E. L. Sonnhammer. Kalign–an accurate and fast multiple sequence alignment algorithm. *BMC Bioinformatics*, 6:298, 2005.

[21] G. Kucherov, K. Salikhov, and D. Tsur. Approximate String Matching using a Bidirectional Index. *ArXiv e-prints*, October 2013.

[22] Andrejs Jermakovics, Alberto Sillitti, and Giancarlo Succi. Mining and visualizing developer networks from version control systems. In *Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering*, CHASE '11, pages 24–31, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0576-1. doi: 10.1145/1984642.1984647. URL http://doi.acm.org/10.1145/1984642.1984647.

[23] Y. Chen, A. Wan, and W. Liu. A fast parallel algorithm for finding the longest common sequence of multiple biosequences. *BMC Bioinformatics*, 7 Suppl 4:S4, 2006.

[24] L. Bergroth, H. Hakonen, and T. Raita. A survey of longest common subsequence algorithms. In *Proceedings of the Seventh International Symposium on String Processing Information Retrieval (SPIRE'00)*, SPIRE '00, pages 39–, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0746-8. URL http://dl.acm.org/citation.cfm?id=829519.830817.

[25] Tao Liu, Shengping Liu, Zheng Chen, and Wei-Ying Ma. An evaluation on feature selection for text clustering. In Tom Fawcett and Nina Mishra, editors, *ICML*, pages 488–495. AAAI Press, 2003. ISBN 1-57735-189-4. URL http://dblp.uni-trier.de/db/conf/icml/icml2003.html#LiuLCM03.

[26] Salem Alelyani, Jiliang Tang, and Huan Liu. Feature selection for clustering: A review. *Data Clustering: Algorithms and Applications*, page 29, 2013.

[27] Joseph Jupin, Justin Y. Shi, and Zoran Obradovic. Understanding cloud data using approximate string matching and edit distance. In *SC Companion*, pages 1234–1243. IEEE Computer Society, 2012. ISBN 978-1-4673-6218-4. URL http://dblp.uni-trier.de/db/conf/sc/sc2012c.html#JupinSO12.

[28] M. Steinbach, G. Karypis, and V. Kumar. A comparison of document clustering techniques. Technical Report 00-034, University of Minnesota, 2000. URL /brokenurl#http://publication.wilsonwong.me/load.php?id=233281699.

[29] J. Pevsner. *Bioinformatics and Functional Genomics*. Wiley, 2009. ISBN 9780470451489. URL https://encrypted.google.com/books?id=Emr0ZABQUAIC.

[30] Y. Loewenstein, E. Portugaly, M. Fromer, and M. Linial. Efficient algorithms for accurate hierarchical clustering of huge datasets: tackling the entire protein space. *Bioinformatics*, 24(13):i41–49, Jul 2008.

[31] Sonia Leach and Larry Hunter. Comparative study of clustering techniques for gene expression microarray data. *Currents in computational molecular biology. Universal Academy Press, Tokyo*, pages 1–2, 2000.

[32] Khaled Hammouda and Fakhreddine Karray. A comparative study of data clustering techniques. *Fakhreddine Karray University of Waterloo, Ontario, Canada*, 2000.

[33] Mohammad-Reza Feizi-Derakhshi, Elnaz Zafarani, et al. Review and comparison between clustering algorithms with duplicate entities detection purpose. *International Journal of Computer Science & Emerging Technologies*, 3(3), 2012.

[34] Manuel Martín-Merino and Alberto Muñoz. Visualizing asymmetric proximities with som and mds models. *Neurocomput.*, 63:171–192, January 2005. ISSN 0925-2312. doi: 10.1016/j.neucom.2004.04.010. URL http://dx.doi.org/10.1016/j.neucom.2004.04.010.

[35] P. Drineas, Alan Frieze, Ravi Kannan, Santosh Vempala, and V. Vinay. Clustering in large graphs and matrices. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '99, pages 291–299, Philadelphia, PA, USA, 1999. Society for Industrial and Applied Mathematics. ISBN 0-89871-434-6. URL http://dl.acm.org/citation.cfm?id=314500.314576.

[36] Alberto Muñoz and Manuel Martín-Merino. Visualizing asymmetric proximities with mds models. In *ESANN*, pages 51–58, 2003. URL http://dblp.uni-trier.de/db/conf/esann/esann2003.html#MunozM03.

[37] J.B. Kruskal. Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis. *Psychometrika*, 29(1):1–27, 1964.

[38] Naohito CHINO. Asymmetric multidimensional scaling. *Journal of the Institute for Psychological and Physical Science*, 3(1):101–107, 2011.

[39] Glenn Milligan and Martha Cooper. An examination of procedures for determining the number of clusters in a data set. *Psychometrika*, 50(2):159–179, June 1985. URL http://ideas.repec.org/a/spr/psycho/v50y1985i2p159-179.html.

[40] Arantza Casillas, MT González De Lena, and R Martínez. Document clustering into an unknown number of clusters using a genetic algorithm. In *Text, Speech and Dialogue*, pages 43–49. Springer, 2003.

[41] Taeho Jo and Malrey Lee. The evaluation measure of text clustering for the variable number of clusters. In *Advances in Neural Networks–ISNN 2007*, pages 871–879. Springer, 2007.

[42] Eep Mane, James Kang, Shashi Shekhar, Jaideep Srivastava, Carson Murray, Anne Pusey, Eep Mane, James Kang, Shashi Shekhar, Jaideep Srivastava, and Anne Pusey. Identifying clusters in marked spatial point processes: A summary of results. Technical report, University of Minnesota, 2006.

[43] Yipeng Wang, Zhibin Zhang, Danfeng Daphne Yao, Buyun Qu, and Li Guo. Inferring protocol state machine from network traces: A probabilistic approach. In *Proceedings of the 9th International Conference on Applied Cryptography and Network Security*, ACNS'11, pages 1–18, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-21553-7. URL http://dl.acm.org/citation.cfm?id=2025968.2025970.

[44] Paolo Frasconi and Yoshua Bengio. An em approach to grammatical inference: input/output hmms. In *Pattern Recognition, 1994. Vol. 2-Conference B: Computer Vision &amp; Image Processing., Proceedings of the 12th IAPR International. Conference on*, volume 2, pages 289–294. IEEE, 1994.

[45] Sean Whalen, Matt Bishop, and James P Crutchfield. Hidden markov models for automated protocol learning. In *Security and Privacy in Communication Networks*, pages 415–428. Springer, 2010.

[46] Xiao Ming-Ming and Yu Shun-Zheng. Recovering models of network protocol using grammatical inference. *Procedia Engineering*, 15(0):3764 – 3768, 2011. ISSN 1877-7058. doi: http://dx.doi.org/10.1016/j.proeng.2011.08.705. URL http://www.sciencedirect.com/science/article/pii/S1877705811022065. {CEIS} 2011.

[47] Tammo Krueger, Hugo Gascon, Nicole Krämer, and Konrad Rieck. Learning stateful models for network honeypots. In *Proceedings of the 5th ACM Workshop on Security and Artificial Intelligence*, AISec '12, pages 37–48, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1664-4. doi: 10.1145/2381896.2381904. URL http://doi.acm.org/10.1145/2381896.2381904.

[48] Kristie Seymore, Andrew Mccallum, and Roni Rosenfeld. Learning Hidden Markov Model Structure for Information Extraction. In *AAAI 99 Workshop on Machine Learning for Information Extraction*, 1999. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.47.8365.

[49] Benjamin Murrell and Jules Raymond Tapamo. Heuristics for state splitting in hidden markov models. *Proceedings of PRASA*, pages 3–8, 2008.

[50] Sajid M. Siddiqi, Geoffrey J. Gordon, and Andrew W. Moore. Fast state discovery for hmm model selection and learning. In Marina Meila and Xiaotong Shen, editors, *AISTATS*, volume 2 of *JMLR Proceedings*, pages 492–499. JMLR.org, 2007. URL http://dblp.uni-trier.de/db/journals/jmlr/jmlrp2.html#SiddiqiGM07.

[51] Dayne Freitag and Andrew McCallum. Information extraction with hmm structures learned by stochastic optimization. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, pages 584–589. AAAI Press, 2000. ISBN 0-262-51112-6. URL http://dl.acm.org/citation.cfm?id=647288.723414.

[52] Raymond C Vasko Jr, Amro El-Jaroudi, and J Robert Boston. An algorithm to determine hidden markov model topology. In *Acoustics, Speech, and Signal Processing, 1996. ICASSP-96. Conference Proceedings., 1996 IEEE International Conference on*, volume 6, pages 3577–3580. IEEE, 1996.

[53] Gideon Schwarz. Estimating the dimension of a model. *The annals of statistics*, 6(2):461–464, 1978. URL http://scholar.google.com/scholar.bib?q=info:sskrpr5jlLwJ:scholar.google.com/&output=citation&hl=en&as_sdt=2000&ct=citation&cd=0.

[54] Takehisa Yairi, Masahito Togami, and Koichi Hori. Learning topological maps from sequential observation and action data under partially observable environment. In Mitsuru Ishizuka and Abdul Sattar, editors, *PRICAI*, volume 2417 of *Lecture Notes in Computer Science*, pages 305–314. Springer, 2002. ISBN 3-540-44038-0. URL http://dblp.uni-trier.de/db/conf/pricai/pricai2002.html#YairiTH02.

[55] Annie TT Ying, Gail C Murphy, Raymond Ng, and Mark C Chu-Carroll. Predicting source code changes by mining change history. *Software Engineering, IEEE Transactions on*, 30(9):574–586, 2004.

[56] Jacek Czerwonka, Rajiv Das, Nachiappan Nagappan, Alex Tarvo, and Alex Teterev. Crane: Failure prediction, change analysis and test prioritization in practice – experiences from windows. In *Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, ICST '11, pages 357–366, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4342-0. doi: 10.1109/ICST.2011.24. URL http://dx.doi.org/10.1109/ICST.2011.24.

[57] Alexander Tarvo. Mining software history to improve software maintenance quality: A case study. *IEEE Softw.*, 26(1):34–40, January 2009. ISSN 0740-7459. doi: 10.1109/MS.2009.15. URL http://dx.doi.org/10.1109/MS.2009.15.

[58] Yun Yan. Python: Smith waterman. https://gist.github.com/Puriney/6286305, 2014.

[59] S. Marsland. *Machine Learning: An Algorithmic Perspective*. Taylor & Francis, 2011. ISBN 9781420067194. URL http://books.google.co.uk/books?id=n66O8a4SWGEC.

[60] Y. Tateno, M. Nei, and F. Tajima. Accuracy of estimated phylogenetic trees from molecular data. I. Distantly related species. *Journal of Molecular Evolution*, 18: 387–404+, 1982.

[61] David B Goldstein, A Ruiz Linares, Luigi Luca Cavalli-Sforza, and Marcus W Feldman. An evaluation of genetic distances for use with microsatellite loci. *Genetics*, 139(1):463–471, 1995.

[62] N. Takezaki and M. Nei. Genetic distances and reconstruction of phylogenetic trees from microsatellite DNA. *Genetics*, 144(1):389–399, Sep 1996.

[63] T. F. Smith, M. S. Waterman, and W. M. Fitch. Comparative biosequence metrics. *J. Mol. Evol.*, 18(1):38–46, 1981.

[64] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403 – 410, 1990. ISSN 0022-2836. doi: http://dx.doi.org/10.1016/S0022-2836(05)80360-2. URL http://www.sciencedirect.com/science/article/pii/S0022283605803602.

[65] Therese Berg, Bengt Jonsson, and Harald Raffelt. Regular inference for state machines with parameters. In *Fundamental Approaches to Software Engineering*, pages 107–121. Springer, 2006.

[66] Miguel Bugalho and Arlindo L. Oliveira. Inference of regular languages using state merging algorithms with search. *Pattern Recogn.*, 38(9):1457–1467, September 2005. ISSN 0031-3203. doi: 10.1016/j.patcog.2004.03.027. URL http://dx.doi.org/10.1016/j.patcog.2004.03.027.

[67] Neil Walkinshaw, Kirill Bogdanov, Mike Holcombe, and Sarah Salahuddin. Reverse engineering state machines by interactive grammar inference. In *Proceedings of the 14th Working Conference on Reverse Engineering*, WCRE '07, pages 209–218, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-3034-6. doi: 10.1109/WCRE.2007.45. URL http://dx.doi.org/10.1109/WCRE.2007.45.

[68] Axel Cleeremans, David Servan-Schreiber, and James L. McClelland. Finite state automata and simple recurrent networks. *Neural Comput.*, 1(3):372–381, September 1989. ISSN 0899-7667. doi: 10.1162/neco.1989.1.3.372. URL http://dx.doi.org/10.1162/neco.1989.1.3.372.

[69] Muhammad Rafi, M. Shahid Shaikh, and Amir Farooq. Article:document clustering based on topic maps. *International Journal of Computer Applications*, 12(1):32–36, December 2010. Published By Foundation of Computer Science.