A Reverse Engineering Tool for Extracting Protocols of Networked Applications

Maxim Shevertalov, Spiros Mancoridis
Department of Computer Science
College of Engineering
Drexel University
3141 Chestnut Street, Philadelphia, PA 19104, USA
{max, spiros}@drexel.edu

Abstract

Networked applications play a significant role in today's interconnected world. It is important for software engineers to be able to understand and model the behavior of these applications during software maintenance. Some networked applications use legacy protocols in ways they were not intended to be used. Others use newly created protocols that are designed in an ad hoc way to simply meet requirements. Protocol usage needs to be understood so that applications can be effectively tested and maintained. In this paper we propose the first step in achieving this goal by presenting a dynamic analysis tool, called PEXT, that can reverse engineer a networked application's underlying protocol by analyzing a collection of packets captured from the application at runtime. We demonstrate the effectiveness of this tool by extracting a protocol from an FTP application, and comparing the extracted protocol to the documented FTP protocol defined in RFC 959.

1 Introduction

Protocols are a foundational aspect 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 entities that need to communicate. The Internet is designed as layers of protocols. 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 needs. For example, the OSI Reference Model [16] defines a high-level layered structure that is imitated in today's packet switched networks. The application layer of the 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.

Networked protocols can be classified as stateless or stateful. In a stateless protocol, such as HTTP, the server answers client requests without retaining any client information. On the other hand, in a stateful protocol, such as FTP, the server "remembers" that the user logged into the server and controls what files that user can access. There are distinct states in FTP conversations, including log-in, log-out, and file transfer requests.

One typically uses an application layer protocol to create an application that can communicate over the network. There are a number of libraries created for various programming languages to help with developing networked applications. As a result, many developers create such applications without understanding that they are creating a new application layer protocol.

However, if a developer is aware of protocols that perform similar tasks, he can implement one of them, thus saving the cost of designing a protocol. Developers may also save time during development and testing by using a ready made implementation of the protocol. By using pre-existing protocols, developers will be able to create applications that can interoperate with other applications that use the same protocol.

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

Another issue 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 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. Our approach is able to extract features of interest to the developer, by simply 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.

Once a protocol has been created, developers need a way 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 [3, 4, 8, 12, 19, 20, 23]. 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 [2, 5, 11, 21]. 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 automatically calculate the similarities between the extracted and designed state machines. 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 regression testing, by reverse engineering the implementation of a particular protocol and comparing it to another implementation. 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.

The rest of this paper presents PEXT, our tool for protocol extractions (Section 2), a case study where we compare extracted protocols with those derived from documentation (Section 3), and conclusions and plans for future work in this area (Section 4).

2 PEXT: Protocol Extraction

PEXT is a tool that automates our approach to reverse engineering a protocol from network traffic collected from the application undergoing scrutiny (Figure 1). The process begins by capturing data from multiple, sufficiently different, execution traces of the application that implements the protocol (Section 2.1). It then uses this captured information to extract an FSM (Section 2.2). 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.

2.1 Data Collection

In this work we use tools that are based on `libPCap` [17] to collect network traffic. `LibPCap` has a number of implementations in a variety of languages. There are also a number of standalone applications that can save traffic data

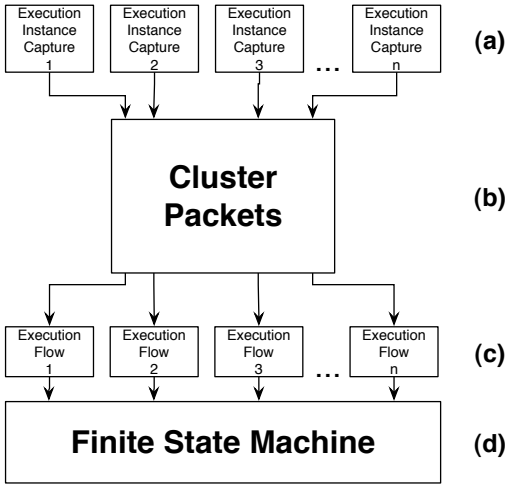

Figure 1: The general overview of the approach. **(a)** Capture packets from distinct execution traces representing specific features (Section 2.1). **(b)** Cluster all of the packets into distinct classes (Section 2.3). **(c)** Produce execution flow graphs for each distinct execution trace using *longest common string* matching to find states (Section 2.4). **(d)** Combine individual flow diagrams into a single, representative, finite state machine.

in `libPCap` format. We used `TCPDump` [17] as our main packet capturing application. `TCPDump` is able to filter captured traffic. We captured all of the packets sent and received by one node in the networked system. We choose to filter out any TCP acknowledgment packets because they represent TCP-specific conventions and are not a part of the application layer protocol we are trying to reverse engineer.

During the data capture, we were careful to design test cases that reflect the typical usage of the application being studied. If, for example, we were not careful and did 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.

2.2 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 2.3).
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 2.4).
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 2.4).
6. Label each remaining packets as a separate state (Section 2.4).
7. Create graphs for each of the test streams (Section 2.5).
8. Merge all of the graphs into a single FSM (Section 2.5).
9. Use the "pull-out" method to guarantee that each transition is deterministic (Section 2.5).
10. Use the "pull-in" method to combine lists of states to simplify and improve the readability of the FSM (Section 2.5).

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.

2.3 Clustering

We chose to use agglomerate hierarchical clustering [7] to group packets into separate classes. This clustering approach requires a function to calculate the distance measure (Section 2.3.1), and a termination criterion to calculate when clustering is complete (Section 2.3.2). While the rest of the paper describes our technique using only automated clustering, PEXT also allows users to cluster the packets manually.

2.3.1 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 into bytes and comparing them at that level of granularity provided us with similar results and improved performance. The final version of PEXT allows the user to choose this granularity.

Irrespective of the granularity, we used a dynamic programming approach to find the longest common subsequence (LCSS)[1] to calculate the distance between any two packets, a and b. The comparisons are normalized using the following formula:

$$D(a,b) = 1 - \frac{length(LCSS(a,b))}{max(length(a), length(b))} \quad (1)$$

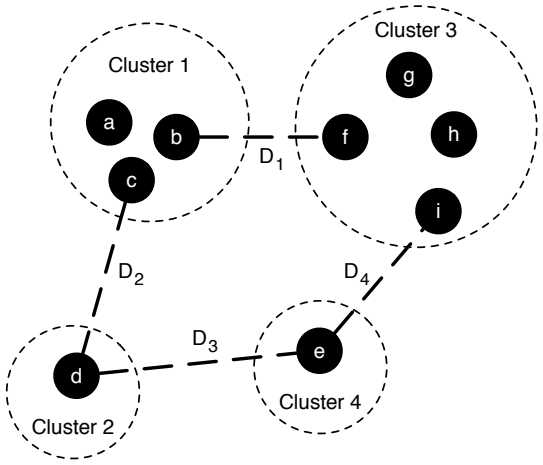

Figure 2: This example is designed to show how the distance between clusters is calculated in PEXT. The distance between cluster A and B is the smallest distance between any two elements $a \in A$ and $b \in B$. For example D_1, or the distance between clusters 1 and 3, is the distance between elements b and f.

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 2).

2.3.2 Termination Criterion

We stop clustering when the next two elements to cluster together have a distance greater than some specified parameter depending on the protocol. We can use this as a termination criterion because during the clustering process, at each step, the distance will stay the same or increase.

We chose to leave the maximum distance at which clustering occurs up to the user, because different protocols have different degrees of verbosity. For example, the FTP [22] protocol is composed of short command strings, and thus has a different maximum distance than the Jabber [10] 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.

2.4 State Selection

At this stage in the process we are presented with a number of test cases that are represented as strings of cluster IDs. We begin 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. We

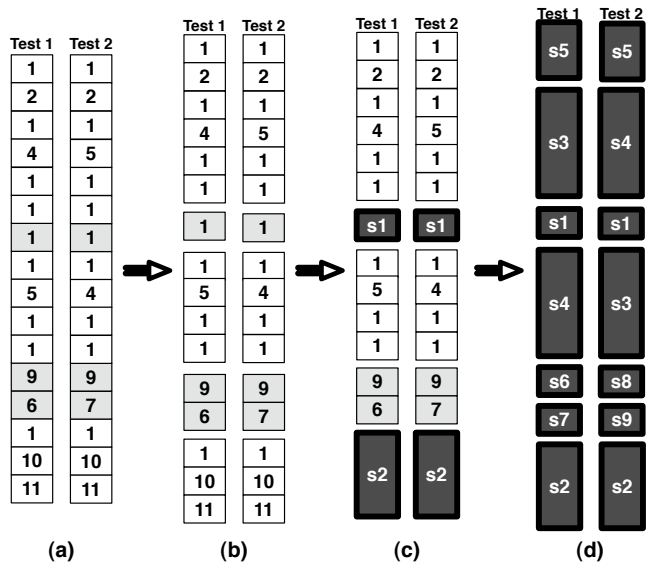

Figure 3: **(a)** The initial streams of test data as clustered packets. Each number represents the cluster ID of a particular packet. Light gray packets are part of the data flow, while white packets are part of the control flow. **(b)** Split the packet stream at each point where the stream transitions 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.

found that 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 we split a test stream into flows at each point where the packet addressing information changes. We maintain 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 our initial states. Since we made a restriction that each state may only contain packets of the same flow, this is a natural way of finding distinct states.

Once identical flows have been identified and labeled as states, we apply the longest common substring algorithm [1] to all leftover flows to derive all of the other states. We attempt to find common continuous subsequences across the entire set of provided test cases. We also restrict 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 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 individual sub-streams

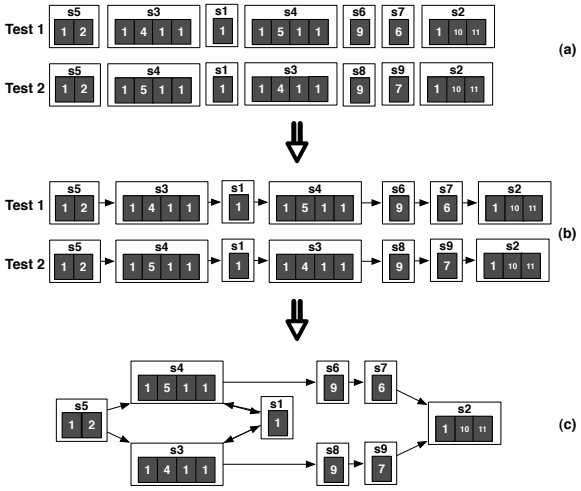

Figure 4: White boxes are individual states composed of packet streams. Dark gray boxes are the actual packets internal to a particular state represented by their cluster ID. **(a)** The initial streams with defined states such that each packet of the stream belongs to a single state. **(b)** The initial graph of states for each test stream. **(c)** The combination of individual test stream graphs into a single FSM.

based on where the protocol transitioned between flows. We then derive states `s1` and `s2` because each of the sub-streams contained within those states has a duplicate sub-stream. At this point each sub-stream can be thought of as a string where each of the symbols is a cluster ID. We search for the largest common non-overlapping substring (of length greater than two symbols) and combine them into states. In Figure 3 we find three common substrings: `(1,4,1,1)`, `(1,5,1,1)`, and `(1,2)`. These sequences become states `s3`, `s4` and `s5` respectively. Thus we group 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 we group each of the remaining packets into individual states, even if they have the same cluster ID. 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.

2.5 FSM Graph Formulation

Graph formulation is broken into two steps. First we combine all of the individual test streams graphs into a single graph. We then apply two methods, "pull-in" and "pull-out", in order to summarize the graph and make it easier to comprehend. At the end of the process we have a finite state machine representation of the application's protocol.

At this stage of the process we have identified the initial states such that each packet in each of the test streams be-

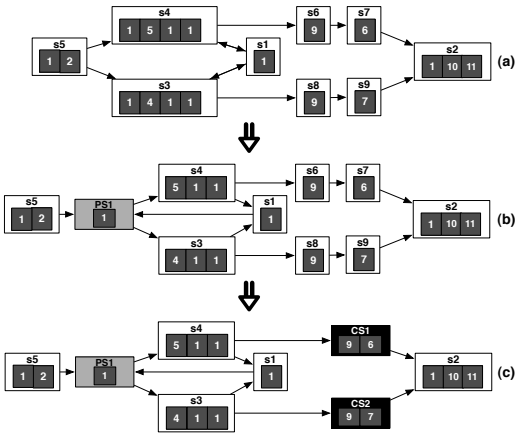

Figure 5: White boxes are original states extracted from the test streams. Dark gray boxes are the actual packets internal to a particular state represented by their cluster ID. Light gray boxes represent states created through the use of the "pull-out" method. Black boxes represent new states created by "pull-in" method. **(a)** The initial graph generated at the end of Figure 4. **(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**.

longs to a particular state. We begin by converting each test case from a stream of packets to a sequence of states. For example, in Figure 4, we go from the representation in Figure 4a, that we derived in Figure 3, to the representation in Figure 4b. Using these sequences we are now ready to extract the FSM.

To extract the FSM we begin by combining all of the test state sequences into a single graph. From these sequences we 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 we can iterate over each of the test sequences and fill in the hash map as we go from state to state. Once the hash map is complete, we can draw a graph simply by drawing edges from any key in the map to all of the associated values. We use Dot [13] to draw the actual graph.

Thus, we are able to derive the graph in Figure 4c from the sequences of Figure 4b. We can see that the branching points in Figure 4c 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, we employ two simplifying transformations. Each transition represents a packet, received or sent. Therefore, we need to guarantee that each transition leaving a state can be uniquely determined. In

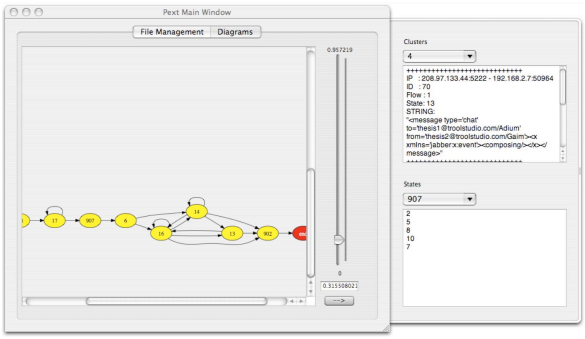

Figure 6: The `Diagrams` tab of the PEXT GUI is designed to allow the user to explore the effects of different cutoff points on the extracted FSM. The user is able to see the graph generated for each cutoff point as well as delve deeper into the clusters and determine precisely which packets cluster together. The `File Management` tab allows the user to add and remove `tcpdump` files.

cases where this uniqueness criterion is violated we apply a simple "pull-out" method to fix the problem. We form a new state with the violating string of packets and create a transition from the violating parent to it. The new state also acquires transitions from itself to the original, violating, children states. Finally we remove the violating stream of packets from the original children states.

This process is demonstrated by Figure 5b. We begin with a graph constructed at the end of Figure 4. We 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, we pull out the violating packet with cluster ID 1 and place it in its own state `PS1`. We then draw a transition from `s5` to `PS1` and transitions from `PS1` to the original children `s3` and `s4`. We have 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, we draw a transition from `s1` to `PS1`.

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 5c 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.

2.6 PEXT Tool Implementation

PEXT was created using Ruby [24] for the back end and Cocoa [14] for the graphical front end. It allows the user to select `tcpdump` files used in extraction. Once the user chooses the `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 6).

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 (Section 3), 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 FTP Case Study

In this section we demonstrate the effectiveness of our approach by employing PEXT on applications using the FTP protocol. In addition to FTP we have also tried our approach on applications implementing the Jabber protocol [10], a single flow, XML based protocol used in text messaging applications. We were able to achieve results comparable to those presented in this section.

We begin by first describing the FTP protocol in Section 3.1. We then present an example that involves extracting the log-in phase of the FTP protocol in Section 3.2. Section 3.3 sets up our case study of extracting the FTP protocol as implemented by two different FTP applications. Section 3.4, describes the results from the case study.

3.1 FTP Protocol Description

The File Transfer Protocol (FTP) was standardized in 1985 [22] and further extended in the late nineties to address emerging security [15] and internationalization [6] 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 *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.

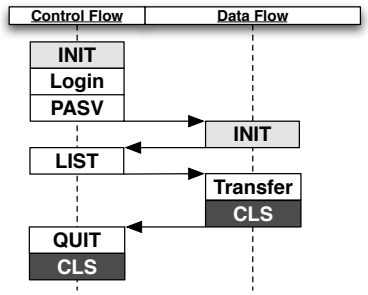

Figure 7: This presents a sequence of events that occurs when an FTP client logs into and out of an FTP server. *INIT* and *CLS* are the physical initialization and breakdown of TCP flows. *Login* is the authentication process that occurs between the client and the server. *PASV* is the client's request 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.

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. In addition, most modern implementations of FTP clients have a graphical user interface and perform a directory listing immediately after logging into the FTP server (Figure 7).

As most available FTP clients are graphical, we decided to use GUI-based FTP clients in our tests. We chose Fetch [9] and Cyberduck [18] because they are both widely used.

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-formated packets. RFC 959, for the FTP protocol, gives a detailed explanation of the login behavior, providing the FSM (Figure 8). 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.

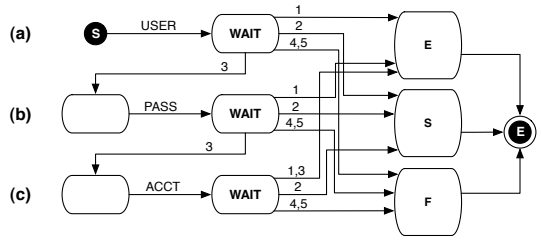

Figure 8: This is the authentication state machine found in the RFC 959 document for the FTP protocol. The *USER* command defines a user name the client wishes to use to log into the server. The *PASS* command sends the password to be used with the provided user name. And the *ACCT* command transmits 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.

In Figure 8 each transition represents a packet that is sent or received by the user.

Once the *USER* command has been processed, the server may request a password for that user, in which case the process continues with the *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 *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 9 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 8), where the *USER* command leads the client to wait for

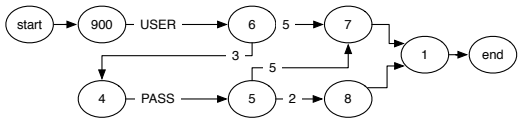

Figure 9: The extracted state machine created by running PEXT over the `tcpdumps` resulting from provided test cases. The labeling of each state is an arbitrary number chosen by PEXT to represent it. This figure contains a hand redrawn and relabeled representation of the original graph generated by PEXT. While the nodes and edges are kept identical to the original, each edge is labeled with the type of the first packet of the state immediately following it. For example state 5's first packet is the *PASS* message, thus the edge from state 4 to state 5 is labels *PASS*.

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.

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 conveniently 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 2.3.2. To do so we matched packets that contained identical control information, but different user information, such as the *USER* command packets. Because we had controller over user information we were able to determine the distance at which packets clustered. 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 FSM extracted from the documentation (Section 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

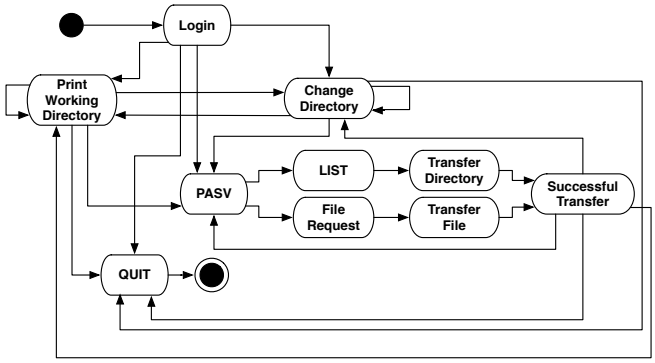

Figure 10: This FSM was extracted directly from the documentation of the FTP protocol. It represents an FTP client that is able to: log in, get the current working directory, change the current working directory, get a directory listing, request a file, and quit. Note that once the user is logged into the server, he is able to perform any of the functions in any order as long as he does not quit.

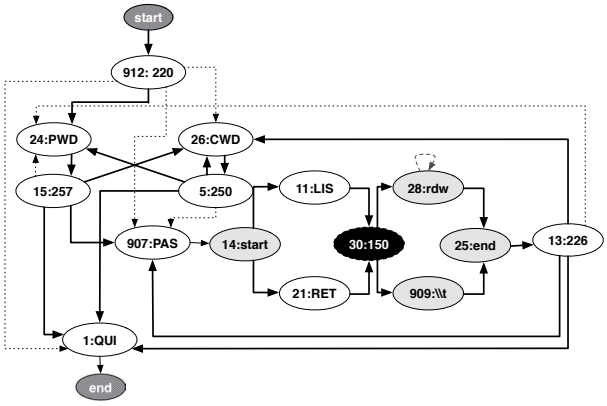

Figure 11: This figure represents the extracted FSM from data collected using the Fetch FTP client. Each state is labeled by an ID assigned by PEXT during extraction, as well as the first three characters of the first packet in the sequence that is represented by the state. Solid edges represent transitions 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.

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.

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 10). 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 [9] client (Figure 11). 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.

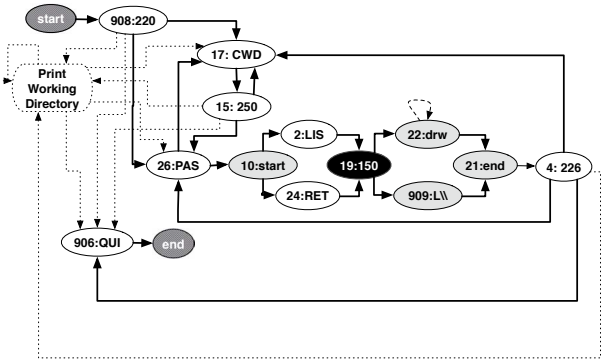

Figure 12: The graph illustrates the extracted FSM from data collected using the Cyberduck FTP client. Each state is labeled by an ID during extraction, as well as the first three characters of the first packet in the sequence that is represented by the state. Solid edges represent transitions extracted 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.

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. This 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: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 [18]. Figure 12 presents the extracted FTP protocol as implemented by Cyberduck. This protocol is missing nine edges and one state. In addition it 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. In addition 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.

4 Conclusions and Future Work

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 in extracting the FTP protocol and how that information can be useful. For example, using PEXT, we can see that Cyberduck does not check to ensure that the "change working directory" command was processed correctly. This type of information is valuable to engineers who are testing networked applications. In the future we want to try using PEXT to reverse engineer a greater number of network protocols. We have tried it on a few so far, but more work needs to be done.

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 create applications that can interoperate with other networked applications.

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 automated this process can be, but it may be as simple as users drawing a few edges in the protocol's FSM to define the connection points between the protocols they wish to merge.

One of the limitations of PEXT is choosing appropriate names (semantic information) for extracted states in the FSM. There are a number of possible solutions we hope to try in the future, including mining the source code or documentation for this information. We may also be able to gather semantic understanding by allowing users to annotate their test cases or provide semantic information about the input data.

In addition to gathering semantic information to correctly name FSM states, 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 and if we can remove extraneous states that have false branching. In addition we want to add automated reflexion functionality, currently done through manual inspection, to PEXT. With only a few user specified correlations, PEXT will be able to determine how closely each extracted FSM correlates to the documented FSM. Finally, we would like to add code generation functionality to PEXT so that we can generate the API of the implemented protocol.

References

[1] *Introduction to algorithms*. MIT Press, Cambridge, MA, USA, 2001.

[2] A. Bhattacharjee, A. Seby, G. Sen, and S. Dhodapkar. Clas: a reverse engineering tool. In *Software Testing, Reliability and Quality Assurance*. IEEE, December 1994.

[3] S. Boyd and H. Ural. On the complexity of generating optimal test sequences. *Software Engineering, IEEE Transactions on*, 17(9):976–978, September 1991.

[4] S. Chanson and J. Zhu. A unified approach to protocol test sequence generation. In *Computer and Communications Societies, Networking: Foundation for the Future*, volume 1, pages 106–114, 1993.

[5] D. Cooper, B. Khoo, B. R. von Konsky, and M. Robey. Java implementation verification using reverse engineering. In *ACM International Conference Proceeding Series*, volume 56, pages 203–211. ACM, 2004.

[6] B. Curtin. Internationalization of the file transfer protocol. `ftp://ftp.rfc-editor.org/in-notes/rfc2640.txt`.

[7] W. Day and H. Edelsbrunner. Efficient algorithms for agglomerative hierarchical clustering methods. *Journal of Classification*, 1(7):7–24, 1984.

[8] K. El-Fakih, N. Yevtushenko, and G. Bochmann. Fsm-based incremental conformance testing methods. *Transactions on Software Engineering*, 30(7):425–436, 2004.

[9] Fetch. Fetch – the mac os x ftp/sftp client. `http://www.fetchsoftworks.com/`.

[10] J. S. Foundation. Xmpp.org. `http://www.xmpp.org/`.

[11] G. Gannod and S. Murthy. Verification of recovered software architectures. In *11th IEEE International Workshop on Program Comprehension*, pages 258–265, May 2003.

[12] G. V. Ghedamsi, A. Bochmann. Test result analysis and diagnostics for finite state machines. In *Distributed Computing Systems*, pages 244–251, June 1992.

[13] Graphviz. Graphviz. `http://www.graphviz.org/`.

[14] C. Guides. Ruby programming language. `http://developer.apple.com/documentation/Cocoa/`.

[15] M. Horowitz and S. Lunt. Ftp security extensions. `ftp://ftp.rfc-editor.org/in-notes/rfc2228.txt`.

[16] IEEE. *The OSI reference model*. IEEE, December 1983.

[17] JWS. Tcpdump public repository. `http://www.tcpdump.org/`.

[18] D. V. Kocher. Cyberduck — ftp for mac os x. `http://cyberduck.ch/`.

[19] D. Lee and M. Yannakakis. Testing finite-state machines: state identification andverification. *Computers*, 43(3):306–320, March 1994.

[20] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines-a survey. *Proceedings of the IEEE*, 84(8):1090–1123, August 1996.

[21] G. C. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: bridging the gap between source and high-level models. *SIGSOFT Softw. Eng. Notes*, 20(4):18–28, 1995.

[22] J. Postel and J. Reynolds. File transfer protocol (ftp). `ftp://ftp.rfc-editor.org/in-notes/std/std9.txt`.

[23] D. Sidhu and T. Leung. Formal methods for protocol testing: a detailed study. *Transactions on Software Engineering*, 15(4):413–426, April 1989.

[24] R. V. I. Team. Ruby programming language. `http://www.ruby-lang.org/en/`.