An Automatic Approach of Reverse Engineering the Protocols of Networked Applications

A Thesis
Submitted to the Faculty
of
Drexel University
by
Maxim Shevertalov
in partial fulfillment of the
requirements for the degree
of
Master of Science in Computer Science
2007
Table of Contents

List of Figures ................................................................. 3
Abstract .............................................................. 5
1. Introduction ............................................................ 6
   1.1 Overview .......................................................... 6
   1.2 Motivation .......................................................... 8
      1.2.1 Protocol Creation ........................................ 8
      1.2.2 Protocol Debugging .................................... 9
      1.2.3 Protocol Reverse Engineering ......................... 9
   1.3 Summary ............................................................ 10
2. Background ............................................................. 11
   2.1 Protocol Development ......................................... 11
   2.2 Packet Capturing ............................................... 13
   2.3 Clustering ........................................................ 14
   2.4 String Matching ............................................... 16
   2.5 Summary .......................................................... 17
3. Technique ............................................................... 18
   3.1 Data Collection ................................................. 18
   3.2 Algorithm ........................................................ 18
   3.3 Clustering ........................................................ 19
      3.3.1 Distance Metric ......................................... 19
      3.3.2 TerminationCriterion .................................. 21
   3.4 State Selection .................................................. 22
   3.5 FSM Graph Formation .......................................... 24
   3.6 PEXT Tool Implementation .................................... 28
   3.7 Summary .......................................................... 29
4. Case Study .............................................................. 30
   4.1 Introduction ...................................................... 30
## List of Figures

1.1 PEXT Work Flow .......................................................... 7
2.1 Reflexion Example .................................................... 12
2.2 Dendrogram Example .................................................. 15
2.3 Longest Common Substring via Dynamic Programming Example .................................................. 16
3.1 Cluster Distance Metric .................................................. 20
3.2 State Identification ..................................................... 23
3.3 Initial Graph Construction ............................................. 25
3.4 Graph Transformation .................................................. 27
3.5 PEXT: Graph Tab ....................................................... 28
4.1 Documented Jabber FSM ............................................. 31
4.2 Extracted Login FSM ................................................... 34
4.3 Extracted Send Messages FSM without the “pull-out” method .................................................. 36
4.4 Extracted Send Messages FSM ....................................... 37
4.5 Extracted Receive Messages FSM .................................... 38
4.6 Extracted Final Jabber FSM without the “pull-out” method .................................................. 40
4.7 Extracted Final Jabber FSM ........................................... 41
4.8 Extracted Jabber FSM Comparison ................................ 43
4.9 FTP Login Sequence ................................................... 45
4.10 FTP Login State Machine ............................................. 46
4.11 FTP Login: Extracted State Machine ............................... 47
4.12 FTP Documented States .............................................. 49
4.13 Fetch Extracted States .................................................. 51
4.14 Cyberduck Extracted States ................................................................. 53
Abstract
An Automatic Approach of Reverse Engineering the Protocols of Networked Applications
Maxim Shevertalov
Advisor: Spiros Mancoridis

As networked application become more relevant in today’s computing world, the tools we use to create, debug, and inter-operate networked protocols need to become more sophisticated. This work poses an initial step in building a protocol reverse engineering tool. This tool can be used to help developers understand and implement communication protocols. It can assist them in reverse engineering protocols to make applications compatible, as well as merging multiple protocols.

This document describes a tool named PEXT, used to extract networked protocols from captured network data. It will present implementation details of PEXT as well as two case studies using it. The first case study will use PEXT to extract the Jabber chatting protocol from a collection of captured network traffic, generated using the Pidgin chat client application. The second case study will use PEXT to investigate the differences between two implementations of the FTP protocol.
1. Introduction

The role of network protocols becomes more important as software becomes more connected. A protocol is an agreement between two entities wishing to communicate. It defines a standard language to support communication between software that is distributed. The Internet is designed as a layer of protocols. For example, the OSI Reference Model [37] defines a high level layered structure that is followed in packet switched networks used today, most notably IP networks. The lowest layer of the protocol stack is used to transmit actual bits across the network cable, while the highest layer is responsible for application specific purposes. The application layer of the stack that is populated with the majority of network protocols. Each application attempting to use the network will either need to implement one of these pre-defined protocols, or implement its own special purpose protocol. Developers typically choose to create their own application layer protocols.

Networked protocols can be classified in a number of ways, one of which is as either stateless, or statefull. For example, HTTP is a stateless protocol. The server answers client requests without saving any client information. Requests are handled independently of the results of previous queries. FTP is an example of a statefull protocol, as it remembers the user logged into the server and can control what files that user has access to. There are distinct states in FTP conversations, including login, log out, and file transfer requests.

1.1 Overview

This document presents a tool, PEXT, to reverse engineering a networked protocol by using captured network traffic generated by an implementation of that protocol. A high level overview of this approach can be found in Figure 1.1. The process begins by capturing traffic from multiple, sufficiently different, execution traces of an application implementing the protocol. The traffic is stored in the form of packets, where each packet represents a message either sent or received by the client. The packets are then clustered into several classes using a similarity metric. PEXT then creates a linear sequence diagram for each of the execution traces. Each diagram is composed of strings of packet classes. Finally, the execution diagrams
Figure 1.1: An overview of the process used by PEXT. (a) Capture packets from execution traces representing specific features (Section 3.1). (b) Cluster all the involved packets into classes (Section 3.3). (c) Produce execution flow graphs for each execution trace by using string matching to find states (Section 3.4). (d) Combine individual flow diagrams into a single, representative, Finite State Machine (FSM).
are merged into a single Finite State Machine (FSM) diagram. This FSM captures the features used in the execution traces, and thus can be different depending on which features were executed. All of this is described in more detail in Chapter 3.

1.2 Motivation

1.2.1 Protocol Creation

One has to use an application layer protocol to create applications that can communicate over the network. Networked applications have become commonplace, and there are several libraries, for various programming languages, to help with their development. As a result, many developers create network components without realizing that they are creating an application layer protocol. These protocols are not designed, but are simply created as the need arises during development.

A developer may approach protocol creation in a number of ways, the most common being creating one from scratch. However, if developers are aware of other protocols that already perform similar tasks, they can implement one of them, thus saving the cost of design. Developers may also find ready made implementations of the protocol and use those, thus saving even more time and cost. Also, by using protocols that already exist, developers will be able to create applications that can interoperate.

While there are numerous advantages to using pre-existing protocols in applications, one disadvantage is having to understand that protocol. A tool that can assist developers in such an understanding, by observing the target protocol in action and presenting developers with a state diagram of that protocol, would be useful. This, in addition to protocol specification, will help developers understand the protocol they are implementing.

Another consideration with implementing a pre-existing, is the scope of such a protocol. For example, one may wish to take an existing protocol and remove some extraneous functionality that is not needed in the current application. The developer needs to know which states can be removed safely and which are needed for the protocol to function correctly. PEXT is able to extract features of interest to the developer by observing those features in action. This will enhance developers’ understanding of those features and assist them in narrowing the scope of the protocol they wish to implement.
1.2.2 Protocol Debugging

Once a protocol is created, developers need a way of debugging it. There are a number of tools and techniques to help them do so. However, they all rely on a high level understanding of the designed protocol state machine [7, 8, 18, 24, 43, 44, 63]. These techniques use the protocol’s designed finite state machine to generate test sequences, which are then executed on the protocol’s implementation.

What we propose is a system more akin to code verification through reflexion techniques found in software engineering research [6, 12, 22]. Using our techniques, the developer would reverse engineer the underlying state machine and compare it to a documented one. This would not only point out inconsistencies in the implementation, but also highlight any missing cases in the testing process, since the generated FSM would have missing states if they were not tested.

Our techniques could also help developers reduce the number of generated test sequences used in testing. For example, the developer could use a large number of tests to generate a finite state machine, which then could be used to create a minimal test sequence to cover the entire FSM.

1.2.3 Protocol Reverse Engineering

There are a number of reasons why one would want to reverse engineer a networked protocol. One reason is interoperability, or being able to communicate with a number of different applications. Examples of reverse engineering techniques applied to the networked applications domain can be found in projects such as those that were created to communicate with the AOL Instant Messenger (AIM) client [49]. The AIM protocol is not known. However a number of people have been able to reverse engineer it and provide tools such as GAIM [27], Trillian [65], and Adium [2]. It was not easy to reverse engineer the protocol and some parts, for example file sharing, are still not functioning correctly for every project.

Other reasons for reverse engineering a protocol includes developing tools that can help the protocol achieve what it is trying to do, identify a particular protocol, or to improve the protocol through packet shaping and traffic modification [23, 40, 45, 69]. By being able to track a protocol through its states, one could manage the traffic going across the network and schedule it to be fair to all users on the network. For example, if one knows that a protocol is about to enter a heavy data transfer phase, one could make the decision to restrict the user’s throughput in that phase.
PEXT is applicable in these scenarios because it can provide the necessary state information about a protocol to the traffic shaper. It can find loops in the protocol to determine which parts are executed the most and it can keep track of packet sizes to study the throughput of each state in the FSM. This provides enough information to make intelligent decisions about traffic on the network and the users generating it.

1.3 Summary

Networked applications are becoming increasingly relevant. Therefore the protocols underlining those applications are important. Since there are numerous tools that make it easier for developers to include a networking component in their code, an increasing number of developers choose to do so. As development efforts are accelerated, there is a need for more tools to assist with the creation, design, and debugging of these protocols. Significant effort goes into creating a new protocol, because it is difficult to understand and reuse an existing one. Therefore, a tool is needed to help developers understand existing protocols. Developers may also wish to create software that can communicate with legacy applications. To accomplish this task they may need to reverse engineer an existing protocol in order to communicate with the legacy application. PEXT can help them achieve this task.

This document consists of the following chapters. Chapter 2 covers background information such as: clustering, protocol development and testing. Chapter 3 describes our approach to reverse engineering the protocols of legacy networked applications. Chapter 4, presents two case studies; a chat client implementing the Jabber protocol and a comparison of two different FTP clients. The process and results for each case study are presented. Chapter 5, concludes with a summary of results, lessons learned, potential uses of this work, and ideas for future research related to this work.
2. Background

This chapter provides a general overview of a number of techniques that are used later in the thesis. First, it presents the current state of protocol development (Section 2.1), including protocol creation, debugging, and reverse engineering for interoperability. It also presents current technologies in packet interception and capture (Section 2.2), and agglomerate clustering (Section 2.3) techniques that are used in this approach. Following that, this chapter presents an overview of various pattern machining techniques (Section 2.4) that pertain to the approach used in this work.

2.1 Protocol Development

Developers of networked applications have to design, implement, and debug their protocols. The protocol design process can vary depending on the requirements of the application. It can be a formal process, such as, for example, when designing a general protocol to be implemented by different applications. It can also entail reverse engineering if the application uses an existing protocol. The design process may also involve modifying an existing protocol to meet the requirements of the application undergoing development. Finally, the protocol may be too simple to warrant a formal/rigorous design process.

Using a formal process to design a protocol has a number of advantages. If one is precise in the definition of the protocol, developers can use model checking techniques [4, 33, 55] to prove the correctness of the protocol [29, 48, 54]. They can also use the designed model to create test cases used in testing and debugging implementations. They can also use reflexion techniques (Figure 2.1) to compare an extracted protocol model with the designed one to see how close the implemented version of the protocol matches its design. A formal specification of a protocol also provides developers with more options when it comes to the implementation and debugging of the protocol such as a variety of automation tools and environments [11, 35].

Protocol development is a process analogous to software development. While previously there was a greater emphasis on a rigid design process [10, 30, 46, 61, 64], currently new technologies and methods are available that allow for early prototyping [25, 51, 68]. Several environments [11, 35] and network
The left side is an example of a protocol design. The right side is an example of a model extracted from an implementation. Node A from the design has been mapped to nodes a, b, and c. Node C has been mapped to nodes d and e. Node D has been mapped to node f. Node B has no mapping in the extracted design, and node g was not mapped to anything. This diagram tells developers that the protocol never enters state B during execution. This may be due to a lack of a test case or an error in implementation. In addition there is an extracted state g that has no mapping in the design. This again indicates an error in implementation or an omission in the design. The reflexion exercise allows developers to catch implementation deviation early.
simulators [38, 50, 62] for protocol development have been created. These allow developers to prototype and test their designs before moving into the development process. Thus catching bugs and problems early, when they can still be easily corrected.

Frequently, developers want to add small networking components to their applications. For these purposes they may not wish to spend the effort of going through a formal specification and design process. Instead, developers may want to modify an existing protocol or build their own. In these cases, there exist a number of implemented protocols and libraries that developers can use. For example Java [66] implements the TCP and UDP transport layer protocols. These implementations make the underlying message passing transparent to the developer. Thus, Java enables developers to add networking features to their applications quickly. In addition to a transport layer protocol implementations, there are other, higher level, implementations. For example the Extensible Protocol Package [31] enables networked applications to send XML documents across an established connection. Also, XML-RPC [3] is a protocol designed to simulate remote procedure calls across multiple languages.

### 2.2 Packet Capturing

There are numerous tools and methods to capture network data. Some of the most popular methods are based on the pCap library [39]. The pCap library is used as a basis for several packet analyzers including TCPDUMP [39], WinPcap [15, 53], jPCap [9], and Ethereal [19]. PCap based analyzers, and other user level based analyzers, have problems keeping up with fast links being filled to their capacity with small packets [16, 59]. They choose to ignore packets in order to keep up with the network traffic.

Due to these issues, a number of alternatives have appeared, which range from customized hardware [16, 36] to modifying operating system drivers [17]. Although these solutions offer more consistent results, they are more difficult to set up and maintain. In addition both user level and the more precise, alternative, solutions have similar performance under normal network conditions. We choose to use a pCap based solution because it is well supported by various libraries and the performance was sufficient for this work.
2.3 Clustering

Clustering is a statistical technique used to aggregate similar objects together. For example, classification of animals is a form of clustering. One can group similar animals into family, genus, and species. This idea of classification can be applied to other fields as well. For example, in software engineering we may want to see how different features of an application are related [42].

There are many different approaches to clustering [5]. For this project we decided to use an agglomerative hierarchical clustering approach [14]. Hierarchical clustering combines elements based on a distance metric. To perform agglomerative hierarchical clustering one must be able to combine multiple elements into a single representative centroid. In the simplest case, a single element from a cluster can serve as the centroid. However, a more common approach involves taking an average of all elements in a cluster to create a representative centroid. A way to visualize such a clustering can be done via a dendrogram. Figure 2.2 presents an example of agglomerative hierarchical clustering. The process begins with each element as its own cluster. At each step, two of the closest clusters are joined together to form a new cluster. Figure 2.2 begins with seven separate elements. At step (1) elements A and B are combined to form a new centroid. After 6 steps all elements are clustered together into a single, unified cluster. In this case a cutoff value was chosen after 4 steps. It was chosen either because exactly three clusters were needed, or because adding F to the (A B C D E) cluster would have made that cluster’s diameter too large.

One problem with this method, as well as other clustering methods, is knowing what is the correct number of clusters. For example, when using k-means clustering [32] one has to chooses the $k$, or the number of clusters required. On the other hand, when using Quality Threshold clustering [32], the user specifies the maximum diameter of a cluster.

When selecting a good cut-off point, one can use human input to look at the dendrogram of clustered data and decide when enough clustering has occurred [60]. On the other hand, one can use the L-method to estimate this automatically [58]. The L-method creates a ratio of the number of clusters and a quality metric. It then attempts to find the elbow of this graph and chooses the corresponding number of clusters.
Agglomerative hierarchical clustering begins with each element as its own cluster. At each step, two of the closest clusters join together to form a new cluster. This example begins with seven separate elements. At step (1) elements A and B are combined to form a new centroid. After 6 steps all elements are clustered together into a single, unified cluster. A cutoff value was chosen after 4 steps. It was chosen either because exactly three clusters were needed, or because adding F to the (A B C D E) cluster would have made that cluster’s diameter too large.
This example compares two words $A = abbac$ and $B = cbacab$. The table is filled by first comparing $A_0$ to $B_0$, followed by $A_1$ to $B_0$, and so on until $A$ and $B$ are exhausted. Each $T_{i,j}$ is equal to 0 if $A_i \neq B_j$. $T_{i,j} = T_{i-1, j-1} + 1$ if $A_i = B_j \land i > 0 \land j > 0$. Or $T_{i,j} = 1$ otherwise. Once the table is complete, the algorithm returns the largest value and its location. Thus, the largest substring is three characters long and can be deduced from $A_2\ldots4 = bac$.

### 2.4 String Matching

This work uses two similar dynamic programming algorithms. One for finding the longest common substring [1] and the other for finding the longest common subsequence between two strings [1]. The common subsequence algorithm is central to the distance function used in clustering. The common substring algorithm is central to the state selection portion of the project. Both of these algorithms execute in $nm$ time where $n$ and $m$ are sizes of two strings.

The longest common substring algorithm, similar to other dynamic programming algorithms, iteratively constructs a table (Figure 2.3). Given two strings $A$ and $B$, such that their length are $n$ and $m$ respectively, it constructs a table, $T$, of size $n \times m$ such that:
Once the table has been constructed one needs to find the largest value inside the table and its coordinates. For example if the largest value is \( x \), at \( i, j \), then the largest substring is of size \( x \) and is equal to \( A_{i-x+1} \ldots i \). This process can be done while the table is being constructed in order to achieve further improvements in performance.

The longest common subsequence algorithm works in a similar manner. It also constructs a table \( T \). However, this table’s size is \( n + 1 \times m + 1 \) and is filled according to the following rule:

\[
T_{i,j} = \begin{cases} 
0 & \text{if } A_i \neq B_i, \\
T_{i-1,j-1} + 1 & \text{if } A_i = B_i \land i \neq 0 \land j \neq 0, \\
1 & \text{otherwise}
\end{cases}
\]

This algorithm works exactly like the common substring algorithm except it does not reset a cell’s value to zero if the compared characters are not equal. Unlike the common substring algorithm, it is more difficult to extract the common subsequence. However, because PEXT uses this to calculate a similarity metric, it is only concerned with the size of the longest common subsequence and thus the problem of extraction does not arise.

2.5 Summary

This section covered several different tools that PEXT uses to extract protocols from collected traffic. It has covered how traffic is captured and the types of operations PEXT will be applying on that traffic to extract useful information. Chapter 3 details how these tools are used when extracting FSMs that represent application level network protocols.
3. Technique

PEXT is a tool that automates our approach to reverse engineering a protocol from a collection of network traffic. The process begins by capturing data from multiple, sufficiently different, execution traces of an application that implements the protocol (Section 3.1). It then uses this captured information to extract a representative Finite State Machine (FSM) (Section 3.2). This FSM codifies the features of the protocol that are used in the execution traces. Note that the FSM can be different depending on which features were exercised.

3.1 Data Collection

This work uses tools based on the libPCap library [39] to collect network traffic. LibPCap has a number of implementations in a variety of languages. There are also several applications that can save traffic data in libPCap format. This work uses TCPDump [39] as the main packet capturing application. TCPDump is able to filter captured traffic. When using PEXT to reverse engineering a protocol, one needs to capture all of the packets sent and received by one node in the networked system. Moreover, TCP acknowledgment packets need to be filtered because they represent TCP-specific conventions and are not a part of the application layer protocol.

During the data capture, one must to be careful to design test cases that reflect the typical usage of the application being studied. If, for example, one does not vary the server names, user names, and passwords for login into FTP servers, PEXT would assume that the provided names were part of the protocol. PEXT is able to capture how a protocol is used in an application, not simply what protocol commands are issued.

3.2 Algorithm

Once network traffic data has been captured, PEXT processes it using the following algorithm:

1. Cluster all of the captured packets based on a user-specified cutoff point (Section 3.3).
2. Label each packet with its cluster ID so that each test stream becomes a sequence of cluster IDs.
3. Separate each test case sequence of packets into individual flows based on the addressing information of packets.
4. Extract states by labeling individual flows that contain identical sequences of clustered packets with the same state ID (Section 3.4).
5. Extract additional states by finding longest common substrings with a length greater than two within all of the leftover streams. We ensure that states do not contain packets from more than one flow (Section 3.4).
6. Label each remaining packets as a separate state (Section 3.4).
7. Create graphs for each of the test streams (Section 3.5).
8. Merge all of the graphs into a single FSM (Section 3.5).
9. Use the “pull-out” method to guarantee that each transition is deterministic (Section 3.5).
10. Use the “pull-in” method to combine lists of states to simplify and improve the readability of the FSM (Section 3.5).

This algorithm stems from our intuition that a state in a protocol can be thought of as a collection of packets representing a conversation between networked entities. Thus, by first clustering individual packets, we are able to determine the vocabulary of these conversations. Then, by observing how this vocabulary is used, we are able to extract different conversations and represent them as states in the FSM.

3.3 Clustering

We chose to use agglomerate hierarchical clustering [14] to group packets into separate classes. This clustering approach requires a function to calculate the distance measure (Section 3.3.1), and a termination criterion to calculate when clustering is complete (Section 3.3.2). While the rest of the paper describes our technique using only automated clustering, PEXT also allows users to cluster the packets manually.

3.3.1 Distance Metric

A packet traveling across a network connection can be thought of as a collection of bits following a pattern. Therefore, when calculating the distance (i.e., difference) between any two packets we first attempted to perform our measurement by comparing individual bits. However, we found that grouping them
Figure 3.1: This example is designed to show how the distance between clusters is calculated in PEXT. The distance between cluster 1 and 2 is the smallest distance between any two elements \( a \in A \) and \( b \in B \). For example \( D_1 \), or the distance between clusters 1 and 3, is the distance between elements \( b \) and \( f \).

Irrespective of the granularity, PEXT uses a dynamic programming approach, described in Chapter 2, to find the longest common sub-sequence (LCSS)[1] and calculate the distance between any two packets, \( a \) and \( b \). The comparisons are normalized using the following formula:

\[
D(a, b) = 1 - \frac{\text{length}(\text{LCSS}(a, b))}{\max(\text{length}(a), \text{length}(b))}
\]  

(3.1)

into bytes and comparing at that level of granularity provided similar results and significantly improved performance. The final version of PEXT allows the user to choose this granularity.
where \( D(a, b) \) is the distance between packets \( a \) and \( b \). This distance metric satisfies the triangle inequality property. At each step of the clustering process, two clusters with the smallest distance between them are grouped to form a new cluster. The distance between two clusters is the minimum distance between any two elements from each of the clusters (Figure 3.1).

### 3.3.2 Termination Criterion

PEXT stops clustering when the next two elements to cluster together have a distance greater than some specified parameter depending on the protocol. It uses this as a termination criterion because during the clustering process, at each step, the distance will stay the same or increase. Meaning \( D_i \leq D_{i+1} \), where \( D_i \) is the smallest distance at the \( i \)'s clustering step and \( D_{i+1} \) is the smallest distance at the clustering step immediately following.

At the end of the clustering step \( i \) involving two clusters \( a_i \) and \( b_i \), \( a_i \) and \( b_i \) have merged into \( c_i \). There are two possibilities at clustering steps, \( i \) and \( i+1 \). Either \( a_{i+1} \neq c_i \cap b_{i+1} \neq c_i \) or \( a_{i+1} = c_i \bigoplus b_{i+1} = c_i \).

To show that the initial statement, \( D_i \leq D_{i+1} \), is true, we have to show that it is true in both scenarios.

The first scenario is straightforward. If we assume that the initial statement is false, meaning \( D_i > D_{i+1} \), then it must be true that the distance between \( a_i \) and \( b_i \) is greater than the distance between \( a_{i+1} \) and \( b_{i+1} \). This however cannot be true because in that case \( a_{i+1} \) and \( b_{i+1} \) would have been chosen at step \( i \), not at step \( i + 1 \). Therefore \( D_i \leq D_{i+1} \).

To show that \( D_i \leq D_{i+1} \) holds in the second scenario we can perform a similar proof. The distance function is defined as the smallest distance between all members of each cluster being compared. Therefore if it was true that \( D_i > D_{i+1} \), it would imply that for some \( x_i \in a_i \) and \( y_i \in b_i \) the distance between them was less than the distance between any other elements in \( a_i \) and \( b_i \). And that the distance between \( x_i \) and \( y_i \) was greater than the distance between \( x_{i+1} \) and \( y_{i+1} \) or the elements used to calculated the distance at clustering step \( i + 1 \). However if the distance between \( x_{i+1} \) and \( y_{i+1} \) was smaller than the distance between \( x_i \) and \( y_i \), then the clustering algorithm would have chosen those clusters that contain \( x_{i+1} \) and \( y_{i+1} \) as step \( i \). Therefore \( D_i \leq D_{i+1} \).

Because of this relationship, the evaluation criteria can guarantee that no clusters are closer than the specified threshold. Therefore we can incorporate a similarity threshold into the tool to adopt it to different reverse engineering projects. This allows one to cluster applications that use verbose control commands as
well as ones that are less so.

PEXT leaves the maximum distance at which clustering occurs up to the user, because different protocols have different degrees of verbosity. For example, the FTP [52] protocol is composed of short command strings, and thus has a different maximum distance than the Jabber [21] protocol, which is composed of the more verbose XML messages.

Once the clustering process is complete, each packet is labeled with a cluster ID. Therefore, each test stream of packets is represented as a string of cluster IDs.

3.4 State Selection

At this stage in the process PEXT is presented with a number of test cases that are represented as strings of cluster IDs. It begins by separating each test stream into its individual flows. A flow is a continuous stream of packets that share the same source and destination addressing information. A number of protocols use separate flows for different aspects of their operation. For example, FTP uses separate flows for command and bulk data transfers.

Thus, each test stream is split into flows at each point where the packet addressing information changes. PEXT maintains flow order, meaning that if packet $p_j$ came after packet $p_i$ in the original stream, $p_j$ also follows $p_i$ in the newly split stream. This ensures that packets belonging to the same state are from the same flow.

In the next step we find identical flows. These flows form the initial states of the FSM. Because PEXT restricts each state to contain only packets of the same flow, this is a natural way of finding distinct states.

Once identical flows have been identified and labeled as states, PEXT applies the longest common substring algorithm [1] to all leftover flows to discover the remaining states. It attempts to find common, continuous strings across the entire set of provided test cases. PEXT also restricts each state to contain at least two packets. At the end of this process, each packet not yet belonging to a state, becomes a single packet in its own state.

Figure 3.2 demonstrates an example of this process. There are two flows interlaced inside each of the test cases. PEXT first splits each of the streams into sub-streams based on where the protocol transitioned between flows. It then derives states $s_1$ and $s_2$ because each of the sub-streams contained within those
Figure 3.2: (a) The initial streams of test data as clustered packets. Each number represents the cluster ID of a particular packet. Light gray packets are part of the data flow, while white packets are part of the control flow. (b) Split the packet stream at each point where the stream transitions from one flow to another. (c) Label flow duplicates as the initial states of the FSM. (d) Label all other sequences of cluster IDs as states by finding longest continuous sequences first, until no sequences of two or more packets are found. Label the leftover individual packets as their own, unique, states.
states has a duplicate sub-stream. At this point each sub-stream can be thought of as a string where the symbols are cluster IDs. PEXT searches for the largest common non-overlapping substring (of length greater than two symbols) and combines them into states. In Figure 3.2 it finds three common substrings: \((1, 4, 1, 1)\), \((1, 5, 1, 1)\), and \((1, 2)\). These sequences become states \(s3\), \(s4\) and \(s5\) respectively. Thus, PEXT groups the first and second packets from test case 1 into state \(s5\), the third, fourth, fifth, and sixth packets from test case 1 into state \(s3\), and the eighth, ninth, tenth, and eleventh packets from test case 1 into state \(s4\). At the end of this process it groups each of the remaining packets into individual states, even if they have the same cluster IDs. Thus, even though the twelfth packet from test case 1 and the twelfth packet from test case 2 share the same cluster ID \((i.e., 9)\), they are grouped into their own, individual, states because the common substring is of length one.

### 3.5 FSM Graph Formation

Graph formation is broken into two steps. First PEXT combines all of the individual test stream graphs into a single graph. It then applies two methods, “pull-in” and “pull-out”, to summarize the graph and make it easier to comprehend. At the end of the process PEXT generates a finite state machine representation of the application’s protocol.

At this stage of the process PEXT has identified the initial states, such that each packet in each of the test streams belongs to a particular state. It begins by converting each test case from a stream of packets into a sequence of states. For example, in Figure 3.3, the initial graphs are formed by transitioning from the representation in Figure 3.3a, that was derived in Figure 3.2, to the representation in Figure 3.3b. Using these sequences PEXT is now ready to extract the FSM.

To extract the FSM, PEXT begins by combining all of the test state sequences into a single graph. From these sequences it can determine where each state can lead to by constructing a hash map with state IDs as keys and sets of state IDs as values. Using this scheme PEXT can iterate over each of the test sequences and fill in the hash map as it goes from state to state. Once the hash map is complete, PEXT draws a graph by drawing edges from every key in the map to all of the associated values. PEXT uses Dot [26] to draw the actual graph.

Thus, the tool is able to derive the graph in Figure 3.3c from the sequences of Figure 3.3b. Note that
Figure 3.3: White boxes are individual states composed of packet streams. Dark gray boxes are the actual packets internal to a particular state represented by their cluster ID. (a) The initial streams with defined states such that each packet of the stream belongs to a single state. (b) The initial graph of states for each test stream. (c) The combination of individual test stream graphs into a single FSM.
the branching points in Figure 3.3c come from the fact that states in the original sequences, transition to multiple different states. For example, s5 has a transition to s3 in test case 1 and to s4 in test case 2. Thus, in the final graph, s5 is a branching state transitioning to either s3 or s4.

Once the initial graph is constructed, PEXT employs two simplifying transformations. Each transition represents a packet, received or sent. Therefore, one needs to guarantee that each transition leaving a state can be uniquely determined. In cases where this uniqueness criterion is violated, PEXT applies a simple “pull-out” method to fix the problem. It forms a new state with the violating string of packets and creates a transition from the violating parent to it. The new state also acquires transitions from itself to the original, violating, children states. Finally it removes the violating stream of packets from the original children states.

This process is demonstrated by Figure 3.4b. It begins with a graph constructed at the end of Figure 3.3. Observe that both states s3 and s4 begin with a packet labeled by cluster ID 1. Because s5 has a transition to both of these states, that transition is non-deterministic. Thus, PEXT pulls out the violating packet with cluster ID 1 and place it in its own state, PS1. It then draws a transition from s5 to PS1 and transitions from PS1 to the original children s3 and s4. One has to be aware of other states that have transitions to s3 or s4. In this case s1 has a transition to both of those states and, thus, PEXIT draws a transition from s1 to PS1. In more complex examples a pulled out state may be identical to a state already in the FSM. In those cases the two states are merged into one.

The “pull-in” method is designed to make the graph more readable by combining nodes that form a list. These are nodes that have a single parent and a single child. Once these parent/child pairs are identified, they can be combined together into a new state. Figure 3.4c demonstrates this process by combining states s6 and s7 into CS1, as well as states s8 and s9 into CS2.

These two transformations produce the final FSM that can be used by reverse engineers for a number of other tasks, including testing, reflexion, and the implementation of new applications. These graphs capture the use cases of different applications and allow designers to see how they may improve the protocol in the future.
Figure 3.4: White boxes are original states extracted from the test streams. Dark gray boxes are the actual packets internal to a particular state represented by their cluster ID. Light gray boxes represent states created through the use of the “pull-out” method. Black boxes represent new states created by the “pull-in” method. (a) The initial graph generated at the end of Figure 3.3. (b) Graph created by extracting packet 1 from states s3 and s4, generating a new state PS1, and connecting it back into the graph. (c) Graph created by combining states s6 and s7 into CS1 and states s8 and s9 into CS2.
3.6 PEXT Tool Implementation

PEXT was created using Ruby [67] for the back end and Cocoa [28] for the graphical front end. It allows the user to select \texttt{tcpdump} files used in extraction. Once the user chooses the \texttt{tcpdump} files, PEXT constructs a clustering hierarchy so that the user can dynamically change the cutoff distance and generate a new FSM. PEXT also allows the user to see which packets clustered together and further investigate the data (Figure 3.5).

These features support the exploration of different clustering cutoffs in order to find the cutoff point for a particular protocol. As we will demonstrate in the case study (Chapter 4), the user can first construct an FSM from simple test cases to determine the cutoff point for the protocol, and then use that cutoff point in other test cases with the same protocol. The cutoff point is the user-specified maximum distance at which clustering may occur.
3.7 Summary

This chapter presented the implementation of PEXT, our system for reverse engineering networked protocols. The process begins with the capture of protocol data from significantly varied application test cases. Once the data is captured, PEXT clusters all of the packets using the longest common subsequence as a distance measure. It then separates each test stream into flows based on the addressing information of the packets in the stream. PEXT then identifies states by first finding identical flows, and then by searching within the remaining flows for common sub-streams. Once individual states are extracted from stream data, PEXT forms the initial state machine. This initial state machine is then corrected by first guaranteeing unique transitions and then summarizing linear lists of states into a single, combined, state. This presents the user with the final product, a state machine capturing various uses of the network protocol.
4. Case Study

4.1 Introduction

To demonstrate the effectiveness of PEXT, described in Chapter 3, we chose to reverse engineer two known protocols. The following section describes these two case studies in detail. The first case study presents the Jabber/XMPP protocol (Section 4.2), which is a new communication protocol designed to allow two users to send XML message to each other over a network connection. This is the open source alternative solution to other proprietary protocols such as AIM’s Oscar and Microsoft’s Instant Messenger. Because Jabber is developed using the open source model, it is well defined and has a number of different implementations. The second case study will examine two different implementations of the FTP protocol (Section 4.3).

4.2 Jabber/XMPP (Adium)

4.2.1 Protocol Description

The Extensible Messaging and Presence Protocol (XMPP) was designed to exchange structured information between two network endpoints in close to real time. While the definition remained general, it was developed with instant messaging applications in mind. A key feature of XMPP is the ability to detect the presence of various entities. An entity is described by a domain name, but may also contain a user and resource names in its description. For example, a single user for a given domain may have multiple devices, such as a notebook or cellphone, that can be used for text messaging.

The presence feature of XMPP is created via a feature called the roster. Each user contains a roster of resources he wishes to monitor. One can add and subtract entities from this roster. Moreover, when one logs into an XMPP server, his roster is updated with whomever is available on the network. This system is similar to that of a buddy list in other chatting services such as AIM.

After reading RFC3920 [56], which describes the core of XMPP, and RCP3921 [57] which deals with the use of XMPP for messaging and presence, we were able to create a simplified FSM for an application
Figure 4.1: Documented Jabber FSM

The design FSM extracted from the documentation. The blank node was created so that the diagram would be easier to read. It is meant to convey the idea that once the user has logged into the system the four events described can be performed in any order until the Quit event is invoked.
implementing this protocol (Figure 4.1). According to RFC3921, a client is not required to get his roster back at the time he logs into a Jabber server. It is, however, recommended and we expect most clients will do so. Once a client is logged in, it is up to him to notify the server of his presence. Once the server is notified, it broadcasts this client’s presence to all other entities subscribed to receive such a message. At this point the client can send and receive messages to or from other clients, respectively. In addition, the client can receive presence broadcasts about other clients he is subscribed to. Finally, the client has the option of quitting, by first sending a presence message specifying that he is no longer on the system, and then disconnecting from the server. This “final” message is not necessary, but encouraged by the protocol writers.

There are other options that the client has that are not captured by this FSM. The client can manipulate his roster in a number of ways. He is able to request subscriptions to users, and to grant his own subscription rights. However, because we are interested in the simple example of messaging, those options are not included so as to not complicate the FSM.

4.2.2 Jabber Test Cases

In this case study we wanted to capture a select subset of features by executing them using a Jabber client. The features we choose to concentrate on include:

1. Log in/out
2. Receive Presence Information
3. Send Message
4. Receive Message

These features are important to someone attempting to recreate their own Jabber client. The features provide the basic text messaging capabilities, and the ability to display who is available to receive messages.

The Jabber application is highly interactive. Events occur not only as a response to one using a Jabber client, but also as responses to the other entities on the network that wish to communicate with the user. For example, a message being received by the client is related not to any action instantiated by the user of the client, but by another resource the user is communicating with. Due to this added complexity, the XMPP
protocol proved to be an effective demonstration of why the graph formation methods described in Section 3.5 are necessary.

In this case study we wanted to demonstrate how an FSM changes as new features are reverse engineered. We will begin with two scenarios, logging into a Jabber network, when none of the resources on the roster are available, and logging into the network when a resource in the roster is available.

The produced FSM demonstrates two paths of execution, one with simply logging into and out of a session, and another with logging into a session and receiving presence information. Using this starting point we will add additional tests. First, we will add a test were the user logs into the session and sends a few messages to a resource in the roster. We will also present a separate case where the user receives messages from a resource in the roster. Then we will add several more test cases where the transmission and reception of messages is mixed. This final FSM will capture all of the features described in Figure 4.1.

4.2.3 Results

This section describes the result of several experiments performed using the Pidgin implementation of the Jabber protocol [27]. It will present and discuss the results of the logging in experiment, discuss the results of adding exclusive sending or receiving of messages to the log in tests, and finally, present the extracted FSM containing all of the features described previously.

Throughout this section diagrams encompassing different collections of test cases will be presented. All states have been annotated with a state ID chosen by PEXT and a description of the first packet handled by that state. This is usually a good indication of what the state does. We decided to summarize the first packet instead of directly reading the first few characters of it, for a more effective presentation.

Login Results

We began this case study with a simple test of logging into and out of a Jabber session. In this case we performed two independent tests. The first scenario has the user log into a session, see that no one in the roster was available, and then log out of the session. The second scenario involved first logging in, then seeing that a resource in the roster was available, and finally logging out of the session. The final, extracted FSM for these tests can be seen in Figure 4.2.
The Finite State Machine extracted from a combination of two test cases: Logging into and out of a Jabber session and logging into a Jabber session, receiving *presence* information, and logging out of the session. Each state is labeled with an ID provided by PEXT and a short description summarizing the first packet found in the state. One can observe a single branch where these two test cases deviate. State 3 represents a single packet send to the client from the Jabber server notifying it of the *presence* of a resource.
The process of logging into a Jabber session is always the same and thus the only difference between the two test cases presented in Figure 4.2 is state 3. State 3 contains all of the packets required when notifying the user of the presence of a resource in the roster. Because only a single resource was available at any one time, this presence notification is a one time event and once it occurs the process moves onto the logout request state.

Send and Receive Results

The next set of experiments performed involved two additional test cases. The first had the user logging into the session and then sending three unique messages to a resource in the roster. The next test case involved the user logging into the session and then receiving three unique messages from a resource in the roster. We treated each as a separate experiment and a combination of the previous two test cases from Section 4.2.3 as well as one of these new cases. We expect that the final FSMs for both of these experiments will look nearly identical to each other.

To demonstrate the necessity of the “pull-out” method described in Chapter 3, Figure 4.3 presents the FSM without using it and Figure 4.4 presents the FSM after the technique is applied. The differences are immediately apparent. Figure 4.3 has two XML start states and no presence notify state. This is due to the fact that PEXT identifies the longest common substring as a state. In this case two test cases contain packets involved in presence notification. Because these packets are identical and happen immediately after the XML start state, PEXT grouped them together with the XML start state as the longest common substring. This left a separate substring representing the XML start state without presence notification, which was grouped into its own state. Thus, PEXT produced two separate XML start states, with the only difference being that one of them has the presence notify packets tacked on to the end.

To solve this problem the “pull-out” method is used. Because the start state leads to both of the states, 1 and 2, it pulls out the common pieces of those states and forms a new state. Because 1 is a duplication of 2 with a presence notification packet tacked onto the end, all of 2 is pulled out and removed. PEXT then updates the link from the next XML start state to the Logout request state. Also the previous state 1 XML start state only has presence packets remaining and thus can be renamed to the presence notify state. Using such a method PEXT is able to get the FSM found in Figure 4.4.
This finite state machine demonstrates the limitations of PEXT if the “pull-out” method described in Chapter 3 is not used. It represents a collection of three test cases: User logging into and out of the Session, user logging into the session, receiving a presence notification, and logging out, and user logging into the session, receiving a presence notification, sending three messages, and logging out.
Figure 4.4: Extracted Send Messages FSM
An extracted FSM that represents a collection of three test cases: user logging into and out of the Session, user logging into the session, receiving a presence notification, and logging out, and user logging into the session, receiving a presence notification, sending three messages, and logging out.
Figure 4.5: Extracted Receive Messages FSM
An extracted FSM that represents a collection of three test cases: user logging into and out of the Session, user logging into the session, receiving a presence notification, and logging out, and user logging into the session, receiving a presence notification, receiving three messages, and logging out.
In addition to send messages, we wanted to make sure that receiving messages produced a similar FSM. Figure 4.5 contains the final FSM using test cases described in Section 4.2.3 and a test case where the user logs into a Jabber session, receives three unique messages from a resource in the roster, and logs out of the Jabber session. The final FSM is mostly identical to that of the send test cases found in Figure 4.4. The only difference being that state 2 Message Send is replaced with the Message Received state. This result was expected as the same packets are sent every time a message is received, and because that was the only difference between the collections of test cases, that should be the only difference in the final FSM.

Full FSM Results

The final experiment performed using the Jabber distribution involved reverse engineering the FSM for the entire set of features. Several different test cases were used in this endeavor and will be described in this section. This experiment used all of the test cases described previously as well as new cases that involved the user and a resource communicating back and forth at different intervals. In all of the new cases the user would first log into the session, receive presence information about a particular resource, perform a test that involved sending and receiving messages in different order, and finally log out of the session.

The experiment used the following tasks: the user sent and then received a message, the user received then sent a message, the user sent two messages and then received one, the user received two messages and then sent one, the user sent a single message and then received two, the user received a single message and then sent two, the user alternated between sending and receiving messages three times starting with a message send, and finally the user alternated between receiving and sending messages three times starting with a message received. Using these messages we hoped to generate a minimal graph were the user is able to send or receive messages after each transition. Using the test cases we hoped to cover several different transitions.

Similar to the previous experiment this collection also allowed us to demonstrate the necessities of the graph transformation techniques introduced in Chapter 3. Figure 4.6 presents the FSM extracted from the test cases without performing any of the specified transformation techniques. Because there were several test cases where after logging into the session and getting presence information the user send or received a message, those are grouped together into their own states. In this experiment there were only two test cases where the user neither send or received information. The only difference between them being that in one
The initial FSM extracted by PEXT, before performing any of the simplification rules. It is used here to demonstrate the effectiveness of the simplification techniques. The FSM contains several states that are a combination of multiple other states found throughout the graph. The simplification techniques applied to it afterwards are able to remove these duplicate states and thus make the final graph easier to comprehend.
Figure 4.7: Extracted Final Jabber FSM
The final extracted FSM based on the provided test cases. It is a simpler version of the FSM found in Figure 4.6. In this case one can observe that once the user has logged into the session and discovered a resource with which he can communicate, they are able to send and receive messages freely.
of the test cases presence information was also received. Thus, they both had the packets belonging the the XML start state in common. Therefore, that was extracted as a separate state.

In addition to extracting three different start states, the initial FSM also contains two incorrect states. States 5 and 6 capture two different orderings of the send message and receive message events. Because multiple test cases contained these orderings they were captured into their own states. This introduces quite a bit of confusion to the FSM. There is effectively an edge for each of the test cases, and the states do not feel natural.

To fix these problems PEXT applies transformations described in Chapter 3 to this graph and produces the graph found in Figure 4.7. PEXT applies the “pull-out” transformation and the groups states that represent identical packets, but have different names, together. This process is done repeatedly until no more transformations are possible. The final graph is far simpler and intuitive to understand. We see that there is a login process, a notification process, and finally the user is able to send or receive messages any number of times. At the end, the user enters the Logout Request state and ends the connection.

Figure 4.8 contains a reformatted version of the diagram in Figure 4.7 created to highlight the similarities and differences between the extracted FSM and the designed one found in Figure 4.1. There are only a few differences between the extracted and the designed FSMs. These can be attributed to the type of test cases that were chosen.

For example, in the designed FSM, getting the roster is an optional event, however in the extracted one the client always requests the roster once the user has been logged in. Thus, the branch from the login state to the “send presence” state does not appear in the extracted FSM, and PEXT groups all of the three states together into a single login state.

The second difference between the extracted and designed FSMs is that, according to the design, the user is able to send or receive messages without receiving any presence information. However this was never the case in our tests and thus those edges are missing. Overall, the extracted FSM closely resembles that of the designed one.
A reorganization of the graph found in Figure 4.7, designed to make it easier to compare the extracted FSM to the documented one. Solid edges are those elements that are found in both the extracted and documented FSMs, while dashed edges are those found in the documented FSM only. Because the states Login, Get Roster, and Send Presence all happen in the same order every time in all of our tests, those states were grouped together by PEXT into the XML Start state. Also, because in the test cases the user only communicated with entities that send a presence notification to him and because no new entities came online in the middle of the test, send presence always goes to either the Quit or the Receive Presence states. And the Receive Presence state can only be reached from the Send Presence state.
4.3 FTP Case Study

In this section we demonstrate the effectiveness of our approach by employing PEXT on applications using the FTP protocol. We begin by first describing the FTP protocol in Section 4.3.1. We then present an example that involves extracting the log-in phase of the FTP protocol in Section 4.3.2. Section 4.3.3 sets up our case study of extracting the FTP protocol as implemented by two different FTP applications. Section 4.3.4, describes the results from the case study.

4.3.1 FTP Protocol Description

The File Transfer Protocol (FTP) was standardized in 1985 [52] and further extended in the late nineties to address emerging security [34] and internationalization [13] concerns. It is used to transfer data between two hosts on a network. FTP assumes a reliable channel and, thus, uses TCP as its underlying protocol.

FTP begins with user authentication. The user must first log into an FTP server in order to acquire access to an array of commands. Once the user has logged in, he can browse available files in a directory structure. He can both send and receive files, assuming that he has the appropriate permissions. At the end of the session, the user logs out of the FTP server by sending a \texttt{QUIT} command.

In our tests, we studied FTP in passive mode, meaning that any transfer, either of a file or a directory listing, opens a new network flow. By a new network flow we mean a new TCP connection between the server and the FTP client. The user first sends a passive mode request to the server. The server responds with the IP and port information of the data flow. The user proceeds to connect to the opened data port and only then sends his transfer request over the control connection.

This process is done for every file or directory listing the client wishes to receive. Most modern implementations of FTP clients have a graphical user interface and perform a directory listing immediately after logging into the FTP server (Figure 4.9).

As most available FTP clients are graphical, we decided to use GUI-based FTP clients in our tests. We chose Fetch [20] and Cyberduck [41] because they are both widely used.
Figure 4.9: A sequence of events that occurs when an FTP client logs into and out of an FTP server. INIT and CLS are the physical initialization and breakdown of TCP flows. Login is the authentication process that occurs between the client and the server. PASV is the client’s request for a new data connection to be opened. LIST is the client’s request for a directory listing. Transfer is the data transfer conversation between the client and the server in which the server transmits the actual data, while the client receives it and responds with TCP acknowledgment packets. QUIT is the client’s request to end the FTP session.
Figure 4.10: This is the authentication state machine found in the RFC 959 document for the FTP protocol. The \textit{USER} command defines a user name the client wishes to use to log into the server. The \textit{PASS} command sends the password to be used with the provided user name. And the \textit{ACCT} command transmits any additional information that may be needed to log into the server. The edges labeled with 1, 2, 3, 4, 5 correspond to FTP server response codes.

### 4.3.2 Reverse Engineering the FTP Login Protocol State Machine

For our first test case we decided to reverse engineer the login process of the FTP protocol. In this case we did not use an FTP client because we wanted to capture the error behavior of sending ill-formatted packets. RFC 959, for the FTP protocol, gives a detailed description of the login behavior, providing the FSM (Figure 4.10). During authentication the client is allowed three distinct commands, one right after the other. At the end of each command the server may respond with a success, in which case the other commands are unnecessary, a request for more information, in which case the client moves on to the next command, or a failure in which case the process is aborted. The state diagram also contains an error state, which captures any other message the server may send, however, this state should be unreachable. In Figure 4.10 each transition represents a packet that is sent or received by the user.

Once the \textit{USER} command has been processed, the server may request a password for that user, in which case the process continues with the \textit{PASS} command. Generally the process ends here with a successful login response, however in some cases, the server may request additional account information that the client would have to respond to with the \textit{ACCT} command. We chose to model the login process by using
only the USER and PASS commands, since that is the common case.

We used telnet to connect to an FTP server to have full control over the packet formation. We then created four test cases to collect the data needed for the protocol extraction process:

- **Success Case**: Login with no errors and exit normally.
- **Invalid USER Packet**: Send an invalid USER request and exit normally. In this case we added unnecessary characters to the request information.
- **Invalid PASS Packet**: Send a correct USER packet and an invalid PASS packet and exit normally.
- **Wrong Password**: Send correctly formed packets, but send the wrong password and exit normally.

Figure 4.11 demonstrates the state machine created by PEXT. Each state in the FSM is a stream of exchanged packets between the FTP server and the telnet session. By simply labeling each edge with the type of packet found first in the state following it, we can see significant similarities between the extracted and designed state machine. For example, we can see that state 6 represents the actual conversation that handles the sending of the USER information. For that state, the server in our test cases may respond in
two ways, first with a 5 (failure) message, or a 3 (requesting more information) message. This is similar to the designed FSM (Figure 4.10), where the *USER* command leads the client to wait for a server response. Because our test cases do not generate server messages 1, 2, and 4 as a result of a *USER* command, we do not see states containing those messages in the extracted FSM. This information is valuable to a test engineer because he is able to see what features are not being tested and modify the test suite appropriately.

### 4.3.3 The FTP Applications Study

This case study captures some of the basic features of FTP. FTP uses a large number of flows to transmit data. It splits packets into a control flow and any number of data flows when working in passive mode. We wanted to capture three functionalities of an FTP client:

- capture logging in and out of the server
- capture directory browsing
- capture file retrieval

We did not capture any error conditions and, thus, the extracted protocols are state machines where everything behaves as expected.

In constructing our test cases, we used two different FTP servers with two different addresses, users, passwords, and file structures. In this way we were able to set the clustering limit described in Section 3.3.2. To do so we matched packets that contained identical control information, but different user information, such as the *USER* command packets. We were able to determine the distance at which packets clustered because we had control over user information. Using these packets as a guide we were able to set the maximum allowed distance for clustering of two entities to 0.64. This provided us with good results, meaning that the extracted FSM closely matched the documented one (Section 4.3.4).

We designed three tests to exercise the functionality of the FTP clients. First, the user logged in and out of each server. In the second test the user logged in, browsed a number of directories, and then logged out. In the final test the user logged in, browsed directories, downloaded a few small files, and logged out. These tests provided us with six `tcpdump` files of collected data which were used by PEXT to extract the FSM. These files contained roughly 500 FTP packets measuring 72KB in total. It took PEXT about 30 minutes to perform the necessary calculations to produce the FSM. Once the calculations are done, the user is free to explore various cut off points interactively.
Figure 4.12: This FSM was extracted directly from the documentation of the FTP protocol. It represents an FTP client that is able to: log in, get the current working directory, change the current working directory, get a directory listing, request a file, and quit. Note that once the user is logged into the server, he is able to perform any of the functions in any order as long as he does not quit.
4.3.4 Results of the FTP Applications Case Study

We began this case study by first extracting a subset of the FTP protocol from the documentation. Once the user is logged in, FTP is able to send a variety of messages. We decided to limit the functionality to:

- log in
- get working directory
- change working directory
- get a directory listing
- request a file
- quit

Using these restrictions we extracted the documented FSM (Figure 4.12). The protocol is bookended by the LOGIN and QUIT states, with all actions in the middle interconnected. For example, once the user has changed the working directory, he can perform any other function, thus there is an edge connecting all of the functionality.

We began our comparison by extracting the FTP protocol as implemented by the Fetch [20] client (Figure 4.13). After performing manual inspection we determined that the extracted protocol has six missing edges, one extraneous edge, and one extraneous state. When Fetch first logs into an FTP server it immediately requests the current working directory and a listing of that directory. Thus the FSM has a single edge leaving the log in state (912:220) leading to the PWD command state. Fetch only performs a request for the working directory when that directory changes to verify the change. Therefore, the only edges leading into state 24:PWD are from the login state and the change working directory state.

Another noticeable difference between the extracted and the documented diagrams can be seen in how the Print Working Directory and Change Working Directory states are handled. In the extracted FSM, both are split into two states, client request and a server response. However we do not count these as added states because they still follow the semantics of the protocol. The RFC abstracts the request state, state 24:PWD, and the response state, state 15:257, into a single Print Working Directory state. We can think of the request state as the entry point into the documented state and the response state as the exit point from the documented state.

One problem with the output of PEXT is the addition of state 30:150. This represents a message from the FTP server stating that the request for information is now being processed over the data flow.
Figure 4.13: This figure represents the extracted FSM from data collected using the Fetch FTP client. Each state is labeled by an ID assigned by PEXT during extraction, as well as the first three characters of the first packet in the sequence that is represented by the state. Solid edges represent transitions extracted by PEXT that match those found in RFC 959. Dotted edges represent transitions found in the documentation that were not found by PEXT. Dashed edges are extraneous edges extracted by PEXT. In this case there was a single extraneous edge leading from state 28:rdw back to the same state. Black nodes are extraneous nodes found by PEXT. Light gray nodes represent the data flow, while white nodes represent the control flow states.
conversation is the same regardless of what information is being sent, be it a directory listing or a file. This is a problem because if we travel to state 30 from state 11, the actual protocol requires that we go to state 28 and not state 909. We believe that in the future this problem may be solved by keeping a history of previous states traversed and only drawing an edge between states if that history matches. This introduces a number of complexities to the problem, such as detecting loops in the history, and is therefore saved for future work.

The final dissimilarity between the extracted and documented diagrams is the loop back in state $28: \text{rdw}$. This edge is drawn because the directory listings took a number of highly similar packets to transmit and those packets where clustered with the same ID. Thus, in the initial diagrams, edges leaving the directory listing packets either led back to themselves or the end of the flow state. This is similar to the problem with the extracted representation of the Print Working Directory and Change Working Directory states. It keeps the semantics of the FTP protocol and does not effect overall comprehension.

In addition to Fetch, we decided to extract the FTP protocol of another client, Cyberduck [41]. Figure 4.14 presents the extracted FTP protocol as implemented by Cyberduck. This protocol is missing nine edges and one state. It also has one extra state and one extra edge. We can immediately see similarities between the Fetch and Cyberduck extracted protocols. Both have the same problem of introducing a new state when trying to transfer a directory listing or a file. Both have a loop back to the state that is responsible for sending directory listings.

However one stark contrast is that Cyberduck does not request the current working directory from the server. In fact Cyberduck assumes that a successful return from the CWD command means that it is in the directory it wanted to change to. Whether or not this design decision is good, PEXT provides insights that can assist developers who are trying to understand the software.
Figure 4.14: The graph illustrates the extracted FSM from data collected using the Cyberduck FTP client. Each state is labeled by an ID during extraction, as well as the first three characters of the first packet in the sequence that is represented by the state. Solid edges represent transitions extracted by PEXT that match those found in RFC 959. Dotted edges represent transitions found in the documentation that were not found by PEXT. Dashed edges are extraneous edges extracted by PEXT. In this case there was a single extraneous edge leading from state 22:rdw back to itself. Black nodes are extraneous nodes found by PEXT. Light gray nodes represent the data flow, while white nodes represent the control flow.
5. Conclusion

This document describes a tool for automatically extracting protocols as Finite State Machines from captured network data. The tool relies on test selection, clustering and state selection, and FSM graph generation. This chapter summarizes that process (Section 5.1) and describes how this work can be used in its current form (Section 5.2). It concludes with future extensions we plan to make to this work (Section 5.3).

5.1 Summary

PEXT is a tool for extracting networked protocols as Finite State Machines from collected network traffic. Users of PEXT provide it with several test cases of the protocol and PEXT returns a state diagram that captures the functionality of those test cases. For example, in the case of the File Transfer Protocol (FTP), a test case can be “login into and out of an FTP server”, or “browse the directory structure of an FTP server”. Each test case can be represented as a tcpdump [39] file containing the collected traffic.

Once presented with test cases, PEXT clusters the packets using agglomerative hierarchical clustering [14]. After clustering, each test case is represented as a list of cluster identifiers and mined for initial states. A sequence of packets is labeled as a state when there is more than one identical sequences found among all test cases. PEXT also guarantees that all packets grouped into the same state have the same addressing information, meaning the same IP and port number information.

With each packet belonging to a state, PEXT forms all of the states into an FSM. This process is performed by first creating individual graphs for each test case, and then merging the graphs such that duplicate states are represented only once and all of the edges from the original graphs are present.

A problem with the initial FSM is that it is ambiguous. A transition from one state to the next means that the next packet sent or received belongs to the same cluster as the first packet in the following state. Thus, if a state has multiple edges leaving it, all of the states it can transition to must have initial packets from different clusters.

To solve this problem, and further simplify the FSM, PEXT executes the following procedures on the
graph. First, the “pull-out” method extracts the ambiguous packets from states and transforms them into a new state. Second, the “pull-in” method converts lists of states into new composite states.

There are two problems with this method. The first has to do with clustering and the termination criterion. It is not clear at what distance one needs to stop clustering packets. PEXT was designed to allow the user to interactively adjust this clustering criterion and observe its effects on the FSM and packet clusters. One approach to resolve this problem is to duplicate all of the test cases with different input. In the FTP case, one can log into different servers using different login information. Then the user can set the clustering distance to the minimum number at which these duplicate packets form a cluster. We found this to be a good approach that produced graphs closely correlating to those found in the documentation of network protocols (i.e., RFCs).

The second problem with this approach is with labeling the states. Without human intervention it is difficult to know the meaning contained in a collection of packets. In the FTP case we were able to resolve this problem by labeling each state with the first three characters of the first packet in that state. While this approach may work for ASCII based protocols, it will not work for binary protocols or more verbose XML based protocols. In the Jabber case study, which uses an XML based protocol, we labeled each state by stuudding the packets within it and summarizing their contents.

5.2 Current Uses

Protocols are a foundation of networked applications and hence it is important for software engineers to be able to understand and model application layer protocols. A protocol is an agreement, or a standard language, between two software entities that need to communicate. The application layer of the network stack, which is the layer that pertains to our work, is populated by the majority of network protocols. Each networked application must choose a pre-defined protocol, or use a newly created special-purpose protocol. Most developers choose to create their own application layer protocols.

While there are numerous advantages to using pre-existing protocols in applications, one difficulty is having to understand that protocol. We believe that PEXT can assist developers by observing the target protocol in action and presenting them with an FSM of that protocol.

Another problem with implementing an existing protocol is the scope of such a protocol. For example,
one may want to take an existing protocol and remove some functionality that is not needed in the current application. The developer needs to know which states can be removed safely and which are needed for the protocol to function correctly. Our approach is able to extract features of interest to the developer by observing the execution of those features. This will enhance developers’ understanding of those features and assist them in narrowing the scope of the protocol they want to implement.

Once a protocol has been created, developers need a method to debug it. There are several tools to assist them, however these tools rely on a high level understanding of the designed protocol state machine space [7, 8, 18, 24, 43, 44, 63]. These tools use the protocol’s designed finite state machine to generate test sequences, which are then executed on the protocol’s implementation.

We propose a system that employs reflexion techniques found in software engineering research [6, 12, 22, 47]. Using our technique, the developer would reverse engineer the underlying protocol’s state machine and compare it to a documented one. After creating this matching, developers can use reflexion software to calculate the similarities between the extracted and designed state machines automatically. This would not only point out any inconsistencies in the implementation, but also highlight any missing cases in the testing process, since the generated FSM would have missing states if they were not tested.

One could also use PEXT to perform back to back testing, by reverse engineering the implementation of a protocol and comparing it to another implementation of the same protocol. For example one could run two similar applications through the same test cases, reverse engineer each use of a protocol, and then compare the derived state machines. If both applications use the protocol in the same way, then the extracted protocols should be similar, however if they are implemented differently, that will be obvious to developers upon examination.

PEXT was developed to address the need for better tools to assist with the comprehension and development of networked applications. PEXT can reverse engineer networked application protocols from a collection of captured network traffic. We have shown its effectiveness by extracting the FTP and Jabber protocols.
5.3 Future Work

Using PEXT to cluster similar packets, we are able to extract the variant data from them. For example, clustering FTP USER packets enabled us to see where the actual user name information appears in them. Having this information may enable us to generate an API that developers can call to communicate with the provided protocol.

One of the limitations of PEXT is choosing appropriate names for extracted states in the FSM. There are a number of possible solutions we intend to try, including mining the source code or documentation for relevant semantic information. We may also be able to allow users to annotate their test cases or provide semantic information about the input data.

In addition to gathering semantic information to name FSM states correctly, we want to investigate how keeping track of the history, meaning the paths that the captured packets take through the FSM, can affect protocol extraction. We also want to add automated reflection functionality, currently done through manual inspection. With only a few user specified correlations, PEXT may be able to determine how closely the extracted FSM correlates to the documented one.

PEXT has the potential to help developers merge two protocols or extract relevant features from a single protocol. For example, developers can exercise the features of interest and get an API that supports only those features. In addition, given two protocols, PEXT may be able to merge them into a single protocol. It is not yet clear how automatic this process can be, but it may be as simple as users drawing edges in the protocol’s FSM to define the connection points between the protocols they wish to merge.

Merging two protocols may also be accomplished by developing a set of test cases that use both protocols. For example, we may want to merge the Jabber and FTP protocols in the following way. We can capture the behavior of a user requesting a file via chat and have chat open a new FTP connection to the client and send the requested file across. Using the packets generated by that scenario, PEXT may be able to extract a combined FSM. While there seems to be little use for this type of information, we can use it in conjunction with code generation features we want to add to PEXT in the future. This would allow PEXT to generate a protocol API that can interact with both Jabber and FTP clients in the manner described by the test cases.

Currently PEXT can only perform protocol extraction, however we want to explore the creation of a
reverse and forward engineering tool. We expect PEXT to allow developers to semi-automatically generate source code that represents extracted protocols and then either add them into their projects or build projects around them. We envision PEXT taking the role of creating and managing the source code of these protocols, such that, if new protocol features need to be added, developers will just create new test cases to exercise those features and let PEXT take care of the rest.
Bibliography


