Automatic Network Protocol Analysis

Gilbert Wondracek[§], Paolo Milani Comparetti[‡], Christopher Kruegel[*], and Engin Kirda[¶]

[§] Secure Systems Lab
Technical University Vienna
gilbert@seclab.tuwien.ac.at

[‡] Scuola Superiore S.Anna
pmilani@sssup.it

[*] University of California,
Santa Barbara
chris@cs.ucsb.edu

[¶]Eurecom Institute,
France
engin.kirda@eurecom.fr

Abstract

Protocol reverse engineering is the process of extracting application-level specifications for network protocols. Such specifications are very helpful in a number of security-related contexts. For example, they are needed by intrusion detection systems to perform deep packet inspection, and they allow the implementation of black-box fuzzing tools. Unfortunately, manual reverse engineering is a time-consuming and tedious task. To address this problem, researchers have recently proposed systems that help to automate the process. These systems operate by analyzing traces of network traffic. However, there is limited information available at the network-level, and thus, the accuracy of the results is limited.

In this paper, we present a novel approach to automatic protocol reverse engineering. Our approach works by dynamically monitoring the execution of the application, analyzing how the program is processing the protocol messages that it receives. This is motivated by the insight that an application encodes the complete protocol and represents the authoritative specification of the inputs that it can accept. In a first step, we extract information about the fields of individual messages. Then, we aggregate this information to determine a more general specification of the message format, which can include optional or alternative fields, and repetitions. We have applied our techniques to a number of real-world protocols and server applications. Our results demonstrate that we are able to extract the format specification for different types of messages. Using these specifications, we then automatically generate appropriate parser code.

1 Introduction

Protocol reverse engineering is the process of extracting application-level protocol specifications. The detailed knowledge of such protocol specifications is invaluable for addressing a number of security problems. For example, it allows the automated generation of protocol fuzzers [23] that perform black-box testing of server programs that accept network input. In addition, protocol specifications are often required for intrusion detection systems [25] that implement deep packet inspection capabilities. These systems typically parse the network stream into segments with application-level semantics, and apply detection rules only to certain parts of the traffic. Generic protocol analyzers such as binpac [24] and GAPA [2] also require protocol grammars as input. Moreover, possessing protocol information helps to identify and understand applications that may communicate over non-standard ports or application data that is encapsulated in other protocols [14, 19]. Finally, knowledge about the differences in the way that certain server applications implement a standard protocol can help a security analyst to perform server fingerprinting [29], or guide testing and security auditing efforts [3].

For a number of protocols (e.g., SMTP, HTTP), specifications and corresponding protocol parsers are publicly available. However, there is also a large number of proprietary, closed protocols (e.g., ICQ, SMB) for which such information does not exist. For these protocols, the traditional way of determining a specification involves a significant amount of manual analysis. Obviously, this is a painful and time-consuming task that can only be justified for very popular protocols such as SMB [27].

To address the limitations of manual protocol analysis, automatic protocol reverse engineering techniques have been proposed. The goal of these techniques is to automatically generate the specification of an application-level protocol, given as input one of two different sources: The first source is the application program that implements a particular protocol. So far, researchers have proposed a static analysis approach that takes as input a binary program and outputs the set of inputs that this program accepts [21]. Beside the fact that it is undecidable to statically determine the complete set of inputs for a program, this approach also suffers from significant scalability issues. As a result, in their paper [21], the authors were only able to extract the protocol for very simple and small prototype applications that they themselves developed.

The second source of input for automatic protocol reverse engineering systems is network traffic. More precisely, a number of systems [9, 10, 16, 17] have been proposed that analyze network traces generated by recording the communication between a client and a server. To this end, the network traces are examined for the occurrence of common structures or bytes that indicate a special meaning for the protocol. While experiments have shown that these systems are able to analyze real-world protocols, their precision is often limited. Because of the lack of information that is present in network traces, messages of the same type are sometimes considered different, and data artifacts are falsely recognized as being protocol keywords.

In this paper, we present a novel technique for automatic protocol reverse engineering that aims to combine the precision of systems that analyze application programs with the scalability of systems that examine message instances in network traffic. The basic idea of our approach is to dynamically monitor the application when it is processing protocol messages. That is, we observe *how* a program is processing protocol messages that it receives. We believe that our focus on the analysis of the application is reasonable because the program itself encodes the protocol and represents the authoritative specification of the inputs that it can accept.

Our proposed system operates directly on binary programs. The analysis works by monitoring an application that accepts network input (e.g., a server) while using another program (e.g., a client) to send messages to this application. We use dynamic taint analysis to mark the input data and track how the monitored application propagates and processes this data. Based on the ways in which tainted data is used and manipulated, we extract a specification of the received message. In a second step, the information obtained from several, individual messages is combined to determine a protocol specification for a particular type of message. Finally, this specification is output as a grammar that we use to generate parsing code. In addition, the specification is augmented with automatically generated, semantic information that provides a human analyst with insight into the meaning of different parts of a message. For our experiments, we analyzed large server programs (such as `apache` or `samba`) that implement complex, real-world protocols such as HTTP, DNS, or NFS. Our results demonstrate that we can generate accurate specifications for different messages types such as HTTP GET requests and DNS queries.

The contributions of this paper are the following:

- We present a novel approach for the automated extraction of protocol information. To this end, we dynamically monitor the execution of an application and analyze how it processes messages that it receives.
- We present techniques to automatically split a single message into different protocol fields. Also, we show how information for individual messages can be combined to obtain a more general and abstract format specification.
- We applied our techniques to a set of real-world server applications that implement complex protocols such as HTTP, DNS, SMTP, SMB, and NFS. Our results show that we can automatically generate specifications that can be used to parse messages of certain types.

2 System design

Automatic protocol reverse engineering is a complex and difficult problem. In the following section, we introduce the problem domain and discuss the specific problems that our techniques address. Then, we provide a high-level overview of the workings of our system.

2.1 Problem scope

In [9], the authors introduce a terminology for common protocol idioms that allow a general discussion of the problem of protocol reverse engineering. In particular, the authors observe that most application protocols have a notion of an *application session*, which allows two hosts to accomplish a specific task. An application session consists of a series of individual messages. These messages can have different types. Each message type is defined by a certain *message format specification*. A message format specifies a number of *fields*, for example, length fields, cookies, keywords, or endpoint addresses (such as IP addresses and ports). The structure of the whole application session is determined by the *protocol state machine*, which specifies the order in which messages of different types can be sent.

Using that terminology, we observe that automatic protocol reverse engineering can target different levels. In the simplest case, the analysis only examines a single message. Here, the goal of the reverse engineering process is to identify the different fields that appear in that message. A more general approach considers a set of messages of a particular type. An analysis process at this level would produce a message format specification that can include optional fields or alternative structures for parts of the message. Finally, in the most general case, the analysis process operates on complete application sessions. In this case, it is not sufficient to only extract message format specifications, but also to identify the protocol state machine. Moreover, before individual message formats can be extracted, it is necessary to distinguish between messages of different types.

While it would be very desirable to have a system that can work at the application session level, we leave this for future work. In this paper, we focus on the goal of determining the format specification of a certain type of message in a completely automated fashion. That is, we propose to analyze a set of messages of one type, and extract the format specification for this message type. We believe that automatically finding message format specifications is an ambitious goal that is valuable in practice. For example, it might be sufficient for a fuzzer or an intrusion detection system to understand only messages of a particular type. Also, extracting message formats is a necessary building block for a system that performs complete protocol recovery. Finally, we augment the message format with additional semantic information that provides useful information for a human analysts about the way in which an application uses the data that it receives (e.g., indication that a certain message field is used to hold the name of a file that is accessed).

2.2 System overview

The goal of our system is to extract the format specification for a certain type of message of an unknown protocol. To this end, the system executes a number of steps:

Dynamic data tainting. In the first step, a number of messages are sent to an application that "understands" the protocol that we are interested in (e.g., a server program implementing a particular network protocol). This application is instrumented, and all instructions that operate on input data read from protocol messages are recorded. More precisely, we use dynamic data tainting to track the bytes of the messages that are read by the application. Similar to previous systems that use tainting [5, 7, 8], each input byte receives a unique label. Then, we keep track of each labeled value as the program execution progresses. As a result of the dynamic data tainting process, an execution trace is produced for each message. This trace contains all operations that have one or more tainted operands. For more details on dynamic data tainting, please refer to Appendix B.

Analysis of individual messages. In the next step, our system analyzes the individual execution traces that are produced for each message. The goal is to leverage the information derived from the way in which the application processes its input to identify the constituent parts of a message. Many protocols make use of delimiter bytes to group the sequence of input bytes into individual fields. Others use length fields to indicate the length of a target field. In addition, protocols can also define a sequence of fixed-length fields. In this case, neither delimiters nor length fields are necessary for the receiver to correctly parse a message. Of course, a protocol can make use of both delimiters and length fields. Moreover, fields can be nested.

By observing how the application processes the message, we attempt to identify delimiters and length fields, as well as the structure they impose onto the message. Furthermore, we extract semantic information for different fields. For example, we can determine when a field contains a protocol keyword, is used to access a file in the file system, or is directly echoed back to the party that the application is communicating with. Our techniques to analyze single message instances are discussed in detail in the following Section 3.

Multiple messages and message format specification. In the third and last step, we combine the information derived for messages of one type to generate a more general format specification. The reason for considering multiple messages is that it is possible that different messages of the same type do not always contain exactly the same number of fields in exactly the same order. To generate a general and comprehensive message format specification, the differences in the individual messages have to be "abstracted away." For this, we compare the results for multiple runs, using an alignment algorithm from the bio-informatics community. The goal is to align similar fields, thereby identifying alternative parts that vary between messages, optional fields, or fields that appear a different number of times. The result of the alignment step is a more general specification, which would not be possible to infer from a single message only. This specification is then output as a regular expression that serves as input for a protocol parser. A more detailed explanation of this process is given in Section 4.

3 Analysis of a single message

When the monitored application has processed a message, the first task of our system is to use the execution trace produced by the dynamic taint analysis to split this message into its components, or fields. Most network protocols use delimiters or length fields (or a combination of both) to impose a structure onto the sequence of bytes that make up the input message. Thus, we have developed two techniques to locate such delimiter fields and length fields in the message. These techniques are discussed in the two following subsections. Once a message is decomposed, the next step is to derive additional semantic information for the fields (discussed in Section 3.3).

3.1 Finding delimiters

A *delimiter* is a byte (sometimes also a sequence of bytes) with a known value that indicates the end of a protocol field. For example, consider the HTTP GET request that is shown in Figure 1 below. In this example, the newline delimiter '`\r\n`' divides the GET request into two lines. Moreover, the space character is used to split the three components of the first line (GET method, requested resource, and HTTP version). When parsing a message, the application searches each consecutive byte in the input stream for the occurrence of a byte with the known delimiter value. Once this byte is found, the application recognizes that the end of a field is reached, and can continue accordingly. This observation directly translates into our approach to identify delimiters.

```
GET /index.html HTTP/1.1\r\n
Host: 127.0.0.1\r\n\r\n
```

Figure 1. HTTP GET request.

To find delimiters, we examine the execution trace of the application for operations that compare a tainted input byte with an untainted value. For each comparison operation, we record the location of the input byte in the message (based on its unique label), as well as the value of the operand. Based on this information, we can create, for each of the 256 possible byte values (single characters), a list that stores the labels that this character was compared against. In other words, we record, for each possible delimiter character, the positions of the bytes in the message that it was compared against. For example, assume that we observe that the application compares the first three bytes of the message against character 'a', and the fourth byte against 'b'. This fact is recorded by adding the labels 0, 1, and 2 to the list that corresponds to character 'a', and by adding label 3 to the list for 'b'. Note that it is possible for the same input byte to be compared against several characters. In this case, the same label is added to multiple lists.

Once all comparison operations are recorded, we traverse each list and check it for the occurrence of consecutive labels. Consecutive labels are merged into intervals. Labels that are not part of any interval are discarded. The assumption is that an application has to scan at least two consecutive input bytes for a particular value when this value is a delimiter. This is because a delimited field should be at least one byte long, and the delimiter itself occupies a second position. In the example introduced in the previous paragraph, we would create the interval [0,2] for character 'a' and discard label 3 in the list for 'b'.

Scopes and delimiter hierarchy. The intervals that are computed for each character indicate regions, or scopes, in the input message where this character is used as delimiter. We call such intervals *scope fields*. A certain delimiter character can be present multiple times in the scope field of a message. In this case, this delimiter splits the scope field into multiple parts. These individual parts are referred to as *delimited fields*. Furthermore, scopes for *different* delimiter characters can overlap, indicating that a delimited field is further broken up into multiple, smaller parts delimited by another character. In other words, a delimited field can itself be a scope field for another character.

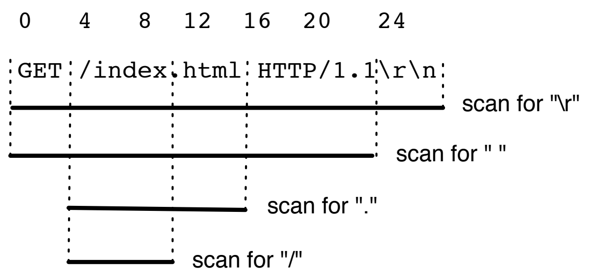
As example, consider the HTTP GET request in Figure 2. One can see that the `apache` web server checks different parts of the message for the occurrence of different delimiter characters. The sequence '`\r\n`' is used to split the entire message into lines, and thus, the server compares every byte of the message against '`\r`'. Hence, the message is a scope field for the character '`\r`', and each line is a delimited field. Then, the space character is used to further split the first line into three parts. Thus, the first line is not only a delimited field (with regard to the '`\r`' delimiter), but also a scope field (with regard to the space character). The complete set of scopes for the exemplary request are shown in the top, left part of Figure 2. The corresponding intervals are depicted on the bottom left.

When extracting the structure of a message, it would be desirable to obtain a hierarchy of fields that reflects nested scopes. To determine such a relationship between scope fields, we analyze the relationship between the intervals that belong to different delimiter characters. When one interval is a subset of another, the character that belongs to the superset is considered to be the parent delimiter, and its corresponding scope is called the outer scope. In the special case that two intervals are the same, the scope whose corresponding delimiter character is checked first in the execution trace is chosen as the outer scope. It is also possible that two intervals overlap, but neither of the two completely contains the other one. Note that we have never observed such a case in practice. However, if encountered, we would deal with this situation by removing both intervals from further consideration. This is because it is not possible to clearly attribute a section of a message to a certain delimiter in this special case. In case there is no overlap between two intervals, the corresponding scope fields are at the same hierarchical level, and the fields are connected to the scope field that encompasses both. When there is no such scope, they are connected to the root node (which represents the entire message).

Once we have identified an outermost scope (a scope field that is not contained in any other scope), we use the corresponding character to break the scope field into fields separated by that delimiter. Then, we apply the same analysis recursively to all the delimited fields that have been created. In the example of the HTTP GET request, the '`\r`' character corresponds to the outermost scope field. Once this character is used to break the complete request into lines, the analysis proceeds recursively for each line. At this point, the scope that corresponds to the space character is the outermost scope, and as a result, the first line is broken into three fields. Eventually, our analysis produces the hierarchy of scopes shown on the right in Figure 2.

Multi-byte delimiters. Some protocols use delimiters that are longer than a single byte. It is, therefore, necessary to extend single byte delimiters to multi-byte delimiters. We achieve this by checking the bytes before and after all occurrences of a particular delimiter. The delimiter is extended in either direction by the maximum number of preceding/succeeding bytes with constant value over all occurrences of the delimiter. In the HTTP example introduced previously, we would observe that the delimiter byte '`\r`' is always followed by the byte '`\n`'. As a result, the line delimiter is (correctly) extended to contain the multi-byte sequence '`\r\n`'.

Interestingly, certain protocols use multi-byte delimiters that have to occur aligned at some word boundary. That is,



	Initial Intervals
"\r"	[0,25]
" "	[0,23]
"."	[4,15]
"/"	[4,9]

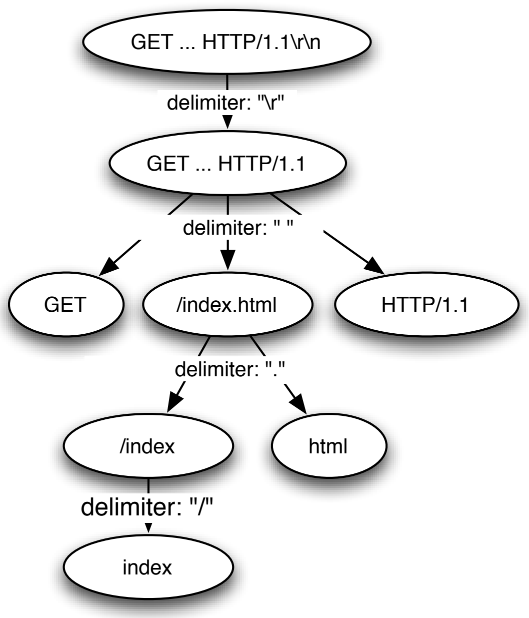


Figure 2. Finding delimiters.

the server does not check for such delimiters at all positions (offsets) in the packet, but only at those that are located at a word boundary. An example of such a delimiter is the Unicode string terminator in the SMB protocol. To detect such delimiters, we use the techniques previously described, but at a word level rather than the byte level (currently only for a word size of two bytes). In other words, we look for comparison instructions with two-byte operands that compare the same constant value against consecutive labels (which are two bytes apart).

3.2 Identifying length fields

An alternative mechanism to structure protocol messages are *length fields*. A length field is a number of bytes that store the length of another field. This other field, from now on referred to as a *target field*, holds a number of bytes or fixed-size records that are a function of the value stored in the length field. The goal of our analysis is to accurately detect both length fields and the corresponding target fields. Initially, we do not make any assumption on the encoding of the length fields, and we do not assume that the target field immediately follows the length field itself. Similar to the case with nested scope fields, a target field can itself contain other fields.

When an application parses a message that contains a length field, this length field is typically used by the program to determine the end of the target field. For example, the length field can be used as a counter that is decremented until it reaches zero. Another possibility is to first compute an "end pointer" by adding the length field to a variable that points to the start of the target field. Then, the program can step through the message until this end pointer is reached. When processing a variable length (target) field, we expect the application to access a number of consecutive bytes of the message. Typically, these repeated accesses are performed in a loop. The condition that determines the end of these accesses (and the exit of the loop) is derived from the length field. This insight is leveraged to specify an algorithm for identifying length fields, together with their target fields.

Static analysis. Because our approach to detect length fields requires the knowledge of loops and loop exit points in the program, an initial static analysis step is required. To this end, we employ a tool that we developed for a previous project [15]. To improve its accuracy with regard to loop detection, a few additional improvements were necessary. First, we implemented the Sreedhar algorithm [28], which correctly handles non-natural/irreducible loops (i.e., loops with multiple entry points). Moreover, we extended the tool with a very simple intra-procedural data flow analysis and a heuristic [6] to recover jump table targets. When the static analysis step terminates, it outputs the program's loops. As a result, we know (a) which comparison instructions represent loop exit points, and (b), the set of loops that each instruction belongs to. Note that a single instruction can be part of multiple loops when loops are nested.

Finding length and target fields. Using the information provided by our static analysis step, we scan the execution trace for all comparison instructions that are both loop exit points **and** that operate on tainted data. When such instructions are found, we know that the corresponding loop is controlled by some bytes of the input. These bytes potentially represent a length field.

We then narrow down the results by further requiring that a loop uses the *same set* of tainted bytes (labels) in its exit condition for *every* iteration. The rationale behind this is that, for length fields, we expect the tainted bytes to appear in every loop iteration. Other uses of tainted data in loop exit conditions, on the other hand, typically do not repeatedly test the same exit condition on the same bytes. Finally, if the taint labels in the set are consecutive, the corresponding input bytes are considered a length field candidate.

Once we have identified length field candidates, we attempt to determine their corresponding target fields. As discussed previously, we can identify those loops that are controlled by a certain length field. Now, we assume that a target field is comprised of all bytes in the message that are accessed by a certain loop. For example, when a length field is used to control a loop that runs a pointer through a target field (either for copying the field, or checking its values), we expect that all of the target field's bytes are accessed by one or more instructions in at least one iteration. Thus, for each length field candidate, we record all the labels (i.e., positions of input bytes) that are "touched" by a loop that is controlled by this length field. By touched, we mean that the label appears as an operand of at least one instruction executed by the loop.

In the next step, we remove all labels that are touched in *every* loop iteration. The reason is that the presence of those bytes is independent of the current loop iteration, and thus, they are likely not related to the target field currently analyzed. As a convenient side effect, this also removes the labels that belong to the length field itself (since, by definition, the labels of the length field have to appear in the exit condition in every loop iteration). Once we have determined the set of input bytes that are accessed, we check whether they are consecutive. If this is the case, we assume that we have correctly identified a length field with its corresponding target field. If the bytes are not consecutive, the length field candidate is discarded.

Once a target field is identified, we can look for padding fields. Padding is used in some protocols to keep fields aligned to a word size (either two, four, or eight bytes). We detect a padding field if we find an appropriate number of unused bytes immediately preceding or following the end of the target field. A byte is unused if, throughout the execution trace, its taint label only occurs as operands of move instructions.

Additional information on length and target fields can be obtained by directly examining the parameters of system calls which read data from the network, such as the Unix `read` and `recv` system calls. If the number of bytes to be read is tainted with a set of labels, those labels clearly correspond to a length field, while the bytes being read are the target field.

3.3 Extracting additional information

Once the input message is decomposed into its constituent fields, we attempt to extract additional information that might provide insight into the semantics of certain fields. Currently, we derive four types of additional information: First, we attempt to detect the use of keywords that have a special meaning for the protocol. Second, we identify fields that are used as file names in file system accesses. Third, we locate input fields that are directly echoed in a program's network output (e.g., part of the response to the host that sent a request). This indicates that the field might be used as cookie or message identifier. Fourth, we identify pointer fields, which encode the absolute or relative offset of another field in the input message.

To identify keywords, we use two different techniques. First, we scan the execution trace for the occurrence of x86 string compare instructions (such as `comps`) that *successfully* compare a sequence of one or more tainted input bytes with a constant (untainted) string. The second check looks for a sequence of comparison instructions that successfully compare one or more bytes in a field with untainted characters. These comparisons have to be equality checks, all other comparisons are ignored. The rationale behind our keyword identification is that protocol keywords are typically hardcoded in some way by the application. When a certain sequence of characters is successfully compared with tainted input bytes, we mark this sequence as a keyword candidate. Note that our keyword detection leverages information derived by the delimiter analysis. This is necessary to exclude delimiter checks from the keyword analysis. Otherwise, all delimiter bytes would be considered as keywords, as they appear as operands in successful comparison operations.

Once a keyword candidate is found, we then attempt to verify it by scanning for this keyword in the server binary. That is, a keyword candidate is considered a keyword only when it is contained as a sequence of bytes in the application binary. Additionally, we require keywords to be at least three bytes long, to avoid false positives where a short string occurs in the program binary by chance. Of course, in general, a keyword does not necessarily have to appear directly in the binary. However, since a keyword is a string (or byte sequence) that is defined by the protocol, and thus, often encoded in the application, we consider this information a valuable confirmation. Moreover, we have so far found all keywords of the protocols that we analyzed embedded in the server binary.

The mechanism outlined above also allows us to implement a technique to extend keywords. More precisely, once we have found a keyword string (or byte sequence), we attempt to extend this string by considering the bytes that follow that keyword in the protocol message. As long as the extended string is still present in the program binary, we extend the keyword by those bytes. This is helpful to correctly identify keywords in cases where programs only compare a part of a keyword with the actual input data. For example, in the SMB Negotiate Protocol message, the

SMB server is only checking for the existence of the string “MICROSOFT NETWORKS ”, however, by employing our technique, we can extract the complete protocol keyword “MICROSOFT NETWORKS 1.03.”

File names are straightforward to recognize. Whenever a sequence of tainted bytes is used as the argument of a system call that opens or creates files, these bytes are assumed to represent a file name. Also, when a sequence of tainted bytes is found in the argument of a system call that sends out network traffic, and the values of these bytes remain unchanged, we consider them as being a field that is echoed back.

Identifying pointer fields is also simple. Whenever tainted bytes are accessed through a pointer tainted with a set of consecutive labels, those labels are marked as a pointer field. The lowest offset in the message of the bytes accessed through the pointer field is taken to be the offset that the pointer points to.

4 Analysis of multiple messages

When analyzing a single protocol message, our system breaks up the byte sequence that makes up this message into a number of fields. As mentioned previously, these fields can be nested, and thus, are stored in a hierarchical (tree) structure. The root node of the tree is the complete message. Both length field and delimiter analyses are used to identify parts of the message as scope fields, delimited fields, length fields, or target fields. Input bytes that cannot be attributed to any such field are treated as individual byte fields or, if they are in a delimiter scope and end at a delimiter, as arbitrary-length token fields. We refer to fields that contain other, embedded fields as *complex fields*. Fields that cannot be divided further are called *basic fields*. In the tree hierarchy, complex fields are internal nodes, while basic fields are leaf nodes.

It is possible, and common, that different message instances of the same type do not contain the same fields in the same order. For example, in a HTTP GET request, the client can send multiple header lines with different keywords. Moreover, these headers can appear in an almost arbitrary order. Another example is a DNS query where the requested domain name is split into a variable number of parts, depending on the number of dots in the name. By analyzing only a single message, there is no way for the system to determine whether a protocol requires the format to be exactly as seen, or whether there is some flexibility in the way fields can be arranged. To address this question, and to deliver a general and precise message format specification, information from multiple messages must be combined.

When combining two (or more) messages, two steps need to be carried out. In the first step, we have to find an alignment between the messages such that similar message structures are aligned. That is, we aim to locate those fields that do not change between messages of the same type. Then, in a second step, the messages have to be combined such that the result generalizes over all input messages. This makes it necessary to identify optional fields, or to find fields that can appear in one or more alternative variants.

Alignment. To find common structures between messages, we make use of a sequence alignment algorithm (the Needleman-Wunsch algorithm [20], which is heavily used in bio-informatics). The goal of this algorithm is to take two sequences as input and find those parts of the sequence that are similar, respecting the order of the elements. These similar parts are then aligned, exposing differences or missing elements in the sequences. An example is shown in Figure 3. Here, the alignment algorithm receives two strings as input and produces an alignment that identifies the elements ’a’ and ’c’ as similar. Also, it shows that there is a gap in the second sequence, because the first one has an additional element ’b’. Finally, the alignment shows that the strings end with different characters ’d’ and ’e’, indicating that the strings can have (at least) two alternative ends.

Sequence alignment algorithms have been used previously [9, 16] for automatic protocol reverse engineering. More precisely, these algorithms have been used on network traces to identify similar byte sequences in messages. However, in previous work, the systems operate directly on the input byte sequences. In our case, the alignment operates on fields that have been extracted by analyzing individual messages. Because we have a significant amount of structural information about individual messages, the alignment algorithm operates on elements of a higher level of abstraction, and thus, the results are more accurate.

To perform alignment, the Needleman-Wunsch algorithm uses a scoring function that defines how well two elements of a sequence match. For example, when applying the algorithm to strings, one can assign a negative score when the algorithm attempts to align two different characters, or when it attempts to align a character in one string with a gap in the other one. When the two characters are the same, a positive score is assigned. Based on the scores, the algorithm finds the alignment that maximizes the overall score.

For our system, the scoring function used for alignment has to take into account that a message is not simply a sequence of basic fields. Instead, a message can be composed of complex, nested fields. To obtain a score for a pair of complex fields (fields arranged in a tree hierarchy), we apply the Needleman-Wunsch algorithm in a recursive fashion. That is, we view the immediate children of the root node (i.e., the first layer of the tree) as a sequence of fields. This is done for both trees, yielding two sequences of child nodes. We then apply the alignment algorithm to these sequences. The score that is calculated is taken as the score for the two root nodes. Of course, because each child node can itself be a complex field, the alignment might be called recursively again.

When comparing two complex fields, the alignment algorithm is called recursively. In order for the recursive calls to terminate, we require a function that can resolve the base cases. More precisely, we require a way to calcu-

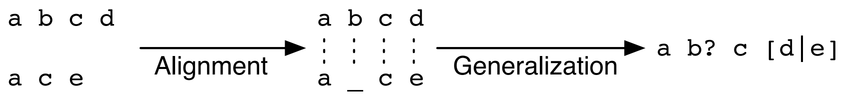


Figure 3. String alignment based on the Needleman-Wunsch algorithm.

late a score for a pair of basic fields, and a basic field that is matched with a complex field.

To calculate the score for a pair of basic fields, we use the following, simple method: We return a value of +1 if two basic fields match, and a value of -1 if they do not match. Also, the penalty for a single gap is set to -1. Two length fields match if they have the same number of bytes. A target field matches another target field only if the corresponding length fields match. A scope field and a delimited field match if they use the same delimiter. A token field always matches another token field. An individual byte field always matches another individual byte field, even if the value of the byte is different. This highlights how our algorithm does not rely on textual similarity between messages, but on the message structure as inferred from server behavior.

Clearly, fields of different types never match. Therefore, for the alignment between a basic and a complex field, we always report a mismatch. However, the score cannot be simply set to -1. This is because a complex field can contain many embedded fields, and thus, the penalty has to be increased proportionally to the number of elements that the complex field contains. We solve this by simply multiplying the penalty score by the number of embedded elements.

It would be possible to further tune the values used for the scoring function. In fact, the authors of previous work [10], who used alignment algorithms on network traces, found it very challenging to select appropriate weights. However, because we operate on input that is well-structured and on a high level of abstraction, tuning was not necessary as our alignment approach immediately returned good results.

Generalization. After the alignment step, it is necessary to produce a generalized result that can serve as an abstraction for the inputs. As an example for the generalization step, revisit the example shown in Figure 3. Note that the character 'b' appears as optional in the generalized regular expression, while there is an alternative expression inserted for the characters 'd' and 'e'.

For generalization, simple rules apply. When a node in the tree is aligned with a gap, then this node becomes an optional node. When two nodes are aligned that do not match, we introduce an alternative node. This alternative node simply indicates that either one of the two structures can appear in the message. When matching, but non-identical nodes are aligned, a new node is created that preserves common properties of the aligned nodes. Two non-identical nodes can be aligned when those nodes have the same type, but their content is different. For instance, when two byte fields are aligned that have different content, the resulting node is a generic byte node that represents arbitrary bytes.

Once we have created a "generalized" tree, this tree can be traversed to produce a regular expression. This regular expression then represents the generalized message format specification. For a more complex example of running the alignment and generalization algorithm on two messages with a hierarchy of fields, please refer to Appendix C.

Repetition detection. A common pattern is for a protocol message format to allow an arbitrary number of repetitions of a certain part of the message. We use a simple but effective heuristic to detect such cases. At the end of the generalization phase, we look for two (or more) consecutive, optional nodes that match. When we find such nodes, we merge them into a single repetition node. For example, using the standard regular expression notation, the sequence '`a?a?a?`' would become '`a*`'. If there is a non-optional field of the same type before or after the newly created repetition node, it is merged into a "one or more" repetition. For example, the sequence '`aa?a?`' would become '`a+`'. The limitation of our technique is that it only detects repetitions of identical nodes, missing more subtle cases in which repeating elements contain a number of different fields, such as in the case of HTTP header lines.

Parsing. We developed a simple parser that leverages the generalized regular expressions that are produced by the previous step to demonstrate that it is possible to parse additional protocol messages of the same type. Since it is a non-deterministic parser, the worst case computational complexity of parsing is exponential. In practice, this has not been a problem: the parser runs on our entire data set in under two minutes. For parsing messages, a regular expression is almost sufficient. However, some additional information is necessary. In particular, to parse a target field, we need to compute its length from the value of the corresponding length field. Otherwise, we do not know when to stop reading bytes from a target field. For this, we need to make some assumptions on the encoding of the length field. In our current system, we assume that a length field stores an integer value with either little-endian or big-endian encoding. With no further loss of generality, we can assume that $T = L * \text{scale} + \text{offset}$ under one of the two encodings, where T and L are, respectively, the length of the target field and the value of the length field, and scale

Test Case	Server	Protocol	Message	#Msg
apache	apache	HTTP	GET	34
lighttpd	lighttpd	HTTP	GET	34
ircnick	iacd	IRC	NICK command	5
ircuser	iacd	IRC	USER command	5
smtphelo	sendmail	SMTP	HELO command	8
smtpquit	sendmail	SMTP	QUIT command	8
smtpmail	sendmail	SMTP	MAIL command	8
dnsquery	named	DNS	Query IPv4 Address	9
nfslookup	nfsd	RPC/NFS	Lookup	12
nfsgetattr	nfsd	RPC/NFS	Getattr	9
nfscreate	nfsd	RPC/NFS	Create	12
nfswrite	nfsd	RPC/NFS	Write	16
smbnegotiate	samba	SMB	negotiate protocol request	8
smbtree	samba	SMB	tree connect andX request	9
smbsession	samba	SMB	session setup andX request	8

Table 1. Test case summary.

and offset are two integer constants. When merging two length fields a and b, we can compute

$$\text{scale} = (T_a - T_b)/(L_a - L_b)$$

$$\text{offset} = T_a - L_a * \text{scale} = T_b - L_b * \text{scale}$$

This computation will only be successful for one of the two encodings (returning an integer result for scale, and a consistent value for offset). This allows us to individually detect the endianess of each length field, without assuming a common encoding of fields in the protocol.

In a similar way, to be able to parse a pointer field, we need to compute the position in the packet that it points to. In addition to the two possible encodings, a pointer field can represent an absolute position in the message or an offset relative to the pointer field itself. As for length fields, we compute a scale and an offset and discard encoding options for which this computation fails. When parsing a pointer field, we check that the offset this field points to is a valid offset inside the message.

5 Evaluation

In this section, we present the experimental evaluation of our approach. We applied our techniques to multiple server implementations of six real-world protocols. In particular, we selected HTTP, IRC, DNS, NFS, SMB and SMTP. These protocols were chosen because they have been used in the evaluation of previous work [4, 9, 10], because they represent a good mix of text-based (HTTP, IRC, SMTP), binary (DNS, NFS) and mixed (SMB) protocols, and because they are implemented by complex, real-world programs (such as `apache`, `bind`, or `samba`). Note that all programs that we analyzed are x86 Linux binaries. This is because our dynamic tainting tool currently onl runs on this platform. However, our general approach equally applies to other systems (e.g., such as Windows) as well.

For each of the six analyzed network protocols, we selected one well-known server program that implements this protocol (two in the case of HTTP), and one or more message types (e.g., a GET request for HTTP, Lookup, Getattr, Write and Create requests for NFS). The full list of test cases is detailed in Table 1. This table shows the server, protocol, and message type that we analyzed. For each test case, we used client programs to send the server a number of messages of the selected type. The number of client messages for each test case is shown as column *Message* in the table.

The server programs were monitored while processing the requests, and we generated an execution trace for each message. In the next step, our system analyzed these execution traces and generated appropriate message format specifications. In Appendix A, we present the full specifications obtained for one message format for each of the tested protocols. Table 2 shows the field detection results for different types of fields in each of the test cases, while Table 3 further details the keyword and delimiter detection results for each server. The results in these tables were obtained by manually comparing our specifications with official RFC documents and with Wireshark [30] output. More details about the results for each individual protocol are discussed in the following paragraphs. In general, however, we can observe that most of the fields were correctly identified.

Finally, the specifications obtained for each test case were used with our simple protocol parser to parse another set of messages (of the same type). Despite imperfections in the inferred formats (as highlighted by Table 2), parsing succeeded in all test cases. This demonstrates that our system is capable of automatically deriving accurate format

Test Case	Length	Target	Padding	Pointer	Delimiter	Keyword	File	Repetition	Total
apache	0	0	0	0	4/5	6/6	1/1	1/2	12/14 (86%)
lighttpd	0	0	0	0	4/5	7/7	1/1	1/2	13/15 (87%)
ircnick	0	0	0	0	1/1	1/1	0	0	2/2 (100%)
ircuser	0	0	0	0	2/2	1/1	0	0	3/3 (100%)
smtphelo	0	0	0	0	1/2	1/1	0	0	2/3 (67%)
smtpquit	0	0	0	0	1/1	1/1	0	0	2/2 (100%)
smtpmail	0	0	0	0	3/5	3/3	0	0	6/8 (75%)
dnsquery	1/1	1/1	0	0	0	0	0	1/1	3/3 (100%)
nfslookup	4/5	4/4	2/2	0	0	0	1/1	0	11/11 (92%)
nfsgetattr	3/4	3/3	1/1	0	0	0	0	0	7/8 (88%)
nfscreate	4/5	4/4	2/2	0	0	0	0	0	10/11 (91%)
nfswrite	4/6	4/4	2/2	0	0	0	0	0	10/12 (83%)
smbnegotiate	2/2	2/2	1/1	0	1/1	10/10	0	0/1	16/17 (94%)
smbtree	2/3	2/2	0	1/1	2/2	3/3	0	0	10/11 (91%)
smbsession	8/9	8/8	0	7/7	2/2	2/2	0	0	27/28 (96%)

Table 2. Field detection results: correctly identified fields / total fields in message format.

Server		Detected	Missed	False positives	Unsupported
apache	keywords	”GET”, ”HTTP/1.1”, ”Host”, ”Connection”, ”close”, ”keep-alive”		”Accept-” ”Accept”	”Accept-Language”, ”Accept-Encoding”, ”Accept-Charset”, ”User-Agent”, ”Keep-Alive”
	delimiters	CRLF, SPACE, ”/”, ”.”	”:”		
lighttpd	keywords	”GET”, ”HTTP/”, ”Host”, ”User-Agent”, ”Connection”, ”close”, ”keep-alive”		”Accept”	”Accept-Language”, ”Accept-Encoding”, ”Accept-Charset”, ”Keep-Alive”
	delimiters	CRLF, SPACE, ”/”, ”.”	”:”		
iacd	keywords	”NICK”, ”USER”			
	delimiters	CRLF, SPACE			
sendmail	keywords	”HELO”, ”QUIT”, ”MAIL”, ”FROM”, ”<”			
	delimiters	CRLF, ”.”, ”:”, ”>”	CRLF ”@”		
samba	keywords	”MICROSOFT NETWORKS 1.03”, ”MICROSOFT NETWORKS 3.0”, ”LANMAN1.0”, ”DOS LANMAN2.1”, ”LM1.2X002”, ”NT LANMAN 1.0”, ”NT LM 0.12”, ”Samba”, ”0xffSMB”, ”IPC”, ”?????” ”NTLMSSP”			”PC NETWORK PROGRAM 1.0”
	delimiters	”0x00”, ”0x0000”			

Table 3. Keyword and delimiter detection results. “Unsupported” keywords are part of the protocol specification but are not supported by the server in the tested configuration.

specifications that can be directly used to parse messages of a certain type. Also, the specifications contain detailed information about the format of the message and the way in which the sequence of bytes at the network level are split into meaningful, application-level fields.

Hypertext Transfer Protocol (HTTP). For our analysis of HTTP, we selected the latest stable versions of `apache` and the `Lighttpd` web server (versions 1.3.39 and 1.4.18, respectively). Then, we used clients such as the Firefox web browser to generate HTTP GET messages that request different pages stored at the server. These messages contained the "Host" header (which is mandatory in HTTP/1.1), as well as a number of additional, optional header lines.

Results for both servers are presented in Table 2 and Table 3. The complete grammar that was automatically derived from analyzing the way in which `apache` processed the GET messages is presented in Appendix A. When examining this grammar in more detail, one can see that our system has extracted a quite accurate specification for HTTP GET requests. The message has been split into lines that are separated by the multi-byte delimiter '`\r\n`', and the first line ("GETLINE") is split into three fields by the space character. The system has identified keywords such as the name of the "GET" operation and of header fields such as "Host" and "Connection." Moreover, thanks to repetition detection, the name of the requested resource ("FILENAME") was recognized as a file name with an arbitrary number of directories, followed by an optional extension. The system also automatically recognized that "FILENAME" is indeed specifying a resource in the file system. Also, the IP address following the "Host" header was split into four character tokens separated by dots (representing the general format of an IP address), and our system determined that the values of the connection header can take one of the two alternatives "keep-alive" or "close". Finally, one can see that the specification correctly captures the fact that the first two lines (the request line and the "Host" header) have to be present in a request, while the additional header lines are all optional. Note that the HTTP RFC does not specify an optional file extension in the URL that is delimited by a dot characters ('.'). However, this is not an error of analysis but reflects the way in which the server actually processes the input. In fact, `apache` uses file extensions to decide which extension module to use to process a request. Interestingly, for `Lighttpd`, the dot character the file name is not used (and thus, not recognized) as delimiter.

Of course, our HTTP grammar is not flawless, exposing some limitations of our system. For example, the tokens "Accept" and "Accept-" are incorrectly identified as keywords[1]. This is because `apache` scans for the supported keyword "Accept-Ranges." As this keyword is not present in our HTTP traffic, parsing always fails after processing the string "Accept-." Nevertheless, our analysis considers the sequence of successful comparison operations as indication that the keyword "Accept-" is part of the protocol message specification.

Another inaccuracy is the fact that our specification accepts header lines with arbitrary content. This can be seen on the right side of the grammar production for "UNUSEDHDR", which includes `(TEXT)+`. The reason for this is that `apache` ignores the "User-Agent" and "Keep-Alive" headers in its default configuration. Thus, our system cannot infer any constraints on the value of these headers, and has to assume that they can hold arbitrary content. This shows that it is difficult for our approach to derive information for parts of a message that are not processed by the server. Similarly, several headers starting with "Accept" are not further parsed by `apache`, so the format inferred for those headers is overly permissive. In fact, we tested `apache` by sending garbage header lines (with no ':'), and they were indeed accepted by the server, as long as they did not start with a supported keyword.

The colon character (':') is not recognized as a delimiter because the server is only checking for it directly after it has identified a keyword. As a result, the server only checks a single input character for the occurrence of the delimiter. This is not sufficient for our delimiter analysis, which requires at least two consecutive input bytes to be analyzed before considering a character as delimiter (see Section 3.1). Finally, our system is not capable of generalizing the request header lines as repeating elements. This is because our repetition detection technique cannot identify repetitions of different (non-matching) nodes (as explained in Section 4).

The grammar extracted from analyzing the 34 GET requests was used as input to our parser. This parser was then applied to an additional set of 20 different GET requests. The additional messages could be parsed correctly, demonstrating that our automatically derived specification is sufficiently accurate to generate parsing code for this type of messages.

Internet Relay Chat (IRC). To analyze IRC, we chose the fully-featured `iacd` IRC server. As for HTTP, we recorded a set of messages that clients send to the server when establishing a connection. Results for the NICK and USER commands are shown in Tables 2 and 3, and the grammar for the USER command is in Appendix A. When examining the specifications produced for these two types of messages, it can be seen that the message format is significantly simpler than the one for HTTP. Our system has produced correct format descriptions and recognized the relevant keywords (such as "NICK" and "USER"). When using these two specifications to parse an additional set of IRC messages, we observed no problems.

Simple Mail Transfer Protocol (SMTP). We analyzed SMTP by examining the well-known `sendmail` SMTP server (version 8.13.8-3). For our experiments, we

[1]"Accept" is in fact a valid HTTP header name, but it is not supported in the tested configuration.

recorded a set of eight SMTP sessions. Our tool produced grammars for the HELO, QUIT, and MAIL commands (shown in Appendix A) that appear in each of these sessions.

The QUIT command was identified correctly. For the HELO command, the space delimiter was detected as a constant part of the token that follows the keyword. The reason is that the server only checks for the space in the position after the HELO keyword, but our delimiter detection requires at least two consecutive checks to recognize a delimiter character.

While our system can correctly identify most of the keywords and delimited fields of the MAIL command, it misses the '@' delimiter in the email address. The reason is that the parser is not explicitly scanning the input data for the '@' character, but for any character in a non-printable range. Also, the '\r\n' sequence at the end of the message is not classified as a delimiter, because, again, the parser is only checking for this sequence once (after the closing angle bracket). However, the angle brackets and dot delimited field are identified correctly, as well as the path repetition in the email address suffix.

Domain Name Service (DNS). For the DNS protocol, we examined the latest `named` daemon from the `bind` distribution (version 9.4.1). We decided to extract the specification for the messages that resolve a DNS name to the corresponding IP address. To this end, we issued a set of nine DNS queries, using the Linux `host` command with different names and flags. The complete specification that we extracted can be found in Appendix A.

When examining this specification, one can observe that our system correctly recognized length fields and the corresponding target fields. The protocol splits the DNS name into the parts (or labels) that are separated by dots. That is, the name `www.example.com` is split into the three labels `www`, `example`, and `com`. Each of these labels is stored in a variable length field, with its length directly stored before. This is reflected in the grammar as a repetition of one or more "NAMEPARTS". Our system also identified that the first two bytes of the message (which represent the message ID) are echoed back to the client in the server response.

Because all queries in our message set contain only a single DNS query (and no additional resource records), the corresponding bytes of the message are identified to have a fixed content. Thus, using our specification, we can parse any DNS query for an IP address. We verified this by using our specification for parsing an additional set of messages created by performing name lookups, and found that the parsing was successful.

Network File System (NFS). We selected NFS because it is a complex, binary protocol that is layered over an other protocol (RPC). For our experiments, we used the NFSv2 daemon (with RPCv2) that ships as the default of the current Ubuntu Linux distribution (version 2.2.beta47). To generate messages, we performed several operations on an NFS-mounted directory, such as obtaining a directory listing as well as writing to and reading from a file. Results for four types of NFS messages are listed in Table 2. The specification extracted for NFS Lookup messages is shown in Appendix A. This is the message used by NFS to look up a file name and obtain a corresponding file handle.

When examining the resulting specification, one can see that our system correctly recognized a number of complex structures. This includes the file name, the machine name, and the set of additional group identifiers that are all transmitted as variable length fields. We further detect that the file name is used by the server to access file information, and we detect that the four byte request ID (XID) is echoed back to the client. The example of NFS also shows that we detect variable length fields that are nested. The field "CREDENTIALS_BODY" is a variable field whose length is stored in the preceding field "CREDENTIALS_LENGTH." Furthermore, the "CREDENTIALS_BODY" field contains a number of additional fields (such as machine name or the list of auxiliary group identifiers) that are themselves of variable length. Finally, note that our system can also deal with variable length padding. We identify that the end of the variable length field "MACHINE_NAME" is padded so that the next field is word aligned.

Again, our specification is limited by the input that we observe. For example, since we always use the same type of authentication mechanism, the field "VERIFIER_LENGTH" is recognized as a constant 0, and we do not detect it as a length field. The reason is that since the length is always zero, our length field detection algorithm has no loop iterations to work on. Also, the values in the RPC part of the protocol are all fixed and represent a NFS message (which is reasonable in this case, but provides very limited information about the general format of RPC). Nevertheless, using this grammar, we could parse a set of additional messages, created from looking up different files, without problems.

In the NFS Getattr and NFS Create test cases, our tool similarly misses the "VERIFIER_LENGTH" length field. In the NFS Write test case, another length field is missed by our system, which is the "total count" field. In this case, the length field is redundant because it always holds the same value as the "data length" field. Therefore, the server does not actually use it in a loop to parse the target field (it uses "data length" instead), and later just verifies that the two values are equal. Similarly, our parser could successfully use the specification produced to parse other NFS write messages, despite this missing field.

Server Message Block protocol (SMB). For our experiments with SMB, we selected a current version of the Samba server (version 3.0.26a). To generate messages, we used the smbclient program (which is part of the Samba distribution) to connect to shared folders and then performed some file operations. Results for three types of SMB messages are listed in Tables 2 and 3.

The specification extracted for the SMB tree connect andX request messages is shown in Appendix A. This is the message used by SMB to connect to a share once a session has been established. It is quite interesting because it contains almost all of the types of fields our tool is capable of detecting: length, target, and pointer fields, which are typically found only in binary protocols, but also delimiter and keyword fields which are typically found in text-based protocols. Both "0x00" and "0x0000" are detected as delimiters, used for ASCII and Unicode strings, respectively. The "SERVICE" is an alternative of the two keywords "IPC" and "?????", which are the only types of services we connected to in our tests. It is also interesting to note that several parts of the message are unused, meaning that the corresponding taint labels never appear as operands of anything but move instructions. This includes the "PROCESS_ID" and "MULTIPLEX_ID" fields that are echoed back to the client without using their values.

The main imperfection of the specification is that we did not recognize "BYTE_COUNT" as length field, which holds the length of the target fields "PASSWORD", "PATH", and "SERVICE". As for the NFS Write test case, the server does not use this field in a loop to parse the target field. Again, it is redundant because the length of those three fields can be determined from the "PASSWORD_LENGTH" field, the "0x0000" delimiter, and the "0x00" delimiter. Tree connect messages can therefore be parsed using the specification produced by our tool.

Results for the SMB Tree test case have the exact same issue, missing the "BYTE_COUNT" field. In the SMB Negotiate test case, the only imperfection is that our tool fails to detect the sequence of "dialects" as a repetition. As for HTTP headers, this is because each dialect node contains a different keyword, and our repetition detection does not detect repetitions of non-matching nodes.

6 Related work

Protocol reverse engineering. So far, protocol reverse engineering has mostly been a tedious and labor-intensive manual task. Such tasks have often been undertaken to analyze popular, but proprietary protocols with the goal to provide free, open source alternatives. For example, Samba [27] is a project that offers a free re-implementation of the SMB/CIFS networking protocol (a closed protocol used by Microsoft for their network file system). Other reverse engineering efforts are targeting popular instant messenger protocols (such as the OSCAR protocol, used by ICQ and AIM [12]) or file sharing protocols (such as FastTrack [13]).

One of the first attempts to perform automated protocol analysis was aimed at application session replay. In the context of honeypots, researchers require modules that can correctly respond to connection attempts initiated by malicious code (e.g., worms) and maintain an application dialog long enough until the malware sends its payload. To generate these modules automatically, systems such as RolePlayer [10] and ScriptGen [16, 17] analyze network traffic and attempt to generalize the traces so that correct replies can be generated to new requests. These systems only focus on the protocol format to the extent necessary for replay, in particular, on the recognition of fields that contain cookie values or IP addresses. In [21], the authors propose to use static as well as dynamic binary analysis (but not dynamic taint analysis) and program verification techniques to provide a sound solution to the replay problem. That is, the system can guarantee to answer correctly to a request, provided that the analysis terminates. Unfortunately, the scalability is limited, and the system was not tested on any real-world program.

In addition to techniques for replaying application dialogs, systems were proposed that attempt to discover the complete protocol format. One such system was developed by the "Protocol Informatics Project" [1]. In their work, the authors propose to apply bio-informatics techniques (such as sequencing algorithms) to network traffic. The goal is to identify protocol structure and fields with different semantics. A related, but improved technique extends the bytewise alignment with recursive clustering and type-based clustering [9]. While these systems show some success in extracting protocol formats, the precision of the analysis is often limited and heavily relies on (protocol-specific) heuristics. Because of the lack of information present in network traces, messages of the same type are sometimes considered as being different, and data artifacts are recognized as protocol keywords.

Independently and concurrently to our work, the authors of Polyglot [4] and AutoFormat [18] propose to extract protocol information by observing the execution of a program while it processes execution traces to detect the fields which compose a message. Polyglot can, like our system, infer some field semantics, such as detecting keyword fields and "direction" fields (which can be either length or pointer fields). AutoFormat, on the other hand, leverages execution contexts (i.e., call stack information) to infer hierarchical relationships between fields, but it does not extract any field semantics. Both of these systems analyze only *single* messages and split them into fields; we go a step further and consider *multiple* messages of the same type and automatically produce a grammar which can be used to parse other messages of that type. Analyzing multiple messages allows us to extract additional semantics such as identifying alternative and optional fields, and sets of keywords. Finally, we employ static binary analysis to improve the quality of length field and keyword field detection.

Dynamic taint analysis. The idea of dynamically monitoring the execution of a program, and tracking the ways in which it processes and propagates information using taint analysis, has been applied to a number of security problems. For example, in [5], the authors use taint information to track the lifetime of data. The goal is to determine the use of sensitive information by the operating system and large applications. Other researchers used taint analysis to monitor program execution for the use of tainted data as arguments to control flow instructions or systems calls [7, 8, 22, 26]. The aim of these systems is to identify memory corruption exploits at run-time, and, in some cases, to create signatures for detected attacks. Finally, taint analysis has also been used to detect malicious software based on their characteristic information flow behavior. In [11], spyware is detected by finding programs that leak sensitive user information, while in [31], a system is presented that can identify a number of malware classes by observing suspicious information access and propagation patterns.

7 Conclusion

Protocol reverse engineering is the process of extracting application-level protocol specifications. With respect to security, having a detailed knowledge of protocol specifications is important for a number of tasks (e.g., intrusion detection, protocol fuzzing, service discovery). Unfortunately, current approaches of determining a specification typically involve a significant amount of manual analysis or yield results with limited accuracy. To address these problems, we introduced a novel approach to automatic protocol reverse engineering. This approach works by dynamically monitoring the execution of an application and analyzing how protocol messages are processed. To this end, we first extract information about the fields of individual messages and then derive more general specifications by aggregating the information collected for multiple messages. Our experiments with real-world protocols and server applications demonstrate that we are able to extract the format specification for different types of messages. Using these specifications, we then automatically generate appropriate parser code. We believe that the techniques that we introduce in this paper will be useful for security practitioners and researchers who need to deal with closed, proprietary protocols.

Acknowledgments

This work has been supported by the Austrian Science Foundation (FWF) and by Secure Business Austria (SBA) under grants P-18764, P-18157, and P-18368. Paolo Milani's work was funded by Telecom Italia S.p.A. under a Ph.D. program.

References

[1] M. Beddoe. The Protocol Informatics Project. In *Toorcon*, 2004.

[2] N. Borisov, D. Brumley, H. J. Wang, J. Dunagan, P. Joshi, and C. Guo. A generic application-level protocol analyzer and its language. In *14h Symposium on Network and Distributed System Security (NDSS)*, 2007.

[3] D. Brumley, J. Caballero, Z. Liang, J. Newsome, and D. Song. Towards Automatic Discovery of Deviations in Binary Implementations with Applications to Error Detection and Fingerprint Generation. In *Usenix Security Symposium*, 2007.

[4] J. Caballero and D. Song. Polyglot: Automatic Extraction of Protocol Format using Dynamic Binary Analysis. In *ACM Conference on Computer and Communications Security (CCS)*, 2007.

[5] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding Data Lifetime via Whole System Simulation. In *Usenix Security Symposium*, 2004.

[6] C. Cifuentes and M. V. Emmerik. Recovery of Jump Table Case Statements from Binary Code. Technical Report Technical Report 444, The University of Queensland, 1998.

[7] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-End Containment of Internet Worms. In *20th ACM Symposium on Operating Systems Principles (SOSP)*, 2005.

[8] J. Crandall and F. Chong. Minos: Control Data Attack Prevention Orthogonal to Memory Model. In *37th International Symposium on Microarchitecture (MICRO)*, 2004.

[9] W. Cui, J. Kannan, and H. Wang. Discoverer: Automatic Protocol Reverse Engineering from Network Traces. In *16th Usenix Security Symposium*, 2007.

[10] W. Cui, V. Paxson, N. Weaver, and R. Katz. Protocol-Independent Adaptive Replay of Application Dialog. In *13th Symposium on Network and Distributed System Security (NDSS)*, 2006.

[11] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song. Dynamic Spyware Analysis. In *Usenix Annual Technical Conference*, 2007.

[12] A. Fritzler. UnOfficial AIM/OSCAR Protocol Specification. `http://www.oilcan.org/oscar/`, 2007.

[13] Open Source FastTrack P2P Protocol. `http://gift-fasttrack.berlios.de/`, 2007.

[14] P. Haffner, S. Sen, O. Spatscheck, and D. Wang. ACAS: Automated Construction of Application Signatures. In *ACM Workshop on Mining Network Data*, 2005.

[15] C. Kruegel, F. Valeur, W. Robertson, and G. Vigna. Static Analysis of Obfuscated Binaries. In *Usenix Security Symposium*, 2004.

[16] C. Leita, M. Dacier, and F. Massicotte. Automatic Handling of Protocol Dependencies and Reaction to 0-Day Attacks with ScriptGen-based Honeypots. In *Symposium on Recent Advances in Intrusion Detection (RAID)*, 2006.

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

[18] Z. Lin, X. Jiang, D. Xu, and X. Zhang. Automatic Protocol Format Reverse Engineering through Conectect-Aware Monitored Execution. In *15th Symposium on Network and Distributed System Security (NDSS)*, 2008.

[19] J. Ma, K. Levchenko, C. Kreibich, S. Savage, and G. Voelker. Unexpected Means of Protocol Inference. In *Internet Measurement Conference (IMC)*, 2006.

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

[21] J. Newsome, D. Brumley, J. Franklin, and D. Song. Replayer: Automatic Protocol Replay by Binary Analysis. In *13th ACM Conference on Computer and Communications Security (CCS)*, 2006.

[22] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Network and Distributed System Security Symposium (NDSS)*, 2005.

[23] P. Oehlert. Violating Assumptions with Fuzzing. *IEEE Security and Privacy*, 3(2), 2005.

[24] R. Pang, V. Paxson, R. Sommer, and L. Peterson. binpac: a yacc for writing application protocol parsers. In *Internet Measurement Conference (IMC)*, 2006.

[25] V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. In *Usenix Security Symposium*, 1998.

[26] G. Portokalidis, A. Slowinska, and H. Bos. Argos: an Emulator for Fingerprinting Zero-Day Attacks. In *ACM SIGOPS EUROSYS*, 2006.

[27] How Samba Was Written. `http://samba.org/ftp/tridge/misc/french_cafe.txt`, 2007.

[28] V. Sreedhar, G. Gao, and Y. Lee. Identifying loops using DJ graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(6), 1996.

[29] S. Venkataraman, J. Caballero, P. Poosankam, M. Kang, and D. Song. Fig: Automatic Fingerprint Generation. In *Symposium on Network and Distributed System Security (NDSS)*, 2007.

[30] Wireshark: The World's Most Popular Network Protocol Analyser. `http://www.wireshark.org`.

[31] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In *14th ACM Conference on Computer and Communications Security (CCS)*, 2007.

Appendix A: Message format specifications

In the following subsections, we present the complete format specifications that our system extracted from different messages by analyzing four real-world protocols. To improve the readability of the results, we assigned meaningful names to the symbols based on their protocol semantics. For example, for HTTP, our system would denote the "GETLINE" non-terminal as "scope field delimited by space". However, all characters and strings that are enclosed in quotes (such as the HTTP keyword "GET" or the delimiter '/') are directly extracted by our system. Also, when using these grammars to generate parsing code, no modifications to the output of our system were necessary. Finally, the grammar contains annotations that show the semantics of certain fields (e.g., indication that a field is a file name, a field that is being echoed back to the host, etc.). These annotations are produced automatically by our system and are shown in angle brackets.

HTTP

```
REQUEST = GETLINE
          HOSTHDR
          [UNUSEDHDR]
          [ACCEPTHDR]
          (XACCEPTHDR)*
          [UNUSEDHDR]
          [CONHDR]
          CRLF
```

```
GETLINE = "GET"
          " "
          FILENAME
          " "
          "HTTP/1.1"
          CRLF

FILENAME    = (PATHELEM)+ [SUFFIXELEM]
<file name>

PATHELEM    = "/" (TEXT)+
<file name>

SUFFIXELEM  = "." (TEXT)+
<file name>

HOSTHDR     = "Host" ": " IP_ADDR CRLF

IP_ADDR     = (TEXT)+
              "."
              (TEXT)+
              "."
              (TEXT)+
              "."
              (TEXT)+

UNUSEDHDR   = (TEXT)+ CRLF

CONHDR      = "Connection"
              ": "
              "keep-alive" | "close"
              CRLF

ACCEPTHDR   = "Accept" (TEXT)+ CRLF

XACCEPTHDR =  "Accept-" (TEXT)+ CRLF

CRLF = "\r\n"
TEXT = printable ASCII character (no delimiter)
```

IRC

```
USERMSG  = "USER"
           " "
           (TEXT)+
           " "
           (TEXT)+
           " "
           (TEXT)+
           " "
           (TEXT)+
           CRLF

TEXT = printable ASCII character
CRLF = "\r\n"
```

SMTP MAIL Command

```
MESSAGE  = "MAIL "
           FROMLINE
           CRLF

FROMLINE = "FROM:"
           EMAIL

EMAIL    = "<"
           (TEXT)+
           (PATH)*
           ">"
```

```
PATH     = "."
           (TEXT)+

CRLF = "\r\n"
TEXT = printable ASCII character
```

DNS

```
QUERY = TRANSACTION_ID
        FLAGS
        QUESTIONS
        ANSWER_RRS
        AUTHORITY_RRS
        ADDITIONAL_RRS
        NAME
        TYPE
        CLASS

TRANSACTION_ID = BYTE{2}
<echoed>

FLAGS          = BYTE  "0x00"

QUESTIONS      = "0x0100"
ANSWER_RRS     = "0x0000"
AUTHORITY_RRS  = "0x0000"
ADDITIONAL_RRS = "0x0000"

NAME           = NAMEPART (NAMEPART)+  "0x00"

NAMEPART       = LENGTH  BODY

LENGTH         = BYTE
<target: BODY>

BODY           = TEXT{N}
<N = LENGTH>

TYPE           = "0x0001"
<echoed>

CLASS          = "0x0001"
<echoed>

TEXT = printable ASCII character
BYTE = any byte
```

RPC-NFS

```
LOOKUP_CALL        = XID
                     MESSAGE_TYPE
                     RPC_VERSION
                     PROGRAM
                     PROGRAM_VERSION
                     CREDENTIALS
                     VERIFIER
                     NFS_PAYLOAD

XID                = BYTE{4}
<echoed>

MESSAGE_TYPE       = CALL
RPC_VERSION        = "0x00000002"
PROGRAM            = NFS
PROGRAM_VERSION    = "0x00000002"
PROCEDURE          = LOOKUP

CREDENTIALS        = FLAVOUR
                     CREDENTIALS_LENGTH
                     CREDENTIALS_BODY

VERIFIER  = VERIFIER_FLAVOUR  VERIFIER_LENGTH

NFS_PAYLOAD        = DIRECTORY_HANDLE
                     FILE_NAME_LENGTH
                     FILE_NAME
                     [PADDING]

CREDENTIALS_LENGTH       = BYTE{4}
<target: CREDENTIALS_BODY>
<littleendian>

CREDENTIALS_BODY = STAMP
                   MACHINE_NAME_LENGTH
                   MACHINE_NAME
                   [PADDING]
                   UID
                   GID
                   AUXILIARY_GIDS

STAMP   = "0X0000" BYTE{2}

MACHINE_NAME_LENGTH    = BYTE{4}
<target: MACHINE_NAME>
<littleendian>

MACHINE_NAME           = TEXT{N}
<N = MACHINE_NAME_LENGTH>
<unused>

UID   = "0X0000"  BYTE{2}

GID   = "0X0000"  BYTE{2}

AUXILIARY_GIDS = AG_LENGTH  AG_BODY

AG_LENGTH        = BYTE{4}
<target: AG_BODY>
<littleendian>

AG_BODY          = BYTE{N}
<N = AG_LENGTH * 4>

DIRECTORY_HANDLE   = BYTE{8}  "0x00"{23}

FILE_NAME_LENGHT   = BYTE{4}
<target: FILE_NAME>

FILE_NAME          = BYTE{N}
<N = FILE_NAME_LENGTH>
<file name>

PADDING = BYTE{1,3}
<pad to 4 bytes>

CALL = "0x00000000"
NFS= "0x000186a3"
LOOKUP = "0x00000004"
FLAVOUR = AUTH_UNIX
AUTH_UNIX = "0x00000001"
VERIFIER_FLAVOUR = "0x00000000"
VERIFIER_LENGTH = "0x00000000"

TEXT = printable ASCII character
BYTE = any byte
```

SMB Tree connect AndX Request

```
MESSAGE = NETBIOS_HDR SMB

NETBIOS_HDR = MESSAGE_TYPE LENGTH

MESSAGE_TYPE = SESSION_MESSAGE
SESSION_MESSAGE = "0x00"

LENGTH          = BYTE{3}
<target: SMB>

SMB = SMB_HDR TREE_CONNECT

SMB_HDR = SERVER_COMPONENT
          SMB_COMMAND
          NT_STATUS
          FLAGS
          FLAGS2
          PROCESS_ID_HIGH
          SIGNATURE
          RESERVED
          TREE_ID
          PROCESS_ID
          USER_ID
          MULTIPLEX_ID

SERVER_COMPONENT = "0xff" "SMB"
SMB_COMMAND = TREE_CONNECT_ANDX
TREE_CONNECT_ANDX = "0x75"
NT_STATUS = STATUS_SUCCESS

STATUS_SUCCESS = BYTE{4}
<unused>

FLAGS = "0x08"
FLAGS2 = "0x01c8"

PROCESS_ID_HIGH = BYTE{2}
<unused>

SIGNATURE = BYTE{8}
<unused>

RESERVED = BYTE{2}
<unused>

TREE_ID = BYTE{2}
<unused>

PROCESS_ID = BYTE{2}
<echoed>
<unused>

USER_ID = BYTE "0x00"
<echoed>

MULTIPLEX_ID = BYTE "0x00"
<echoed>
<unused>

TREE_CONNECT = WORD_COUNT
               ANDX_COMMAND
               RESERVED2
               ANDX_OFFSET
               FLAGS3
               PASSWORD_LENGTH
               BYTE_COUNT
               PASSWORD
               PATH
               SERVICE

WORD_COUNT = BYTE
<points-to: BYTE_COUNT>
<offset: value*2>

ANDX_COMMAND = NO_FURTHER_COMMANDS
NO_FURTHER_COMMANDS = "0xff"

RESERVED2 = BYTE
<unused>

ANDX_OFFSET = BYTE{2}
<unused>

FLAGS = "0x0000"

PASSWORD_LENGTH = BYTE{2}
<target: PASSWORD>
<bigendian>

BYTE_COUNT = BYTE "0x00"

PASSWORD = BYTE{N}
<N = PASSWORD_LENGTH>

PATH = (UNICODE_CHAR)+ "0x0000"
SERVICE = ("IPC" | "?????") "0x00"

UNICODE_CHAR = BYTE{2}
<no delimiter>
```

Appendix B: Dynamic data tainting

The section discusses the used taint propagation approach in more detail. When monitoring an application, we taint (i.e., tag) each input byte that is introduced into the address space of the application process by being read from the protocol message. Each byte receives a unique label. This label establishes a relationship between a particular input byte and a location in memory and allows us to determine the location in the message that certain instructions operate on. Then, we keep track of each labeled value as the program execution progresses. To this end, the output of every instruction that uses a labeled value as input is tagged with the same label as well.

In addition to tracking *direct dependencies*, we also consider *address dependencies*. Such dependencies occur when a tainted operand is used to determine the location from which a value is loaded. In this case, the outcome of the load operation depends not only on the loaded value itself (a direct dependency) but also on the memory address where this value is taken from. A typical example for an address dependency is the use of tainted data as an index into a table. In this case, the result of a table lookup does not directly depend on the input value, but is indirectly influenced by the selection of the respective table element. Here, it is important that address dependencies are tracked as well. Otherwise, the simple transformation of a string contained in the message into its uppercase representation (e.g., using the `toupper()` function) could break the dependencies between the resulting string and the original input. This is because `toupper()` relies on a table that stores the mappings of all 255 possible input characters to their corresponding uppercase representations.

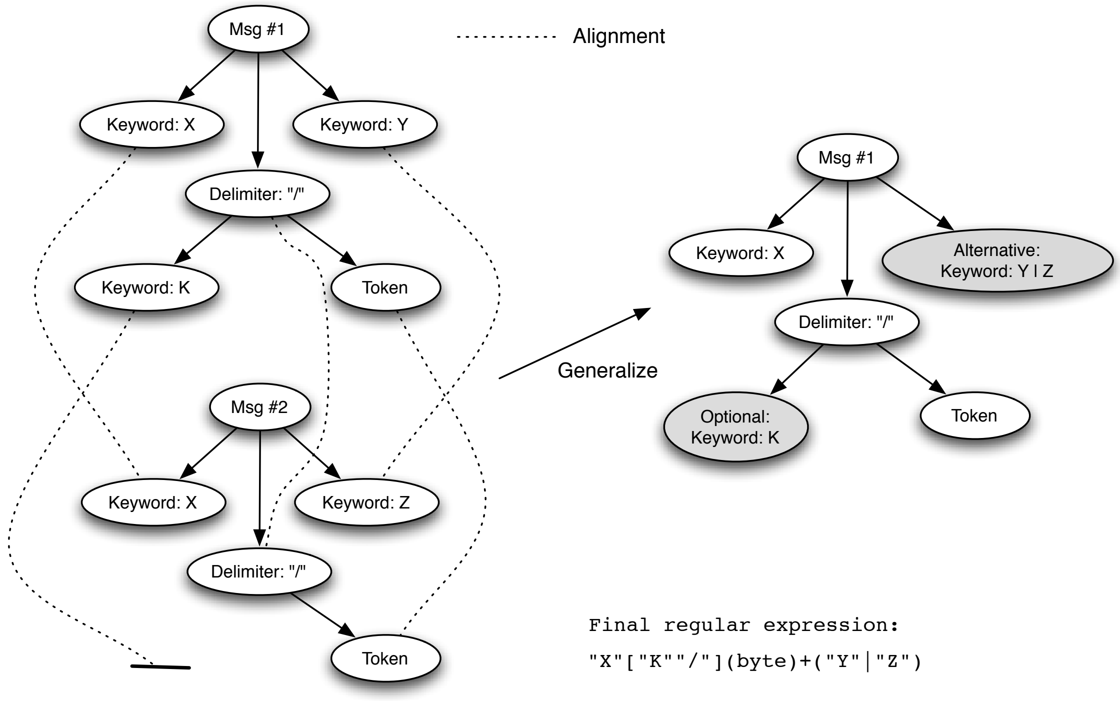


Figure 4. Alignment of two complex messages.

As a result of the dynamic data tainting process, an execution trace is produced for each message. This trace contains all operations that have one or more tainted operands. In these execution traces, we are particularly interested in control flow decisions that depend on tainted input bytes. That is, we check for conditional branch instructions (i.e., comparison operations) that operate on tainted bytes. The reason is that in such cases, values taken from the message drive the control flow of the application, and thus, are likely to hold information that is relevant and related to the protocol specification. Note that when looking for comparisons, we do not only take into account the standard operations such as `test` and `cmp`, but also `sub` or bit-wise `and` instructions that are often used by compilers as fast alternatives. An interesting technical detail is related to the fact that the Intel x86 instruction set does not contain conditional branch instructions that use register or memory operands. Instead, these branch instructions evaluate a number of flag bits, which are usually set by preceding compare or test instructions. As a consequence, our dynamic analysis has to retain the label sets of operands of *compare* and *test* operations until a subsequent conditional branch operation is encountered.

Appendix C: Message alignment example

An example of an alignment between two messages is shown in Figure 4 below. In this figure, the algorithm has determined that the first fields contain the same keyword, and thus, are properly aligned. At the first level, the second nodes are both slash-delimited scope fields. However, in one case, this field holds a keyword and a token field, while in the other case, only a single token field is present. Because the alignment algorithm proceeds recursively, it aligns the two token fields at the second level, and inserts a gap for the keyword field of the first message. Finally, the third fields hold two different keywords, and are aligned with a penalty.

The regular expression generated for the generalized tree is shown at the bottom of the figure. Note that the token field is replaced with a sequence of bytes (as we do not know which and how many bytes will be present in the actual messages). Also, appropriate optional and alternative keyword fields have been introduced. Similar to the grammars shown in Appendix A, keyword and delimiter values are enclosed by quotes.