Discoverer: Automatic Protocol Reverse Engineering from Network Traces

Weidong Cui
Microsoft Research
wdcui@microsoft.com

Jayanthkumar Kannan
UC Berkeley
kjk@cs.berkeley.edu

Helen J. Wang
Microsoft Research
helenw@microsoft.com

Abstract

Application-level protocol specifications are useful for many security applications, including intrusion prevention and detection that performs deep packet inspection and traffic normalization, and penetration testing that generates network inputs to an application to uncover potential vulnerabilities. However, current practice in deriving protocol specifications is mostly manual. In this paper, we present *Discoverer*, a tool for automatically reverse engineering the protocol message formats of an application from its network trace. A key property of Discoverer is that it operates in a protocol-independent fashion by inferring protocol idioms commonly seen in message formats of many application-level protocols. We evaluated the efficacy of Discoverer over one text protocol (HTTP) and two binary protocols (RPC and CIFS/SMB) by comparing our inferred formats with true formats obtained from Ethereal [5]. For all three protocols, more than 90% of our inferred formats correspond to exactly one true format; one true format is reflected in five inferred formats on average; our inferred formats cover over 95% of messages, which belong to 30-40% of true formats observed in the trace.

1 Introduction

Application-level protocol specifications are useful for many security applications. Penetration testing can leverage protocol specifications to generate network inputs to an application to uncover potential vulnerabilities. For network management, protocol specifications can also be used to identify protocols and tunnelings in monitored network traffic. Generic protocol analyzers (GAPA [1] and binpac [16]) are important mechanisms for intrusion detection or firewall systems to perform deep packet inspection. These analyzers take protocol specifications as input for their analyses.

To date, protocol specifications for the above applications are specified from documentation or reverse engineered manually. Such efforts are painstakingly time-consuming and error-prone. It took the open-source SAMBA project 12 years to manually reverse engineer the Microsoft SMB protocol [18]. In another example, the Yahoo messenger protocol has also been persistently reverse engineered, despite which, the open source clients [6] regularly require patching to support proprietary changes in the Yahoo protocol. Sometimes, the period between the availability of an official client and an open-source client has been a month, with some open-source projects simply abandoning the effort due to the frequent changes initiated by Yahoo.

To address this pain, we tackle the problem of automatic protocol reverse engineering. There can be two sources of given input for the reverse-engineering task: network traces and application code. In this paper, we present our tool, *Discoverer*, which performs automatic reverse engineering from network traces. We leave application-code-based reverse engineering as future work.

In Discoverer, we focus on reverse engineering the message format specification and leave the protocol state machine inference to our future work. To automatically reverse engineer message formats for a wide range of protocols, we face three main challenges: (1) We have very few hints from the network trace. The only evident information from the trace is the directionality of byte streams. (2) Protocols are significantly different from each other. (3) Protocol message formats are often context-sensitive where earlier fields dictate the parsing of the subsequent part of the message.

To make our tool general, we base our design on inferring protocol idioms commonly seen in message formats of many protocols. To cope with the few hints, we dissect

the formless byte streams into text and binary segments or tokens as a starting point for clustering messages with similar patterns, where each cluster approximates a message format. By comparing messages in a cluster and observing the characteristics of known cross-field dependencies (such as a length field followed by a string of the length), we infer additional properties for the tokens, which in turn can be leveraged to refine and divide the clusters of messages, where each subcluster approximates a more precise format. This process continues recursively until we can no longer divide up any message clusters based on the newly finished inference. After this recursive clustering phase, we look at all message clusters globally through a type-based sequence alignment algorithm, and merge similar clusters into one. This way, we can produce more concise message formats.

We have evaluated Discoverer over traces of a representative set of protocols consisting of one text protocol (HTTP) and two binary protocols (RPC and CIFS/SMB). We calibrated our design over some of these traces, and used the remaining for validation. The three main metrics for our tool are *correctness* ("does one inferred format correspond to exactly one true format?"), *conciseness* ("how many inferred formats is a single true format reflected in?"), and *coverage* ("how many messages are covered by the inferred formats?"). Across all protocols we tested, more than 90% inferred formats correspond to exactly one true format; one true format is reflected in five inferred formats on average; our inferred formats cover over 95% messages, which belong to 30-40% of true formats observed in the trace. Such significant difference between message and format coverage is due to the heavy-tail distribution of message format popularity commonly seen in practice.

Although our reverse-engineered message formats are imperfect, we anticipate them to be still practical for the aforementioned applications. For instance, penetration testing guided by our reverse-engineered formats is likely to be much more effective than that with random inputs. Protocol fingerprinting and tunneling detection probably do not require perfect protocol specifications. For applications like firewalls which would err with imperfect specifications, our tool could still serve as a help to ease the manual protocol specification process.

We organize the rest of the paper as follows. We discuss common protocol idioms and the scope of Discoverer in Section 2. We describe the design of Discoverer in detail in Section 3. We present our evaluation methodology and results in Section 4. We discuss related work in Section 5, and limitations and future work in Section 6. Finally, we summarize the paper in Section 7.

2 Problem Statement

Many application-level protocols share common protocol idioms which correspond to the essential components in a protocol specification. To make our reverse-engineering algorithm applicable to many protocols, we base our design on inferring the common protocol idioms. In this section, we first describe these idioms and then explain the scope of Discoverer.

2.1 Common Protocol Idioms

Most application-level protocols involve the concept of an *application session*, which consists of a series of *messages* (also known as Application-level Data Units or ADUs) between two hosts that accomplishes a specific task. The structure of an application session is determined by the application's *protocol state machine*, an essential component in a protocol specification that characterizes all possible legitimate sequences of messages. The structure of an application message is determined by the application's *message format specification*, another essential component in a protocol specification. A message format specifies a sequence of *fields* and their semantics. Common field semantics include *length* (reflecting the size of a subsequent field with a variable length), *offset* (determining the byte offset of another field from a certain point like the start of the message), *pointer* (a special offset that specifies the index of a field in an array of arbitrary items), *cookie* (session-specific opaque data that appears in messages from both sides of the application session; session IDs are an example of cookie fields), *endpoint-address* (encoding IP addresses or port numbers in some form), and *set* (a group of fields that can be put in an arbitrary order).

One particular type of fields is the *Format Distinguisher* (FD) field. The value of this field serves to differentiate the format of the subsequent part of the message, which reflects the context-sensitive nature in the grammar of many application-level protocols. A message may have a sequence of FD fields, particularly when multiple protocols are encapsulated. For instance, a CIFS/SMB message consists of a NetBIOS header encapsulating an SMB header, which in turn may encapsulate a RPC message. This implies that the applications need to scan a message from *left-to-right*, decoding a FD field before parsing the subsequent part of the message.

2.2 Scope of Discoverer

In this paper, we focus on deriving the message format specification and leave protocol state machine inference

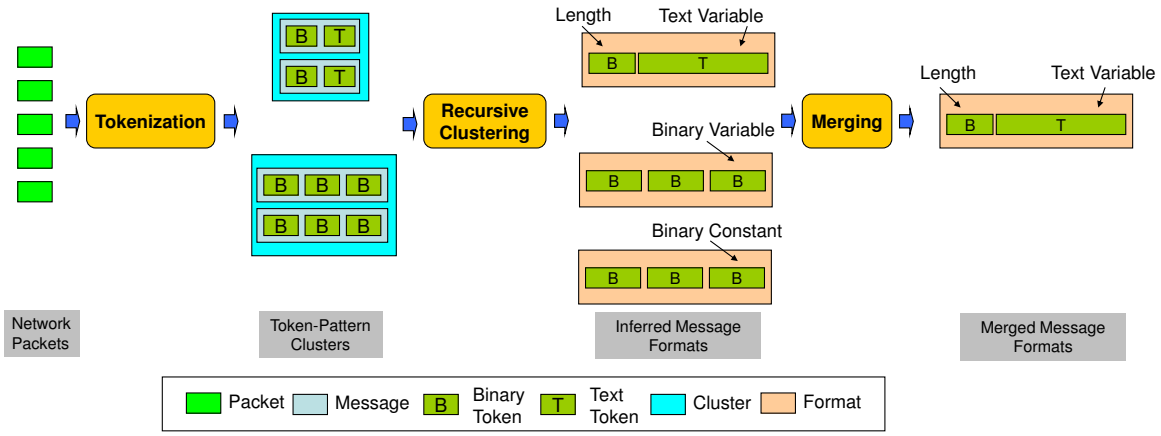

Figure 1: Overview of Discoverer's architecture. In the example, we assume there is a single true message format which has two fields: the first binary field of a single byte represents the length of the second text field. There are two token patterns because, when the text field is shorter than a threshold, it is treated as binary. In the merging phase, this kind of tokenization errors is corrected.

to our future work. We assume synchronous protocols to identify message boundaries. A message is a consecutive chunk of application-level data sent in one direction. It spans one or more packets in a TCP or UDP connection, where a UDP connection is a pair of unidirectional UDP flows that are matched on source/destination IP address/port number. We only aim to deal with applications that do not obfuscate their payloads. We do not aim to capture timing semantics (e.g., "message 1 usually follows message 2 within 10 seconds").

3 Design

In this section, we first present an overview of Discoverer, then describe the three main phases of Discoverer in detail, and finally give a concrete example of a message format inferred by Discoverer.

3.1 Overview

The basic idea of Discoverer is to cluster messages with the same format together and infer the message format by comparing messages in a single cluster. We achieve this in three main phases (illustrated in Figure 1).

- **Tokenization and Initial Clustering**: This phase operates on the raw packets, and helps in identifying field boundaries in a message and giving the first order structure to the unlabeled messages. We first reassemble the packets into messages, and then break up a message into a sequence of tokens which is an approximation to a sequence of fields. Tokens belong to one of two *token classes*: binary or text. We then classify messages into various clusters based on each message's token pattern, which is simply represented by the message direction and classes of its tokens.

- **Recursive Clustering**: Since messages with the same token pattern do not necessarily have the same format, this phase further divides clusters of messages so that messages in each cluster have the same format, and infers the message format by comparing messages in each single cluster. To do so, we mimic the left-to-right recursive parsing of applications processing messages by recursively repeating the following steps. We first infer the message format that captures the content of all messages in a cluster. Then we identify the first FD field (which decides the format of the subsequent part of the message) in a left-to-right scan and use the values of this FD field to divide the cluster into subclusters.

- **Merging**: This phase mitigates the over-classification problem, namely, messages of the same format may be scattered into multiple clusters. To do so, we merge similar message formats by using a *type*-based sequence alignment algorithm that compares the field structure of two inferred message formats.

A key design rationale for Discoverer is to be *conservative*: it may scatter messages of the same format into more than one cluster, but it should not collate messages of different formats into the same cluster. This rationale is to ensure the correctness of inferred formats because, if there are messages of more than one format in a cluster, the inferred format might be too general by trying to capture multiple message formats at once.

3.2 Tokenization and Initial Clustering

3.2.1 Tokenization

A token is a sequence of consecutive bytes likely to belong to the same application-level field. We require that the tokenization process works *without* any particular distinction between text and binary protocols, since our tool is intended to be fully automatic and we wish to spare the user from the manual effort required to distinguish between text and binary protocols. Further, it is hard to declare a protocol as purely text or purely binary, since text protocols can contain binary bytes (e.g., an image file transferred over HTTP) and most binary protocols contain a few text fields (e.g., the name of a file).

Our tokenization procedure generates two *classes* of tokens: text and binary. A text token is intended to span the several bytes of a single message field representing some text (such as "GET" in an HTTP request). Our procedure for finding text tokens is as follows: we first identify text bytes by comparing them with the ASCII values of printable characters, and then consider a sequence of text bytes sandwiched between two binary bytes as a text segment. To avoid mistaking binary bytes for text bytes, we require this sequence to have a minimum length. Then we use a set of delimiters (e.g., space and tab) to divide a text segment into tokens. We also look for Unicode encodings in messages. For binary fields, identifying field boundaries is very hard; so we instead simply declare a single binary byte to be a binary token in its own right. Note that this procedure can admit errors: consecutive binary bytes with ASCII values of printable characters are wrongly marked as a text token; a text string shorter than the minimum length is wrongly marked as binary tokens; a text field consisting of some white space characters is wrongly divided into multiple text tokens. We correct this kind of errors in the merging phase (see Section 3.4).

3.2.2 Initial Clustering by Token Patterns

Byte-wise sequence alignment based on the Needleman-Wunsch algorithm [15] has been used in previous studies [3, 11, 17] for aligning and comparing messages. We find that byte-wise sequence alignment, while ideally suited to align messages with similar *byte patterns*, is not suitable for aligning messages with the same *format*. For instance, fields with variable lengths may lead to mis-alignment of two messages of the same format. Further, parameter selection for sequence alignment is also hard as shown in [3].

To avoid aligning messages, we cluster messages based on their token patterns. The token pattern assigned to a message is a tuple, $(dir, class\_of\_token\_1, class\_of\_token\_2, \cdots)$, where dir is the direction of the message (client to server or vice versa), followed by the classes of all tokens in the message. We consider the message direction because messages in opposite directions tend to have different formats. An example of a token pattern is $(client\_to\_server, text, binary, text)$.

Note that this initial clustering is coarse-grained since messages with different formats may have the same token pattern. For instance, SMTP commands typically have two text tokens ("MAIL receiver", "RCPT sender", "HELO server-name"). In the recursive clustering phase, we improve the granularity of this clustering by recursively identifying FD tokens and dividing clusters.

3.3 Recursive Clustering

Our recursive clustering relies on identifying format distinguisher (FD) tokens. To find FD tokens, we need to invoke both format inference and format comparison. In this section, we first explain these procedures before describing how we recursively identify FD tokens and divide clusters.

3.3.1 Format Inference

The format inference phase takes as input a set of messages and infers a format that succinctly captures the content of the set of messages. Our inferred *message format* is defined to be a sequence of token specifications which include not only *token semantics* but also *token properties*. We introduce token properties because we cannot infer the semantic meaning for every token and certain token properties are useful for describing the message format. Token properties currently cover two perspectives: *binary* vs. *text* and *constant* vs. *variable*. The first property reflects the token class, and the second decides if a token takes the same value across all messages of the same format (i.e., constant token) or different values in different application sessions (i.e., variable

token). We also define the *type* of a token to be the sum of its semantic and property.

We now describe how token properties and semantics are derived.

Property Inference: Token class is already identified during the tokenization phase. Constant or variable tokens can also be easily identified. Since the set of messages come from a single token-pattern cluster, tokens in one message can be directly compared against their counterparts in another message by simply using the token offset. Thus, constant tokens are those that take the same value across the entire set of messages, and variable tokens are those that take more than one value.

Semantic Inference: We currently support three semantics: length, offset, and cookie (see Section 2.1 for definitions). We will discuss how it may be possible to support other semantics in Section 6. We identify cookie fields at the end of the merging phase since it requires correlating multiple messages in the same session. We employ the heuristics in RolePlayer [3] for doing this. Our heuristics for identifying length and offset fields are an extension of those in RolePlayer. The intuition for identifying length fields is that, for a specific pair of messages, the *difference* in the values of potential length fields (at most four consecutive binary tokens or a text token in the decimal or hex format) reflects the *difference* of the sizes of the messages or some subsequent tokens. We thus simply check for a match between the value difference and the size difference. If a match holds for all pairs of messages in the cluster, the potential length field is declared to be a length field. For offset fields, we compare the value difference with the difference of the offsets of some subsequent tokens.

3.3.2 Format Comparison

The goal of this procedure is to decide if two inferred message formats are the same. Given two formats, it scans these two formats token-by-token from left-to-right and matches the inferred type (i.e., semantic and property) of a token from one format against its counterpart from the other. If all tokens match, the two formats are considered to be the same.

Ideally, two tokens can be considered to match if their semantics match. However, since there are always tokens that we do not have semantics for, we need to compare their values (they have the same token class since the two formats have the same token pattern). We allow a constant token to match with a variable token if the latter takes the value of the former at least once. We also allow a variable token to match with another if there is an overlap in the two sets of values taken by them. Note that these policies are conservative, which is in line with our design rationale.

3.3.3 Recursive Clustering by Format Distinguishers

We identify FD tokens with the following algorithm. First, we invoke format inference on the set of messages in a cluster. Then, we scan the format token by token from left to right to identify FD tokens. We use three criteria in determining if a token is a FD:

1. We first check if the number of unique values taken by this token across the set of messages is less than a threshold, referred to as the maximum distinct values for a FD token. This is because a FD token typically takes a few values corresponding to the number of different formats.

2. For tokens satisfying the first criterion, we perform a second test as follows. The cluster is divided into subclusters, one for each unique value taken by this token. Each subcluster consists of messages where the candidate FD token takes a specific value. We then require that the size of the largest subcluster exceeds a threshold, referred to as the minimum cluster size. This is to guarantee that we can make a meaningful format inference in at least one subcluster. Otherwise, we gain nothing by continuing this splitting.

3. If the potential FD token passes the second criterion, we invoke format comparison across subclusters to see if their formats are different from each other. We then merge those that manifest the same formats and leave others intact.

This process is recursively performed on each of the subclusters because a message may have more than one FD token. We find the next FD token by scanning further down the message towards the right (end) of the message. It is necessary to scan all the way to the end because we need to recognize all FDs to obtain a good clustering and format inference.

When looking for the next FD token, the format inference is invoked again on the set of messages in each subcluster. This is because the inferred token properties and semantics might change because the set of messages has become smaller, and it is possible for stronger properties to hold. For instance, a previously variable token might now be a constant token; a previously variable token might now be identified as a length field.

3.4 Merging with Type-Based Sequence Alignment

In the tokenization and recursive clustering phases, we are conservative to ensure that the format inference procedure operates correctly on a set of messages of the same format. However, this leads to a new problem of over-classification, namely, messages of the same format may be scattered into more than one cluster. This problem can be quite severe; for instance, over a CIFS/SMB trace of almost four million messages, there are about 7,000 clusters/formats as input to this phase, while the total number of true formats is 130. The goal of the merging process is to coalesce similar formats from different clusters into a single one.

The key observation behind our merging phase is that, while sequence alignment [15] cannot be used for clustering messages of the same format, it can be used to align *formats* for identifying similar ones across different clusters. This is because we can leverage the rich token types (i.e., semantics and properties) inferred in the recursive clustering phase. For instance, knowing that a particular token is a length field in a format necessitates that its counterpart in another format is also a length field for these two formats to be considered a match. We refer to our algorithm for aligning formats as *type-based* sequence alignment.

In our type-based sequence alignment, we only allow two tokens of the same class (binary or text) to align with each other. We claim two aligned tokens are matched if they either have the same semantic or share at least one value (see Section 3.3.2 for details).

To compensate for tokenization errors, we allow gaps in our type-based sequence alignment. In addition to using gap penalties to control gaps, we introduce extra constraints to avoid excessive gaps. First, consecutive binary tokens in one message format are allowed to align with gaps if they precede or follow a text token in the other message format in the alignment, and the number of binary tokens is at most the size of the text token if the text token is aligned with a gap, or the size difference if it is aligned with another text token. This constraint is for handling the case of mistaking a sequence of binary tokens to be a text token or vice versa. Second, a text token is allowed to align with a gap, but we allow at most two gaps of this kind. This constraint is for handling the case that a text field consisting of some white space characters is mistakenly divided into multiple tokens.

When we align and compare two message formats to decide whether to merge them, we first check if the gap constraints can be satisfied. If no, we stop and claim the two formats are mismatched; otherwise, we continue to check the number of mismatches. If there is at most one pair of aligned tokens mismatched, we claim the two formats are matched and merge them. Note that this is conservative because the mismatched token can be treated as a variable token that takes values from a new set covering both formats.

Since we use the gap constraints and the number of mismatches to decide whether to merge two message formats, our merging performance is insensitive to sequence alignment parameters—scores for match, mismatch and gap.

3.5 An Example

For better understanding, here we present a concrete example based on the SMB "Tree Connect AndX Request" message format to explain the design and output of Discoverer. We obtain the true message format from Ethereal (see Figure 2 and Figure 3). The final inferred format by Discoverer is shown in Table 1.

We can see that the inferred format is a sequence of tokens with token properties (binary vs. text, constant vs. variable) and semantics (e.g., length fields). For tokens with unknown semantics, their possible values are also taken into account in the format. Before the merging step, messages of this true format were scattered into 24 clusters in 18 different token patterns. Different token patterns are due to the "smb.signature" field. Since this field may take any random values, we will have a different token pattern when more than three consecutive bytes at a different offset take values from the printable ASCII range and are wrongly treated as a text token. Messages in some token patterns were further split into fine-grained clusters in the recursive clustering phase due to our conservative approach. Our merging technique mitigates this over-classification problem effectively. At the end, all of the 24 clusters were merged into a single one.

This example also shows the possibility of imprecise field boundaries. For example, the first null byte of the field "smb.nt_status" was treated as the null terminator for the text token before it. However, we believe this kind of imprecision will not affect the effectiveness of the inferred format but instead create some extra inferred formats with different values for "smb.nt_status".

4 Evaluation

We implemented Discoverer in 5,700 lines of C++ code on Windows. The tool takes a network capture file either in the libpcap [12] or Netmon [14] format as input and outputs inferred message formats: a sequence of tokens with the inferred properties and semantics. Our

```
<proto name="nbss" showname="NetBIOS Session Service" size="4" pos="54">
 <field name="nbss.type" showname="Message Type: Session message" size="1" pos="54" show="0" value="00"/>
 <field show="Length: 156" size="3" pos="55" value="00009c"/>
</proto>
<proto name="smb" showname="SMB (Server Message Block Protocol)" size="156" pos="58">
 <field show="SMB Header" size="32" pos="58">
  <field show="Server Component: SMB" size="4" pos="58" value="ff534d42"/>
  <field name="smb.cmd" showname="SMB Command: Tree Connect AndX (0x75)" size="1" pos="62" show="0x75"
value="75"/>
  <field name="smb.nt_status" showname="NT Status: STATUS_SUCCESS (0x00000000)" size="4" pos="63"
show="0x00000000" value="00000000"/>
  <field show="Flags: 0x18" size="1" pos="67" value="18">
  <field show="Flags2: 0xc807" size="2" pos="68" value="07c8">
  <field name="smb.pid.high" showname="Process ID High: 0" size="2" pos="70" show="0" value="0000"/>
  <field name="smb.signature" showname="Signature: 05A09637B7419166" size="8" pos="72"
show="05:a0:96:37:b7:41:91:66" value="05a09637b7419166"/>
  <field name="smb.reserved" showname="Reserved: 0000" size="2" pos="80" show="00:00" value="0000"/>
  <field name="smb.tid" showname="Tree ID: 0" size="2" pos="82" show="0" value="0000"/>
  <field name="smb.pid" showname="Process ID: 65279" size="2" pos="84" show="65279" value="fffe"/>
  <field name="smb.uid" showname="User ID: 2048" size="2" pos="86" show="2048" value="0008"/>
  <field name="smb.mid" showname="Multiplex ID: 128" size="2" pos="88" show="128" value="8000"/>
 </field>
 <field show="Tree Connect AndX Request (0x75)" size="124" pos="90">
  <field name="smb.wct" showname="Word Count (WCT): 4" size="1" pos="90" show="4" value="04"/>
  <field show="AndXCommand: No further commands (0xff)" size="1" pos="91" value="ff"/>
  <field name="smb.reserved" showname="Reserved: 00" size="1" pos="92" show="00" value="00"/>
  <field name="smb.andxoffset" showname="AndXOffset: 156" size="2" pos="93" show="156" value="9c00"/>
  <field name="smb.connect.flags" size="2" pos="95" value="0c00">
  <field name="smb.pwlen" showname="Password Length: 1" size="2" pos="97" show="1" value="0100"/>
  <field name="smb.bcc" showname="Byte Count (BCC): 113" size="2" pos="99" show="113" value="7100"/>
  <field name="smb.password" showname="Password: 00" size="1" pos="101" show="00" value="00"/>
  <field name="smb.path" showname="Path: \\SP-SIN-DCF-01.SOUTHPACIFIC.CORP.MICROSOFT.COM\IPC$"
size="106" pos="102" show="\\\\SP-SIN-DCF-01.SOUTHPACIFIC.CORP.MICROSOFT.COM\\IPC$" value="5c005c00..../>
  <field name="smb.service" showname="Service: ?????" size="6" pos="208" show="?????" value="3f3f3f3f3f00"/>
 </field>
</proto>
```

Figure 2: Ethereal's XML output of an example SMB "Tree Connect AndX Request" message (edited for better presentation).

current un-optimized implementation takes about 6-12 hours for a trace of several million messages (the merging procedure is the slowest due to the need of pairwise comparisons of all inferred formats). Before discussing the experimental results, we first describe our data sets and evaluation methodology.

4.1 Data Sets

We tested Discoverer on traces from two different sites: a honeyfarm site [2] (which responds to unsolicited, mostly malicious traffic) and a busy enterprise (which has diverse and high-volume traffic). The honeyfarm trace consists of CIFS/SMB only. The enterprise trace includes HTTP, CIFS/SMB, and RPC. The honeyfarm trace and the HTTP trace were used as the *calibration* data to help guide the design process and set tunable parameters. Our results are presented based on the output of our tool on the traces from the enterprise site, which served as the *evaluation data*. Thus, CIFS/SMB can be seen as the evaluation case where the tool was trained on the trace from a different site, whereas RPC is the case where the tool is evaluated over a new protocol. Though CIFS/SMB messages may encapsulate the RPC layer, the RPC trace consists of RPC traffic exclusively. The HTTP trace was used for both calibration and evaluation, but we hardly tailored our tool to HTTP.

In our experiment, the CIFS/SMB and RPC trace from the enterprise site contains traffic in one direction only. This will not affect our evaluation because the protocol formats in both directions are equally complicated based on Ethereal's parsing results of the honeyfarm CIFS/SMB trace. This is not to say that if we can infer the format in one direction, we are guaranteed to infer the format in the other direction; but the performance in one direction does give an indication of the performance in the other direction. In addition, since we do not put messages in different directions into the same cluster, unidirectional traffic does not make the problem any easier.

For the HTTP trace, our tool reassembled consecutive data sent in one direction into a message. For the CIFS/SMB and RPC traces, we leveraged Ethereal to parse them and identify message boundaries. A summary of these traces is shown in Table 2.

```
nbss.type;Length;Server Component;smb.cmd;smb.nt_status;smb.flags;smb.flags2;smb.pid.high;
smb.signature;smb.reserved; smb.tid;smb.pid;smb.uid;smb.mid;smb.wct;AndXCommand;smb.reserved;
smb.andxoffset;smb.connect.flags;smb.pwlen;smb.bcc;smb.password;smb.path;smb.service
```

Figure 3: The "name" of the true format for the example message in Figure 2 concatenates the human readable names of all the fields.

Token	True Field	Token	True Field	Token	True Field	Token	True Field
C(b,00)	nbss.type	C(b,00)	smb.pid.high	C(b,00)	smb.tid	C(b,00)	
C(b,00)	Length	C(b,00)		C(b,00)		V(b,2)	smb.connect.flags
C(b,00)		V(b,256)	smb.signature	C(b,ff)	smb.pid	C(b,00)	
L(b)		V(b,256)		C(b,fe)		C((b,01)	smb.pwlen
C(b,ff)	Server Component	V(b,256)		V(b,33)	smb.uid	C(b,00)	
C(tn,SMBu)	smb.cmd	V(b,256)		V(b,32)		L(b)	smb.bcc
C(b,00)	smb.nt_status	V(b,256)		V(b,13)	smb.mid	C(00)	
C(b,00)		V(b,256)		V(b,211)		C(00)	smb.password
C(b,00)		V(b,256)		C(b,04)	smb.wct	V(tun,664)	smb.path
C(b,18)	smb.flags	V(b,256)		C(b,ff)	AndXCommand	C(tn,?????)	smb.service
C(b,07)	smb.flags2	C(b,00)	smb.reserved	C(b,00)	smb.reserved		
C(b,c8)		C(b,00)		L(b)	smb.andxoffset		

Table 1: Discoverer's inferred format for the true format in Figure 3. For C(x,y), C means constant, x means binary ("b") or text ("t"; in text tokens, "u" means Unicode and "n" means it is null terminated), y is the hex value or string of the token; for V(x,z), z is the number of different values for the token.

Protocol	Source	Size (B)	# Messages	# True Formats
HTTP	Enterprise	4.6G	5,950,453	2,696
RPC	Enterprise	179M	351,818	50
CIFS/SMB	Enterprise	1.0G	3,818,267	301
CIFS/SMB	Honeyfarm	1.1G	1,439,744	1220

Table 2: Summary of network traces used in the evaluation.

4.2 Evaluation Methodology

Our evaluation methodology is to compare the quality of our output with the set of *true* message formats. To obtain the true format, instead of trying to manually extract it from documentation and RFCs, we used the protocol analyzers in Ethereal [5]. Ethereal can parse a network trace and produce, for each message in the trace, an XML output that includes the list of fields in the message, the values of those fields, some human readable names and their sizes. Based on this output, we assign every message a true format "name", which is simply the concatenation of the human readable names of all the fields. An example of Ethereal's XML output and the true format name is shown in Figure 2 and Figure 3.

We characterize the performance of our tool and highlight the results in the following metrics:

- *Correctness*: If a cluster contains messages from more than one true format, then Discoverer will make incorrect inference. Thus we measure the correctness by checking the number of different true formats followed by the messages in a cluster. For all three protocols, over 90% clusters contain messages from a single true format.

- *Conciseness*: Our conservative clustering may cause multiple inferred formats to cover subsets of a single true format. A large number of redundant formats will affect the conciseness of the protocol specifications generated by our tool. Thus we measure conciseness by the ratio from the number of inferred formats to the number of true formats followed by their messages. For all three protocols, we achieved a low 5 to 1 ratio.

- *Coverage*: We measure the trace coverage from two perspectives: the fraction of messages covered by our inferred formats and the fraction of true formats

Parameter	Value
Maximum message prefix	2048 bytes
Minimum length of text segments	3 letters
Minimum cluster size	20 messages
Maximum distinct values for FD	10
Alignment match score	1
Alignment mismatch score	0
Alignment gap score	-2

Table 3: Summary of parameters.

followed by covered messages. For all the three protocols, our message coverage is above 95% while our format coverage is around 30-40%.

As for the semantic inference, all the length fields inferred by Discoverer are correct; certain length fields are missed due to the trace limitation. For instance, some true formats in CIFS/SMB have a fixed message size. In this case, Discoverer will treat the length fields that reflect the message size as constant tokens, and it will not affect parsing messages of these formats in practice.

4.3 Tunable Parameters

Discoverer has just a few tunable parameters (see Table 3). For a message larger than 2048 bytes, we only consider the first 2048 bytes, referred to as the maximum message prefix. The minimum length of text segments controls the tokenization procedure (Section 3.2.1). The minimum cluster size and the maximum distinct values for FD are used in the recursive clustering phase (see Section 3.3.3). The match/mismatch/gap scores are parameters for sequence alignment [15]. We observed that the performance of Discoverer is not sensitive to the settings of these parameters. For instance, we saw similar performance when we changed the maximum prefix size from 2048 bytes to 1024 bytes or changed the minimum cluster size from 20 messages to 10 messages. In addition, our type-based sequence alignment is not sensitive to the match/mismatch/gap scores as we discussed in Section 3.4. Thus we take the same parameters for our evaluations on all three protocols.

In the rest of this section, we present the experimental results on the enterprise traces for HTTP, RPC, and CIFS/SMB. Note that we use the inferred format and cluster interchangeably because we infer one format from each cluster.

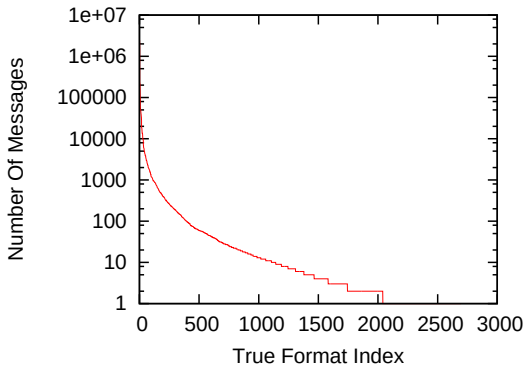

Figure 4: Heavy-tail distribution of message format popularity in HTTP.

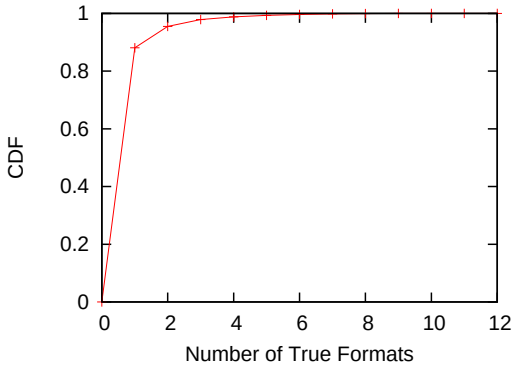

Figure 5: Correctness for HTTP: CDF of the number of true formats followed by messages of a cluster for all clusters before the merging phase.

4.4 HTTP

The HTTP protocol allows an arbitrary number of "parameter: value" pairs in an arbitrary order. We refer this to be the "set" semantic. Currently we are unable to identify this set semantic. So we treat each ordering of the set elements as a distinct format. By doing so, we observed 2,696 formats from the parsing results of Ethereal. We leave the identification of set semantic to be future work (see Section 6).

In Figure 4, we show the number of messages of each true format in the HTTP trace. Note that the y-axis is in logarithmic scale. This clearly reveals the heavy-tail distribution; most messages (more than 99%) fall in the first top 1000 true formats. We observed a similar trend in the RPC and CIFS/SMB trace as well. The implication for our tool is that the format coverage and message

coverage are likely to be very different; the latter will be much higher compared to the former. In HTTP, we inferred 3,926 formats, which covered 5,938,511 out of 5,950,453 messages (99.8%). The covered messages belong to 865 out of 2,696 true formats (32%). Since we have a hard requirement on the minimum size of a cluster, we conjecture that the coverage ratio in terms of true formats will improve when the trace grows and each format has more messages.

Figure 5 plots the CDF of the number of true formats followed by messages of a cluster for all clusters before the merging phase. This reflects the correctness of our tool. This figure shows that about 90% of our inferred clusters are correct. They correspond to only one true format. The number rises to over 95% if we include inferred formats that match two true formats.

By manually inspecting the results, we found that the clustering errors are mainly due to the inaccuracy in Ethereal parsing. For example, in some message formats, Discoverer infers that there is a token that can be either "Connection" or "Proxy-Connection". Discoverer does not treat it as a FD because both may be followed by the same set of values such as "Close" or "Keep-Alive". However, Ethereal does not recognize "Proxy-Connection" as a parameter for HTTP, and returns a null string for this field in its parsing result, while it returns "http.connection" for "Connection". So we will have two true formats for a cluster that contains both "Connection" and "Proxy-Connection". Thus, our conciseness number may improve if Ethereal has more accurate parsing.

The merging phase reduced 4,465 clusters to 3,926 clusters. Since the covered messages belong to 865 true formats, this gives us a 5 to 1 ratio. In fact, almost 80% true formats are scattered into at most five clusters. On one hand, our conservative strategy eliminated false positives (i.e., wrongly merging two clusters that correspond to two different true formats). On the other hand, it did not help much in merging clusters for HTTP. The reason is as follows. HTTP allows many parameters in the form of "parameter: value". We treat the "parameter:" and "value" as separate tokens because of the space in between. Since the "value" token for certain parameters such as "PROXY" may be arbitrary strings, it is likely for such "value" tokens in two clusters to not have overlapped values. In this case, we will treat them as a mismatch. If two clusters happen to have more than one such mismatch, we will not merge them based on our conservative policy.

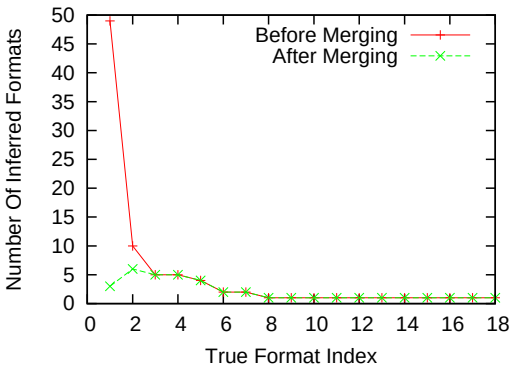

Figure 6: Effectiveness of merging for RPC: Number of inferred formats into which messages of a single true format are "scattered": before and after merging.

4.5 RPC

The RPC trace consists of exclusively RPC traffic. Though the trace size is 179MB, one order less than the HTTP and CIFS/SMB trace, we observed the similar trend in the distribution of the number of messages in each true format. Overall, we inferred 33 formats, which covered 340,624 out of 351,818 messages (96.8%). The covered messages belong to 18 out of 50 true formats (36%).

The recursive clustering generated 83 clusters, among which 78 clusters contain messages from a single true format, and the rest five clusters have messages from two true formats. The merging phase helped reduce the overall clusters from 83 to 33 without introducing false positives. This shows that our merging phase compensates the tokenization errors well by recognizing wrongly classified binary and text tokens. From Figure 6 we can see that, for each of 11 true formats, its messages were merged into in a single cluster.

4.6 CIFS/SMB

CIFS/SMB is a fairly complex binary protocol which includes several layers of protocols: it consists of the NetBIOS Session Service (NBSS) headers which encapsulate a SMB header which in turn is layered over RPC. Overall, we inferred 679 formats, which covered 3,640,239 out of 3,818,267 messages (95.3%). The covered messages belong to 130 out of 301 true formats (43%).

In Figure 7, we plot the CDF of the number of true formats followed by messages of a cluster for all clusters before the merging phase. We can see that 57%

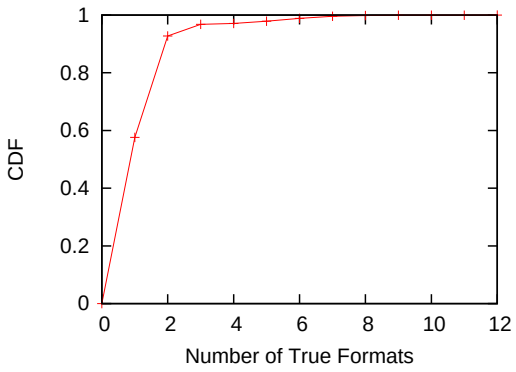

Figure 7: Correctness for CIFS/SMB: CDF of the number of true formats followed by messages of a cluster for all clusters before the merging phase.

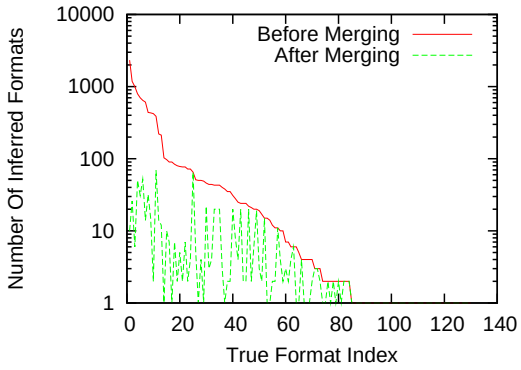

Figure 8: Merging effectiveness for CIFS/SMB: Number of inferred formats into which messages of a single true format are "scattered": before and after merging

clusters contain messages from a single true format, and 35% clusters have messages from two true formats. We manually checked these clusters and found that it is due to the imprecise parsing of Ethereal. It recognized the last field as a "dcerpc.nt.close.frame" field for some messages and as a stub data for other messages while those messages have the same format according to our manual inspection. If we take into account this factor, more than 90% clusters contain messages from a single true format, which is consistent with HTTP.

We further inspected the clusters consisting of messages from more than two formats and found that, for many of such clusters, the only difference in the true formats followed by their messages is the last field. It is in the form of "Stub data (XX bytes)", and the difference is in "XX" which says the size of the stub data. Based on our manual inspection, we conjecture that these stub data likely follow the same format and the size difference is due to a text field with a variable length embedded in the stub data. Therefore, 90% is a conservative measure on the correctness.

In Figure 8, we plot the number of inferred formats into which messages of a single true format are scattered before and after merging. Like RPC, the merging technique is effective on CIFS/SMB. Overall, we reduced the number of clusters from 7180 to 679 without introducing false positives, which gives us a 5 to 1 ratio against the 130 true formats.

5 Related Work

We divide related work into three categories. First, we discuss the state of the art in protocol reverse engineering. Second, we present the previous work that was geared towards a specific application rather than performing all-purpose protocol reverse engineering. Finally, we discuss grammar inference.

To date, most protocol reverse engineering appears to be a painstaking manual process, which involves looking at available documentation, source code, and traces. Two popular examples in the community include the SAMBA project [18] and the messenger clients [6]. Automatic protocol reverse engineering appears to have received much less attention. The most closely related work to our paper that we are aware of is the Protocol Informatics project [17]. It aims to employ sequence alignment techniques to infer protocol formats from a trace of the protocol. Its main limitation is that the byte-wise sequence alignment, while ideally suited to aligning messages with similar *byte sequences*, is not suitable for aligning messages with similar *formats*. In addition, selecting weights to tune the alignment is hard as shown in [3].

Previous studies have also performed some level of protocol reverse engineering with a specific purpose in mind, namely, application-level replay and protocol identification.

RolePlayer [3] and ScriptGen [10, 11] both leverage byte-wise sequence alignment techniques to achieve application-level replay by heuristically detecting and adjusting some specific fields such as network addresses, lengths, and cookies. A driving application for application-level replay is to build a protocol-independent, application-level proxy to filter known attacks in a honeyfarm. To improve the performance of the Needleman-Wunsch algorithm [15], RolePlayer uses the so-called *pairwise constraint matrix*, which specifies whether the ith byte of the first message can or cannot be aligned with the jth byte of the second message based on

the field semantic of the ith byte. However, if the semantic of the ith byte in the first message is unknown, it can be aligned with any byte in the second message, which may lead to alignment errors. There are two key differences between Discoverer and these two systems. First, RolePlayer and ScriptGen only discover the protocol format to the extent necessary for replay, while Discoverer is aimed to discover the complete protocol format. Second, instead of using the byte-wise sequence alignment, we first cluster messages based on token patterns and then use a novel type-based sequence alignment technique to align and compare message formats based on token types. This represents a significant improvement: on one hand, we can avoid byte-pattern alignment in the recursive clustering phase to achieve a good performance on correctness; on the other hand, we can mitigate overclassification by merging similar inferred formats. Furthermore, compared with ScriptGen which clusters messages by comparing the whole messages at once, our *recursive* clustering technique performs better because we not only look at the potential FD token itself but also look into "the future" by comparing the subsequent tokens in the messages. Some of our techniques for identifying semantically important fields (such as length fields) are borrowed from RolePlayer.

Ma *et. al.* [13] perform protocol identification, that is, they classify the set of sessions in a trace into various protocols without relying on port numbers. They develop three techniques for profiling messages exchanged in a protocol: product distributions of byte offsets, Markov models of byte transitions, and common substring graphs of message strings. The main difference between their work and ours is that we have different goals. They aim to characterize a protocol based on the first n (e.g., 64) bytes in sessions of the protocol; we leverage the format inference and type-based sequence alignment techniques to decipher the message formats of the entire session.

The problem of protocol reverse engineering is related to the theoretical problem of grammar inference, which aims to deduce the grammar given a set of sample strings drawn from it. This problem is unfortunately theoretically unsolvable, even when the grammar is in the simplest form of Chomskian grammar, the regular language [7]. Since even a regular language can be potentially infinite and the sample set cannot be, it turns out that this task is impossible. The language implicit in application-level protocols is often substantially more complex than a regular language, involving fields such as length fields. Because of this complexity, we were unable to directly apply any results from the grammar inference community. There have been extensions based on Kolmogorov complexity [4] to learn the simplest finite language from only positive examples, but once again, they appear too complicated to apply to the context sensitive grammars that network protocols involve.

Techniques used in the speech recognition community, such as, probabilistic Markov chain analysis [8], were not applied in our work, since the correlation between protocol fields makes it difficult for the byte sequence in a message to be modeled as independent samples from a Markov process.

6 Limitations and Future Work

In this section, we discuss the limitations of our approach. We categorize our limitations into two categories: ones that are fundamental to the problem we want to solve and those that are due to the heuristics in our tool. We will also describe future research directions for solving these limitations.

There are two main fundamental limitations.

- *Trace Dependency*: The format generated by any tool that operates only on the trace is limited by the diversity of traffic seen in the trace. If certain message formats never occur in the trace, or if certain variable fields never take more than one value in the trace, it is impossible for such a tool to infer those message formats or identify those fields as variable fields.

- *Pre-Defined Semantics*: Only a set of pre-defined semantics can be inferred. Since it is not possible to find all the possible semantics of all fields just from a trace, the best one can hope for is to have an extensible framework where new semantic modules can be added as desired.

We now move on to the imprecision problems that are directly related to the design of our tool. The following are the major imprecisions in our inferred message formats:

- *Semantics*: At present, we cannot capture the following semantics. (a) Set semantics: For instance, HTTP allows an arbitrary number of parameters to be specified in any order. Identifying this list of supported parameters as a set that allows re-ordering during encoding would considerably improve our performance. (b) Pointer field: This is a field whose value is the offset of another field in an array of some arbitrary items. Such fields occur in DNS. (c) Array length: This is a field whose value is the number of items in an array of some arbitrary items (e.g., DNS). We plan to study the inference of these semantics in the future.

- *Coalescing Fields*: We identify a binary field as a sequence of binary tokens each spanning a single byte. This is a limitation since ideally we would want such a field identified as a single binary token. Unlike text fields, no clue may be available in delimiting binary fields. The only way out is techniques based on frequency analysis (e.g., does this byte vary as much as the other one?). However, this kind of techniques tend to be unreliable. For instance, in a two-byte process ID field, the more significant byte may change much less frequently than the less significant one since an operating system usually issues process IDs incrementally from zero. Thus we chose to list such fields as a series of single byte tokens. Our plan is to enrich our semantic inference modules so that a sequence of binary bytes with a common semantic can be identified as a single field. For example, a length field spanning four bytes will be identified as a single field because of the semantic module for detecting length fields.

- *Asynchronous Protocols*: With asynchronous protocols, it is difficult to even delimit messages from network packets. This is because messages in one direction may be interrupted by those in the other direction, and messages in one direction may be delayed allowing two back-to-back messages in the other direction. We have not experimented with any asynchronous protocols so far.

- *Application Sessions*: Currently, our tool analyzes each connection in isolation. However, if we had a good session description of the various connections and various hosts involved, it would be trivial to process the trace with such session knowledge, and derive formats for the whole session. A previous study [9] aimed to semi-automatically discover session structures.

- *State Machine Inference*: Currently, we only envision a state machine constructed from the trace by using the inferred message formats to assign a type to each message, and then simply inferring the FSA that captures the sequences of messages in all sessions in the trace. However, this is hardly the compact FSA that the application developer had in mind. In such case, using FSA minimization techniques [19] may simplify the FSA considerably.

Many of the limitations above are due to the limited information available from network traces. To tackle these limitations and achieve a better reverse-engineered protocol specification, we can use program analysis to gain more information and insight into the parsing and processing of the input in the program. For instance, we may easily identify two consecutive bytes as a WORD (i.e., a two-byte integer) from run-time analysis by observing that they are processed as a WORD throughout the execution.

We have focused on reverse engineering network protocols in Discoverer; it would be useful to reverse engineer the input specifications for file-based applications, since we have seen a significant growth in file-based attacks.

7 Conclusion

Protocol reverse engineering is a highly manual process today, which is still suffered through because of the immense value of protocol knowledge. We have developed Discoverer, a tool that aims to automate this reverse engineering process. Discoverer leverages recursive clustering and type-based sequence alignment to infer message formats. We have demonstrated Discoverer can infer message formats effectively for three network protocols, CIFS/SMB, RPC, and HTTP. In the future, we plan to enrich our semantic inference, research on the protocol state machine inference, explore the direction of using program analysis to reverse engineer the specifications of both network and file input, and apply reverse-engineered protocol specifications to real world applications.

Acknowledgments

We would like to thank Vern Paxson, Ion Stoica, Christian Kreibich, and Gautam Altekar for their valuable comments on an early draft of this paper. We thank Joseph Kravis for providing network traces to us. We would also like to thank the anonymous reviewers for their insightful comments.

References

[1] N. Borisov, D. J. Brumley, H. J. Wang, J. Dunagan, P. Joshi, and C. Guo. A Generic Application-Level Protocol Analyzer and its Language. In *Proceedings of the 14th Annual Network & Distributed System Security Symposium (NDSS)*, March 2007.

[2] W. Cui, V. Paxson, and N. Weaver. GQ: Realizing a System to Catch Worms in a Quarter Million Places. Technical Report TR-06-004, ICSI, 2006.

[3] W. Cui, V. Paxson, N. C. Weaver, and R. H. Katz. Protocol-Independent Adatpive Replay of Application

Dialog. In *Proceedings of the* 13^{th} *Symposium on Network and Distributed System Security (NDSS 2006)*, February 2006.

[4] F. Denis. Learning Regular Languages from Simple Positive Examples. *Machine Learning*, 44(1/2):37–66, 2001.

[5] Ethereal: A Network Protocol Analyzer. `http://ethereal.com`.

[6] Gaim Instant Messaging Client. `http://gaim.sourceforge.net`.

[7] E. M. Gold. Language Identification in the Limit. *Information and Control*, 10(5):447–474, 1967.

[8] F. Jelinek and J. D. Lafferty. Computation of the Probability of Initial Substring Generation by Stochastic Context-Free Grammars. *Computational Linguistics*, 17(3):315–323, 1991.

[9] J. Kannan, J. Jung, V. Paxson, and C. E. Koksal. Semi-Automated Discovery of Application Session Structure. In *Proceedings of the 2006 Internet Measurement Conference (IMC)*, Rio de Janeiro, Brazil, October 2006.

[10] C. Leita, M. Dacier, and F. Massicotte. Automatic Handling of Protocol Dependencies and Reaction to 0-Day Attacks with ScriptGen based Honeypots. In *Proceedings of the 9th International Symposium on Recent Advances in Intrusion Detection*, Hamburg, Germany, September 2006.

[11] C. Leita, K. Mermoud, and M. Dacier. ScriptGen: An Automated Script Generation Tool for Honeyd. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC 2005)*, December 2005.

[12] The libpcap Project. `http://sourceforge.net/projects/libpcap/`.

[13] J. Ma, K. Levchenko, C. Kreibich, S. Savage, and G. Voelker. Unexpected Means of Protocol Inference. In *Proceedings of the 2006 Internet Measurement Conference*, 2006.

[14] Microsoft Corporation. Microsoft Network Monitor 3. `http://www.microsoft.com/downloads/details.aspx?FamilyID=aa8be06d-4a6a-4b69-b861-2043b665cb53`.

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

[16] R. Pang, V. Paxson, R. Somer, and L. Peterson. binpac: A yacc for Writing Application Protocol Parsers. In *Proceedings of the 2006 Internet Measurement Conference*, October 2006.

[17] The Protocol Informatics Project. `http://www.baselineresearch.net/PI/`.

[18] How Samba Was Written. `http://samba.org/ftp/tridge/misc/french_cafe.txt`.

[19] B. W. Watson. A Taxonomy of Finite Automaton Minimization Algorithms. Technical Report CS-93-44, Eindhoven University of Technology, Eindhoven, The Netherlands, 1993.