Polyglot: Automatic Extraction of Protocol Message Format using Dynamic Binary Analysis

Juan Caballero[*], Heng Yin[†*], Zhenkai Liang[*], Dawn Song[‡*]
[*]Carnegie Mellon University [†]College of William and Mary [‡]UC Berkeley
{jcaballero, zliang, dawnsong}@cmu.edu hyin@cs.wm.edu

Abstract

Protocol reverse engineering, the process of extracting the application-level protocol used by an implementation, without access to the protocol specification, is important for many network security applications. Recent work [17] has proposed protocol reverse engineering by using clustering on network traces. That kind of approach is limited by the lack of semantic information on network traces. In this paper we propose a new approach using program binaries. Our approach, *shadowing*, uses dynamic binary analysis and is based on a unique intuition—the way that an implementation of the protocol processes the received application data reveals a wealth of information about the protocol message format. We have implemented our approach in a system called *Polyglot* and evaluated it extensively using real-world implementations of five different protocols: DNS, HTTP, IRC, Samba and ICQ. We compare our results with the manually crafted message format, included in Wireshark, one of the state-of-the-art protocol analyzers. The differences we find are small and usually due to different implementations handling fields in different ways. Finding such differences between implementations is an added benefit, as they are important for problems such as fingerprint generation, fuzzing, and error detection.

1 Introduction

Protocol reverse engineering, the process of extracting the application-level protocol used by an implementation without access to the protocol specification, has become increasingly important for network security. Knowledge of application-level protocol message format is essential for many network security applications, such as vulnerability discovery [11, 20, 26, 31], intrusion detection systems [19], protocol analyzers for network monitoring and signature-based filtering [10, 33], fingerprint generation [12], application dialogue replay [18, 23, 30], detecting services running on non-standard ports [25], and mapping traffic to applications [21]. Many protocols in use, especially on the enterprise network [25, 32], are closed protocols (i.e., no publicly available protocol specification). Even for protocols with a publicly available specification, certain implementations may not exactly follow the specification. Protocol reverse engineering aims to extract the application-level protocol used by an implementation, without access to the protocol specification. Thus, protocol reverse engineering is an invaluable tool for the above network security applications.

Currently, protocol reverse engineering is mostly a painstaking manual task. Attempts to reverse engineer closed protocols such as the MSN Messenger and Samba protocols from Microsoft [4, 1], the Yahoo Messenger protocol [3], or the OSCAR and ICQ protocols from AOL [7, 2], have all been long term efforts lasting many years. In addition, protocol reverse engineering is not a once-and-done effort, since existing protocols are often extended to support new functionality. Thus, to successfully reverse engineer a protocol in a timely manner and keep up the effort through time, we need automatic methods for protocol reverse engineering.

Despite the importance of the problem of automatic protocol reverse engineering, there has been very little research done on the topic. Recently, Cui et. al. [17] proposed using clustering to extract the protocol message format from network traces. The fundamental limitation of their approach is the lack of protocol semantics in network traces, which in turn generates other problems, as we explain at the end of the section.

This material is based upon work partially supported by the National Science Foundation under Grants No. 0311808, No. 0433540, No. 0448452, No. 0627511, and CCF-0424422. Partial support was also provided by the International Technology Alliance, and by the U.S. Army Research Office under the Cyber-TA Research Grant No. W911NF-06-1-0316, and under grant DAAD19-02-1-0389 through CyLab at Carnegie Mellon. The views and conclusions contained here are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of ARO, NSF, or the U.S. Government or any of its agencies. This work was also supported in part by the Korean Ministry of Information and Communication (MIC) and the Korean Institute for Information Technology Advancement (IITA) under program [2005-S-606-02, Next Generation Prediction and Response Technology for Computer and Network Security Incidents].

Approach: In this paper we propose *shadowing*, a new approach for automatic protocol reverse engineering. Instead of extracting protocol information purely from network traces, we leverage the availability of a program binary implementing the protocol. Compared to network traces, which only contain syntactic information, program binaries also contain semantic information about how the program processes and operates on the protocol data. In addition, they are the main source of information about the *implementation* of a protocol. Thus, by extracting protocol information from the program binary implementing the protocol, rather than purely from network traces, our approach can be more accurate and provide richer information about the protocol and its implementation. We call our approach *shadowing* because we closely monitor the program as it processes an input.

Given the program binary, we could extract protocol information from it by using either static or dynamic analysis. Purely static analysis of binaries is difficult due to challenges such as memory aliasing analysis or unresolved indirect jumps. In this paper, we focus on using dynamic analysis for automatic protocol reverse engineering. In the near future, we plan to extend our work to combine dynamic analysis with static analysis.

Shadowing uses dynamic analysis and is based on a unique intuition—the way that an implementation of the protocol processes the received application data reveals a wealth of information about the protocol. In particular, by monitoring all the operations done by a program while processing its input, we can extract the format of the received data.

Scope of the problem: Protocol reverse engineering is a complex task that involves extracting: 1) the *protocol message format*, which describes the message format for the messages that comprise the protocol, and 2) the *protocol's state machine*, which specifies the protocol states and the transitions between states according to the messages sent or received. In this paper, we focus on extracting the protocol message format, because 1) it is a necessary step to extract the protocol's state machine, and 2) as we will show, it is challenging enough in its own right.

Our approach to extract the protocol message format is to process one message at a time and output the message format for each given message. Our problem is then to automatically extract the message format, when given as input a program binary implementing the protocol, and a message received by that program. The main challenge to extract the message format is to find the field boundaries in the message. Protocols include both fixed-length and variable-length fields. The difficulty with fixed-length fields is to determine the boundary between consecutive fields to avoid joining two fixed-length fields together or splitting a single fixed-length field into many. The difficulty with variable-length fields is that protocols can use different elements to mark the field boundary such as 1) *direction fields*, that store information about the location of another target field in the message, for example length fields and pointer fields, and 2) *separators*, i.e., constant values that mark the boundary of the field. Thus, we first need to locate those elements to identify the field boundaries. Another challenge in extracting the protocol message format is to identify the *protocol keywords*. Keywords are protocol constants that appear in the messages sent over the network. In this paper we present techniques to address all these challenges.

To realize our shadowing approach, we have designed and implemented a system, called *Polyglot*. Our system works on stripped binaries and does not require the availability of source code or any debugging information in the binaries. We have extensively evaluated our approach using eleven implementations from five different protocols: DNS, HTTP, IRC, Samba and ICQ. We have included both clients and servers working under Windows and Linux. The protocols analyzed, include difficult to find elements such as length fields, pointer fields, separators and keywords. We compare our results with the manually crafted message format, included in Wireshark, one of the state-of-the-art protocol analyzers. The differences we find are small and usually due to different implementations handling fields in different ways. Finding these differences between implementations is an added benefit, as they are important for problems such as fuzzing [31], error detection [11] and fingerprint generation [12].

Contributions: In summary, this paper makes the following contributions:

- **New approach for extracting the protocol message format using program binaries:** We propose to use a new paradigm called *shadowing* to automatically extract the message format from some input application data. Our approach is based on the intuition that analyzing how a program processes its input allows us to understand the format of the received data. This is in contrast to previous techniques that try to infer protocol information purely from network traces [17].
- **New techniques for detecting direction fields:** *Direction fields* are fields used to mark the boundary of variable-length fields, such as length fields or pointer fields. Currently, the only available techniques to detect some types of direction fields are the ones used in [17, 18]. Those techniques are limited because they need to assume the encoding of the length field. We propose the first techniques to detect direction fields, without making assumptions about the encoding used by the direction field.
- **New techniques for detecting separators:** Separators are constant values that can be used, instead of direction fields, to mark the boundary of variable-length fields. We propose what we believe are the first techniques to discover separators using no prior assumption about the separator values. Thus, our techniques can handle unknown protocols that use separators, independently of the separator value. In contrast, previous work assumes separators can only be used in text

protocols and assume that separators only take a few predefined values such as white space or tab [17].

- **Finding multi-byte fixed-length fields:** We present a method to find the boundary of multi-byte fixed-length fields, by examining how the program groups together the input bytes. The intuition is that fields are semantic units and thus programs need to use multi-byte fields as a single unit. Even though our technique has limitations, it is still a significant improvement over previous work, which cannot find boundaries between consecutive binary fields, and thus have to assume that each byte that shows binary encoding is a separate field [17].
- **New techniques for extracting protocol keywords:** Current techniques for extracting keywords find protocol keywords by looking for tokens that occur repeatedly at the same position in multiple messages [17, 21, 25]. In contrast, our techniques extract the keywords present on a single message. Thus, previous work might miss keywords which appear in different positions across multiple messages (e.g., in HTTP). Our solution would still identify the keywords in each message and the different locations where they occur.

The remainder of the paper is organized as follows. Section 2 defines our problem. In Section 3 we describe the approach and system architecture. Then, in Sections 4–6 we present our techniques to find the field boundaries and the protocol keywords. We evaluate our system in Section 7 and summarize related work in Section 8. Finally, we present future work in Section 9 and conclude in Section 10.

2 Problem Definition

In this section we introduce the terminology used in this paper, then the scope of the problem, and finally, the problem definition.

2.1 Terminology and Scope of the Problem

Protocol terminology can sometimes be intricate and may not be standard. Thus, for clarity, we first introduce the terminology we will use in this paper.

Protocol elements: Protocols have a hierarchical structure comprised of *sessions*, which are comprised of a sequence of *messages* which are in turn comprised of a sequence of *fields*, where a *field* is the smallest contiguous sequence of application data with some meaning. For example, an HTTP session may contain multiple messages such as several GET requests to fetch different contents (e.g., one for the HTML file, another for an image, etc), and an HTTP GET message may contain several fields such as the method, the URL, and the version, as illustrated in Figure 2.

Scope of the problem: Reversing a protocol is a complex task that involves extracting: 1) the *protocol message format*, which describes the format of the messages that comprise the protocol, and 2) the *protocol's state machine*, that depicts the different protocol states and the transitions between states according to the messages sent or received. As explained in Section 1, in this paper we focus on extracting the protocol message format and leave the study of the protocol's state machine for future work.

Attribute	**Value**
Field Start	Start position in message
Field Length	Fixed-size, Variable-size
Field Boundary	Fixed, Direction, Separator
Field Type	Direction, Non-Direction
Field Keywords	Position and value of the field's keywords

Table 1: Field format attributes used in this paper.

The *protocol message format* is the set of the protocol's *message formats*, where a message format is a sequence of *field format*s and a field format is a group of attribute-value pairs.

Our approach to extract the protocol message format is to process one message at a time and output the message format for the given message. We assume synchronous protocols to identify the message boundaries. In this paper, we consider five pivotal attributes in the field format: the field start position in the message, the field length, the field boundary, the field type and the field keywords. Table 1 shows these five attributes. The field start attribute captures the position of the field in the given message. The field length attribute states if the field has a fixed length (and the corresponding value) or if it has variable length. Then, the field boundary attribute determines how the program finds the boundary of the field (i.e., where the field ends). For fixed-length fields, the value is always *Fixed* since the program knows a priori the length. For variable-length fields, it can be *separator*, i.e., a constant value that marks the boundary of the field, or *direction*, i.e., a field that stores information about the location of another target field in the message.

The field type attribute provides semantic information about the field. Currently, we only consider whether a field is, or is not, a direction field. Finally, the field keywords attribute contains a list of protocol keywords contained in the field, specifically their value and position. We define *keywords* to be protocol constants that appear in the protocol application data. There are other protocol constants that do not appear in the protocol application data and thus are not keywords, such as the maximum length of a variable-length field, known to the parties but never sent over the network.

Extracting the keywords is important because they allow to differentiate which parts of the field are protocol-dependant and which are user or session dependant. This information is useful for multiple problems such as fingerprint generation, application dialogue replay and error detection. In addition, keywords can be used to map traffic to specific protocols, which in turn can be used to identify tunneled protocols, such as P2P traffic over HTTP, and services running on non-standard ports [21, 25].

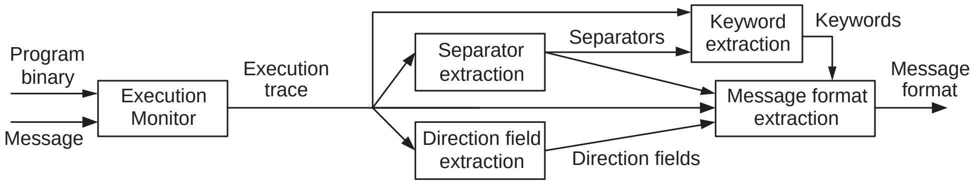

Figure 1: System Architecture.

2.2 Problem Definition

This paper deals with the problem of extracting the protocol message format. The protocol message format includes multiple message formats. Our problem is then, given a number of messages received by a program binary implementing the protocol, to individually extract the message format of each of those messages.

The main challenge in extracting the message format is to find the field boundaries. Protocols include both fixed-length and variable-length fields. For fixed-length fields, the boundary is known a priori by the program. The difficulty is to determine the boundary between consecutive fixed-length fields, to avoid joining two fixed-length fields together or splitting a single fixed-length field into two. We deal with the problem of finding the boundary of fixed-length fields in Section 6.

For variable-length fields, the program needs to determine the field boundary dynamically. Here, the difficulty is that protocols can use different elements to mark the field boundary such as 1) *direction fields*, that store information about the location of another target field in the message, for example length fields and pointer fields, and 2) *separators*, i.e., constant values that mark the boundary of the field. Thus, we first need to locate these elements. We deal with the problem of finding the boundary of variable-length fields using direction fields in Section 4.1, and using separators in Section 5.1.

Another challenge in extracting the message format, is to identify the keywords contained in each field. We deal with the problem of extracting the protocol keywords in Section 5.2.

More complex properties: Polyglot extracts the message format of a single message. However some properties might require analyzing information about multiple messages simultaneously. For example, some protocols such as HTTP allow some of their fields to be ordered differently. We term a field with this property to be a *floating field*. This in turn, means that keywords associated with a floating field can appear in different positions in the message. Currently, given two messages with the same keywords in different positions, we can identify for each individual message what keywords it contains. Clearly, without looking at multiple messages, we cannot determine that the keywords (and their associated fields) are floating, i.e., they can appear at different positions. Thus, we rely on an analyst to infer this kind of properties when given the message formats for multiple messages (possibly with keywords present in multiple different positions). We leave the automatic extraction of such properties for future work.

To summarize, our problem statement is as follows:

Problem Definition: Given an implementation of a protocol in the form of a program binary and a message received by that program, our problem is to output the message format, with no a priori knowledge about the protocol that the message belongs to. Extracting the message format consists of two main tasks: 1) find the field boundaries for fixed-length and variable-length fields, which includes identifying the separators and direction fields, and 2) identify the keywords in each field.

3 Approach and System Architecture

In this section we present our approach and introduce the system architecture of Polyglot.

Our approach, using dynamic analysis for protocol reverse engineering, is based on a unique intuition—the way that an implementation of a protocol processes the received application data reveals a wealth of information about the protocol message format. Using this intuition, we propose *shadowing*, a new paradigm based on dynamically analyzing how a program binary processes its input to extract the format of the received application data.

To enable dynamic analysis for automatic protocol reverse engineering, the high-level architecture of Polyglot has two phases. First, we watch over the program execution as it processes a given message. This phase generates a record of the program's processing, which contains all necessary information about the execution. Second, we analyze the record of the program's processing and extract information about the field boundaries and the keywords that form the basis for the message format.

Figure 1 shows the system architecture. The first phase is implemented by the *execution monitor* [37]. It takes as input the program's binary and the application data, and dynamically monitors how the program processes the application data. The output of the execution monitor is an *execution trace* that contains a record of all the instructions performed by the program. The execution trace forms the input to our analysis in the second phase.

The execution monitor implements dynamic taint analysis [13, 14, 15, 28, 34, 35]. In dynamic taint analysis, input data of interest is marked (i.e., tainted) when it arrives and any instruction that operates on the tainted data (e.g., moving it to another location or performing an arithmetic or logical operation), propagates the taint information to the destination. For our purposes, we taint any data received from the network. Thus, the execution trace contains, for each tainted register and memory location used in an instruction, the offset positions that were used to compute its value. For example, if the method field in Figure 2 is moved to a processor register (e.g., EAX), the register gets tainted with positions 0 through 3, corresponding to the original offset in the received data. Dynamic taint analysis is well understood and we provide its details in Appendix A.

In the second phase, we analyze the execution trace to locate the field boundaries and the keywords. Note that currently our analysis is offline (using the execution trace), but it could also be performed online, integrated with the execution monitor. This phase consists of four modules: the *separator*, *direction field*, *keyword* and *message format extraction modules*, which we now describe.

First, the *direction field* and the *separator extraction modules* take care of finding the boundaries of variable-length fields. We introduce them in Sections 4.1 and 5.1 respectively. Next, the *keyword* extraction module takes as input the separators and the execution trace and outputs the keywords. We present the keyword extraction module in Section 5.2. Finally, the *message format* extraction module takes care of finding the boundaries of fixed-length fields and of combining all previous information to generate the message format. It takes as input the previously found separators, direction fields and keywords, as well as the execution trace, and outputs the message format.

4 Direction Field Extraction

In this section we describe our techniques for identifying direction fields, which store information about the location of another target field in the message.

4.1 Direction Field Extraction

4.1.1 What is a direction field?

Direction fields are fields that store information about the location of another field in the message (called the *target field*). The most common direction fields are *length* fields, whose value encodes the length of a target field. The target field usually has variable-length and the length field allows to find the end of the target field. Figure 3 shows an example length field and its target. In addition to length fields, other types of direction fields are: *pointer* fields and *counter* fields. Pointer fields, encode the displacement of a field start with respect to some other position in the message. One example of an pointer field is the compression scheme used in DNS labels to avoid repetition, which represents a position from the beginning of the DNS header. Counter fields encode the position of a field in a list of items. One example of a counter field is the number of DNS authoritative records in a DNS response.

4.1.2 Techniques for identifying direction fields:

The intuition behind our techniques for direction field detection is the following. The application data is stored in a memory buffer before it is accessed (it might be moved from disk to memory first). Then a pointer is used to access the different positions in the buffer. Now when the program has located the beginning of a variable-length field, whose length is determined by a direction field, it needs to use some value derived from the direction field to advance the pointer to the end of the field. Thus, we identify direction fields when they are *used* to increment the value of a pointer to the tainted data. For example, in Figure 3 we identify the length field at positions 12-13 when it is used to access positions 18-20.

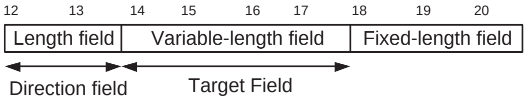

Figure 3: Direction field example.

We consider two possibilities to determine whether a field is being used as a direction field: 1) Either the program computes the value of the pointer increment from the direction field and adds this increment to the current value of the pointer using arithmetic operations; or 2) the program increments the pointer by one or some other constant increment using a loop, until it reaches the end of the field, indicated by a stop condition.

Below, we describe how to identify the direction fields in these two cases.

Incrementing the pointer using arithmetic operations: For the first case, the program performs an indirect memory access where the destination address has been computed from some tainted data. Thus, when we find an indirect memory access that: 1) accesses a tainted memory position, and 2) where the destination address has been computed from tainted data (i.e., the base or index registers used to compute the memory address were tainted), we mark all the consecutive positions used to compute the destination address as part of a length field. In addition, we mark the smallest position in the destination address as the end of target field. For example, in Figure 3 if the instruction is accessing positions 18-20, and the address of the smallest position (i.e., 18) was calculated using taint data coming from positions 12-13, then we mark position 12 as the start of a direction field with length 2, and position 18 as the end of the target field. If a direction field is used to access multiple positions in the buffer, we only record the smallest position being accessed. For example, if we have already found the length field in Figure 3 directs to position 18, and it appears

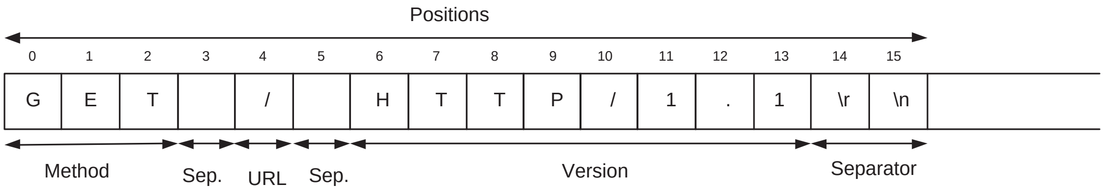

Figure 2: Simple HTTP GET query. When the program moves or operates on the input data, the destination gets tainted with the original offset position of the input data.

again in an indirect memory access to position 27, we still consider the end of the target field to be position 18.

Incrementing the pointer using a loop: For the second case, since the pointer increment is not tainted (i.e., it is a constant) then the previous approach does not work. In this case we assume that the stop condition for the pointer increment is calculated using a loop. Thus, we look for loops in the trace that have a tainted condition.

Our loop detection component extracts the loops present in the execution trace. For this, we search for sections of repeated code that include a backwards jump, that is, a jump to a lower instruction pointer. After extracting the loops we check if the loop stop condition is generated from tainted data, if so we flag the loop as tainted. Every time the program uses a new position, we check if the closest loop was tainted. If so, we flag a direction field.

Our techniques are not complete because there are other possibilities in which a program can indirectly increment the pointer, for example using switch statements or conditionals. But, these are hardly used since the number of conditions could potentially grow very large, up to maximum value of the direction field. We plan to incorporate support for other types of indirect increments in the future.

Variable-length fields: Direction fields are normally used to locate the end of the target field, since the target field usually has variable length. To determine the start of the target variable-length field, without assuming any field encoding, we use the following approach. Direction fields need to appear before their target field, so they can be used to skip it. Most often, as mentioned in [18] they precede the target field in the field sequence. After we locate a direction field, we consider that the sequence of bytes between the last position belonging to the direction field and the end of the target field, corresponds to a variable length field. For example, in Figure 3, when the length field at positions 12-13 is used to access positions 18-20, we identify everything in between (i.e., 14-17) to be a variable-length field. Thus, if a fixed-length field follows the variable length field and it is not used by the program either because the field is not needed or not supported by the program, then we will include the fixed-length field as part of the variable length field.

Note that our approach detects direction fields by looking at pointer increments and thus, it is independent of the encoding used in the direction field. In contrast, previous work uses techniques for identifying length fields that assume the length is encoded using some pre-defined encoding, such as the number of bytes or words in the field [17, 18]. Thus, those techniques would miss direction fields if they use other encodings, which do not belong to the set of pre-defined encodings being tested.

Pointer decrements: So far, we have described techniques that consider only pointer increments. There are some types of direction fields such as backward pointers, commonly used in DNS replies, which are used to decrement a pointer rather than increment it. Suppporting pointer decrements would require modifying our techniques to account for target fields that have already been explored. We believe the needed modifications to be small and plan to implement them in the near future.

5 Separator and Keyword Extraction

In this section we describe our techniques for identifying separators and keywords.

5.1 Separator Extraction

5.1.1 What is a separator?

Separators are elements used by protocols to mark the boundary of variable-length fields. A *separator* is a pair of two elements: 1) a constant value, and 2) a scope. The constant value marks the boundary of the field, while the scope contains a list of position sequences in the application data, where the separator is used.

If the separator is used to separate fields, then it will have to be compared against each byte in the application data and its scope will be the position sequence that encompasses all positions, from zero to the number of bytes received. For example, in Figure 2 the Carrier-Return plus Line-Feed $(\backslash r \backslash n)$ sequence at positions 14 and 15 is a field separator, and its scope would be 0 through 15. On the other hand, the scope of an in-field separator, used to separate different elements inside a field, will usually be the position sequence where the field appears. For example, in Figure 2 the slash character could be used to separate the HTTP keyword from the version number, and its scope would be 6 through 13.

Protocols can have multiple separators, usually for different scopes: one for message end, another for field end and possibly various in-field separators. Also, sometimes multiple separators can be used at the same scope. For example in HTTP both the space (0x20) and the tab (0x09) are in-field separators, used in different fields to separate the field name from the field data.

Separators are part of the protocol specification and are known to the implementations of the protocol. They can be used in binary, text or mixed protocols. For example separators are used in HTTP which is mainly a text protocol and also used in Samba which is mainly a binary protocol. Separators can be formed by one or more bytes. For example HTTP uses Carrier Return plus Line Feed (0x0d0a) as field separator, while Samba uses a null byte (0x00) to separate dialect strings inside a Negotiate Protocol Request.

5.1.2 Techniques for identifying the separators:

To find the field boundaries, programs need to identify the value and location of any separator that appears in the application data. This is done by comparing the different bytes received from the network against the separator values and when a true comparison is seen, a field boundary has been found[1].

Clearly, not all true comparisons against constants are separators. What distinguishes a separator from another constant is that the separator needs to be compared against most (usually all) the bytes in its scope. For example a message separator would be compared against all bytes received. The same applies to a field separator, but an in-field separator would only be compared against the bytes in that specific field. Since the scope of message and field separators are the same, currently we cannot distinguish them. We assume synchronous protocols to identify the message boundaries.

To find the separators, we look for tokens that are compared against consecutive positions in the stream of data. Note that we do not require these comparisons to appear in consecutive instructions, only in consecutive positions of the buffer. That is, we do not require the program to perform a sequential scan of the buffer looking for the separator. This is more general since a program could for example scan backwards to find an in-field separator.

Our concept of a comparison extends to multiple instructions that compilers use to compare operands. In addition to normal comparison operations, we also include substraction operations, string comparisons and some operations that compilers use to cheaply compare if an operand has zero value, such as performing a logical AND operation with itself using a test instruction.

Currently, we identify separators in a three-step process: First, we generate a summary of all the program's comparisons involving tainted data. Then, we use this summary to extract byte-long separator values. Finally, we extend separator bytes into multi-byte separators, if needed. We now explain these three steps in detail.

		Offset Positions						
		0	1	2	3	4	5	6
Tokens	0x0a (\n)	x	x	x	x	x	x	x
	0x47 (G)	x						
	0x45 (E)		x					
	0x54 (T)			x				
	0x2f (/)					x	x	x

Figure 4: Token table. Each entry in the tokens-at-position table represents one column of the token table. Each entry in token-series represents one row of the token table.

1) Generating the token tables: The first step is to extract a summary of all the programs's comparisons. This summary is shown in Figure 4 as a conceptual *token table*, that displays the comparisons performed by the Apache webserver on the first 7 bytes of a HTTP GET request. The rows represent token values that appear in comparisons and the columns represent positions in the application data. A token is a byte-long value. An X in the table means that the token from that row was compared, at some point in the program, against the positions from that column.

We implement this conceptual token table using two hash tables: the *tokens-at-position* table and the *token-series* table. The token-series table contains for each token, all the buffer positions to which the token was compared, thus each entry corresponds to a row of the token table. The tokens-at-position table contains for each buffer position, the ordered list of tokens that it was compared against, thus each entry corresponds to a row of the token table. The tokens-at-position and the token-series tables are also used in the keyword extraction module.

To populate the tables, we scan the execution trace and for each comparison found that involves at least a tainted byte, we update the tables with the token, the position, the value it was compared against, and the result of the comparison.

2) Extracting byte-long separators: Our intuition is that any comparison between a tainted byte and a non-tainted byte, can potentially denote a separator. Thus we scan the token-series table and for each token, we extract the list of consecutive buffer positions it was compared with. We require a minimum series length of three, to avoid spurious comparisons. We also require the token to appear in at least one position in the series to avoid easy obfuscation by generating innocuous comparisons. The output of this phase is a list of byte-long separators with the associated context (i.e., positions) where they are used.

3) Extending separators: When a separator value consists of multiple bytes, such as the field separator in HTTP (0x0d0a), the program can use different ways to find it, such as searching for the complete separator, or only searching

[1] If the data contains the separator value, escape sequences can be used.

for one separator byte and when it finds it, checking if the remaining separator bytes are also present. Thus, in our previous phase we might have identified only one byte of the separator or all the bytes but as independent byte-long separators.

In this last phase, we try to extend each candidate separator. For each appearance of the byte-long separator in its context, we check the value of the predecessor and successor positions in the application data. If the same value always precedes (or succeeds) the byte-long separator being analyzed, and the program performs a comparison against that value, then we extend the separator to include that value. We do not extend byte-long separators that appear less than a minimum number of times in the session data (currently four) to avoid incorrectly extending a separator. Also, we don't extend any separator beyond a maximum length (currently four), since long separators are uncommon.

Note that our approach does not assume any separator value. Thus, we can potentially support any unknown protocol that uses separators. This is in contrast to previous work that assumes the separators to be white space, tab or any non-printable byte [17].

Multiple valid separators for the same scope: Protocols might use multiple separators for the same scope. For example, HTTP allows both space and tab as valid separators in the Status-Line. Thus, in Figure 2 any of the spaces at offsets 3 and 5 could be replaced by a tab. Note that, even if multiple separators are allowed, the application receiving the message does not know apriori which separator was used and therefore it needs to compare the received data against each of the valid separators in the context. This presents us with a trade-off. As explained in step 2 above, we require a token to appear in at least one position in the sequence of comparisons before marking it as a byte-long separator. This helps to avoid easy obfuscation by generating innocuous comparisons, but it does not allow us to detect other valid separators in the context if they are not present in the message. Thus, currently our default behavior favors robustness against obfuscation, but we allow the analyst to manually change this default behavior to allow detection of other separators not present in the message.

5.2 Keyword Extraction

5.2.1 What is a keyword?

We have defined keywords to be protocol constants that appear in the received application data. As explained in Section 2, in this work, we do not attempt to extract *all* protocol constants, since there are constants, such as the maximum length of a field, that never appear in the application data. The problem is to extract the *subset* of all protocol constants, that are 1) supported by the implementation, and 2) present in the application data received by the program. Thus, we want to identify which segments of the application data correspond to protocol keywords supported by the implementation. In the near future, we plan to combine our dynamic approach with additional static analysis to locate other protocol constants that do not appear in the application data.

Any protocol, whether text-encoded, binary-encoded or mixed can use keywords. Keywords can be strings (i.e., the HTTP *Host* field name) or numbers (i.e., the version in the IP header). One can be misled to think that in text-encoded protocols, keyword extraction is trivial, but given the different text encodings and the problem of distinguishing a keyword from other data (i.e., user or session data), this is not commonly so.

5.2.2 Techniques for identifying the keywords:

Keywords are known a priori by the protocol implementations. Thus, when application data arrives, the program compares the keywords against the received application data. Our intuition is that one can locate the protocol constants present in the session data by following the *true comparisons* between tainted and untainted data.

The keyword extraction process is comprised of two phases. The first phase is identical to the first phase of the separator extraction module, that is, to populate the tokens-at-position and token-sequences tables. The second phase differs in that it focuses on the true comparisons, rather than all the comparisons. It consists of exploring, in ascending order, each position in the tokens-at-position table. For each position, if we find a true comparison, then we concatenate the non-tainted token to the current keyword. If no true comparison was performed at that position, we store the current keyword and start a new one at that position. We also break the current keyword and start a new one if we find a separator value in the middle of the keyword. Note that our approach is general, in that it does not assume that the multiple bytes that form the keyword appear together in the code or that they are used sequentially.

In addition to protocol keywords, configuration information such as DNS records, or HTML filenames can also be seen when analyzing the true comparisons. To differentiate between configuration information and protocol keywords, we need to define file reads also to contain sensitive information and thus data read from file also becomes tainted, though with different taint origin, flow identifier and offset values.

6 Fixed-length Field Extraction

In Sections 4 and 5.1 we have presented our techniques to identify the boundaries of variable-length fields. In this Section we present our techniques to identify the boundaries of fixed-length fields. As defined in Section 2, a field is a contiguous sequence of application data with some meaning. As such, programs take decisions based on the value of the field as a whole. Thus, when a field is composed of multiple bytes, those multiple bytes need to be used together, forming a semantic unit that can be used in arithmetic operations, comparisons or other instructions. In addition, most fields are independent of other fields, so bytes belonging to different fields rarely are used in the same instruction. The exception to this rule are special relationships such as length fields, pointer fields or checksums.

Our approach for identifying multiple bytes belonging to the same field is the following. Initially, we consider each byte received from the network as independent. Then, for each instruction, we extract the list of positions that the taint data involved in the instruction comes from. Next, we check for special relationships among bytes, specifically in this paper we check for direction fields, using the techniques explained in Section 4.1. If no direction field is found, then we create a new fixed field that encompasses those positions. For example if an instruction uses tainted data from positions 12-14 and those positions currently do not belong to a length field, then we create a fixed field that starts at position 12 and has length 3.

If a later instruction shows a sequence of consecutive tainted positions that overlaps with a previously defined field, then we extend the previously defined field to encompass the newly found bytes. One limitation is that fixed-length fields longer than the system's word size (four bytes for 32-bit architectures, eight for 64-bit architectures) cannot be found, unless different instructions overlap on their use. Note that fields larger than 4 bytes are usually avoided for this same reason, since most systems have 32-bit architectures where longer fields need several instructions to be handled. For fields longer than 4 bytes, our message format truncates them into four-byte chunks. Note that this does not affect variable-length fields which are identified by finding the separators and the direction fields.

Even with this limitation, our approach is an improvement over previous work [17], where each binary-encoded byte is considered a separate field. Using that approach, two consecutive fixed-length fields, each of length 4 bytes, would be considered to be 8 consecutive byte-long fixed-length fields.

7 Evaluation

In this section we present the evaluation results of our system. We have evaluated our system extensively using 11 different programs implementing 5 different protocols (HTTP, DNS, IRC, ICQ and Samba) as shown in Table 2. Most of the binaries analyzed are Windows servers but we also include one ICQ client and a Samba server running on Linux Fedora Core 5, to show that our system can potentially work on any IA-32 binary. The test suite shows that our approach can handle real protocols and real size programs, such as Apache, Bind and Samba.

Program	Version	Type	Size
Apache	2.2.4	HTTP server (Win)	4,344kB
Miniweb	0.8.1	HTTP server (Win)	528kB
Savant	3.1	HTTP server (Win)	280kB
Bind	9.3.4	DNS server (Win)	224kB
MaraDNS	1.2.12.4	DNS server (Win)	164kB
SimpleDNS	4.00.06	DNS server (Win)	432kB
TinyICQ	1.2	ICQ client (Win)	11kB
Beware ircd	1.5.7	IRC server (Win)	148kB
JoinMe	1.41	IRC server (Win)	365kB
UnrealIRCd	3.2.6	IRC server (Win)	760kB
Sambad	3.0.24	Samba server (Lin)	3,580kB

Table 2: Different program binaries used in our evaluation. The size represents the main executable if there are several.

The protocols under study include difficult-to-find elements such as length fields, pointer fields, and separators, and we compare our results, obtained with no protocol knowledge, with the manually crafted message formats included in Wireshark[2] one of the state-of-the-art protocol analyzers. The results show that we correctly identify most field boundaries and that the differences are usually due to different implementations handling fields in different ways. Finding these differences between implementations is important for problems such as fingerprint generation, fuzzing, and error detection.

7.1 Message Format Results

Protocols are comprised of many different messages. For each protocol under study, we select one representative message and capture an execution trace while the program processes that message. We now present the results of extracting the message format for the different messages.

DNS query: The session under study is composed of two messages: a DNS query and a DNS reply. The query is the same for all three servers. It requests the IP address of the host *www.example.net* and we analyze how the request is parsed by each DNS server. Figure 5 shows the message format obtained from each server compared to the one from Wireshark, shown on the left.

The results show that we correctly identify the message format including length fields and variable fields. The word 'Unused' in the figure indicates that there was no operation, other than moves, performed by the program on those specific bytes. For example, we see that Bind does not perform any check on the Identification (ID) field. Since any 16-bit value is allowed for ID, Bind can move the field directly into the data structure used to construct the reply, with no further operation. Also, SimpleDNS operates on the Total Queries field but it ignores the Total Answers, Authority and Additional fields. This behavior is fine since those fields are not used in the request, though it does not allow to detect malformed requests that set those fields to values different than zero. Knowledge about differences between implementations of the same protocol, such as unused fields, is important for applications like fingerprint generation and error detection.

[2] Previously known as Ethereal

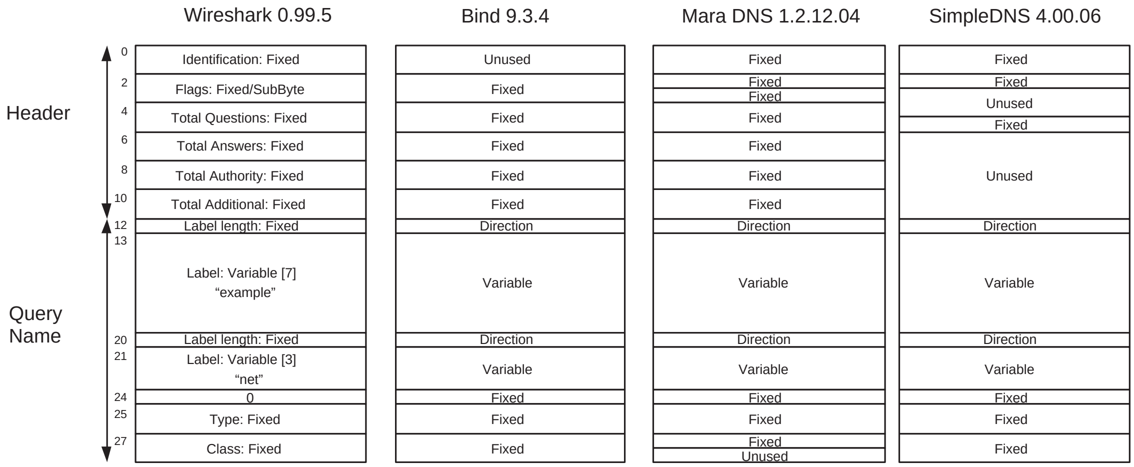

Figure 5: Message format extracted from a DNS query sent to all three DNS servers. On the left we present the message format from Wireshark.

DNS query using pointer: DNS allows using a compression scheme which eliminates the repetition of domain names in a message. An entire domain name or a list of labels at the end of a domain name is replaced with a pointer to another occurance of the same name [27]. To verify the detection of an pointer field, we create a DNS query with a forward pointer, So the first name is *www* and a pointer to the next name, which holds the value *example.net*. The complete query is *www.example.net*. This type of forward pointer is only allowed by the SimpleDNS server. The results show that the pointer field and the length fields are properly identified, the rest of the fields are similar to the standard DNS query above. For brevity, the message format is shown in Figure 8 in Appendix B.2.

ICQ login: The session under study is a login session, where the client sends a fake username and password to the server, and the server replies denying the session. We extract the field format from the reply sent by the server.

Again, we properly identify the direction fields. The main difference with the DNS results is that there are some unused fixed-length fields following the variable-length fields, and those fixed-length fields are incorrectly merged into the variable-length one. This happens because the ICQ client is adding a constant of value 2 to the value of the length field to skip both the follow-up variable-length target field and the unused field simultaneously. This shows that our techniques are able to identify the pointer increments but are not currently able to describe how the increment was derived from the direction field value, which would be needed to observe that a constant of value two was added to the length field. For brevity, the message format is presented in Figure 7 in Appendix B.1 and it shows the Value ID fields being merged with the variable-length Field Data fields preceding them.

Separator	Apache	Savant	Miniweb
0x0d0a ('CRLF')	field	field	field
0x2f ('/')	in-field	in-field	-
0x2e ('.')	in-field	in-field	in-field
0x20 (' ')	-	in-field	in-field
0x3a20 (': ')	in-field	-	-

Table 3: Separators extracted from an HTTP GET request sent to all three HTTP servers.

HTTP GET query: The session under study is a HTTP GET request for the *index.html* webpage and it corresponding reply. We analyze the GET request. So far, we have shown the message format for protocols that use direction fields to mark the boundary of variable-length fields. But, separators can also be used to mark those boundaries.

Table 3 shows the results from the separator extraction module for the three HTTP servers. The HTTP GET request used, includes several separators but does not include any direction fields. Thus, the message format is determined by the location of the separators. Each row represents a separator value, shown both in hexadecimal and ASCII, and each table entry states if the field was used or not by the server. For brevity, rather than the full scope (i.e., all sequences where it appeared), we show a tag with values field/in-field to indicate the scope of the separator.

The results show that the three servers use a similar set of separators. For all three servers, the field separator has been properly expanded to two bytes (0x0d0a). The space character (0x20), or semicolon and space for Apache, is used to parse the fields, separating the field name from the field data. Another separator is the dot character (0x2e), used to find the file extension in all three servers, plus being used by Apache to parse the IP address in the *Host* field. Finally, Apache and Savant use the slash character (0x2f) to find the beginning of the path in the URL.

Samba negotiate protocol request: The session under study is a Samba *negotiate protocol* request and its cor-

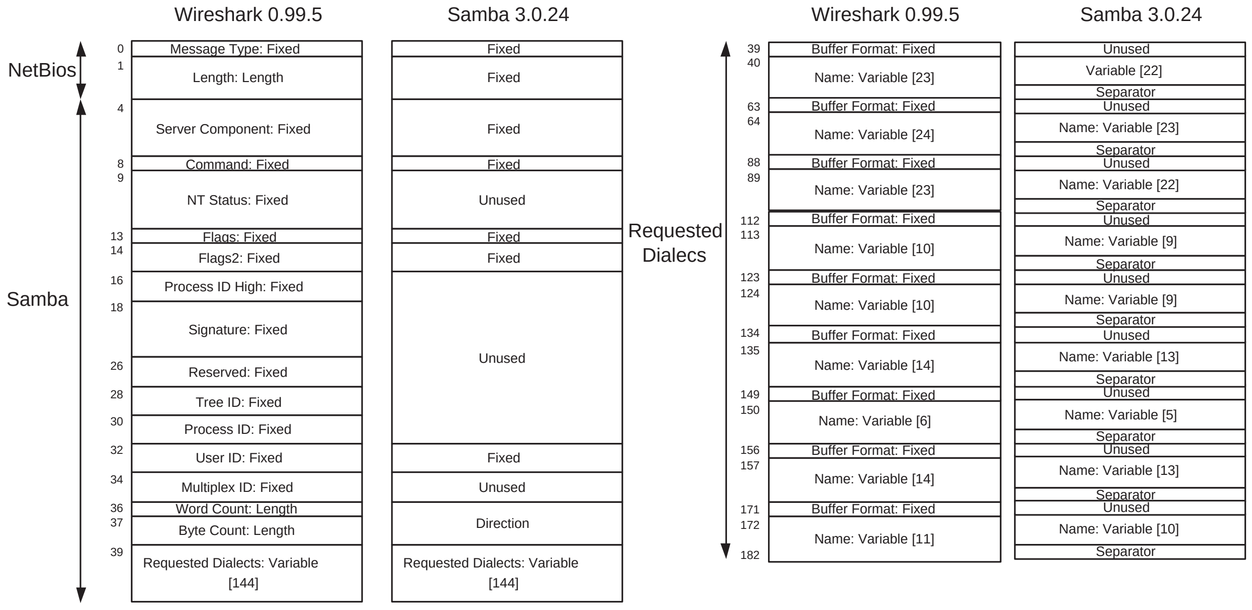

Figure 6: Message format extracted from a Samba negotiate protocol request. On the left we present the message format from Wireshark.

responding reply. We analyze the request. So far, the extracted message format have used either only direction fields or only separators to mark the boundary of variable-length fields. The Samba negotiate protocol request uses both. It uses length fields to mark the end of the Samba request, and the null byte (0x00) to mark the end of variable-length strings. Figure 6 shows the results, after both separators and direction fields have been extracted. On the left is the message format while on the right we zoom into the *requested dialects* field which uses the separators.

The message format shows both the *Samba word count* and *byte count* (near the bottom) combined together into a single direction field. This is because the server uses both fields simultaneously to establish the total length of the request (2*wordCount + byteCount). The Netbios length field near the top is reported as a fixed-length field because although the server combines the two bytes of the field into a two-byte field, it does not uses the resulting field. Instead, it uses the Samba word count and byte count fields to identify the end of the Samba request. In the requested dialects field, we can see the separators being used at the end of each string, and that the Samba server ignores the one byte field describing the type of each string (i.e., the buffer format).

7.2 Keyword Results

We now present the keyword extraction results. Table 4 shows the keywords found in the same HTTP GET request used in the separator's results. An entry with value *Yes* means that the keyword was found at the proper position, while an entry with value *NS* means that the keyword was not found because the server in its default configuration does not support those keywords. Thus, the server in other configurations, for example when loading different modules in Apache, might support those keywords.

The results show no missing supported keywords. They also show some instances, where there is a partial keyword match (shown with the partial keyword in the table entry). This happens because there might be two keywords that start the same. Thus, when the server compares the received keyword, with its set of supported keywords, it will obtain a sequence of true comparisons up to the first difference, and that sequence of true comparisons is output by our module as a partial match. For example, Savant supports the *Accept* and *Accept-Language* fields but not the *Accept-Encoding* or *Accept-Charset fields*. For the unsupported fields, there is a partial match with *Accept*. Note that these partial keywords still mark protocol-dependant data in the message.

Table 5 shows additional keywords, with the number of occurrences in parenthesis, that were found in the HTTP GET query. It includes the HTTP version and another *Keep-Alive* keyword, which is different from the one shown in Table 4. This one is the field data for the *Connection* field in Table 4. We obtained similar results for an IRC login request. An interested reader can find them in Appendix B.3.

8 Related Work

We divide the Related Work into groups dealing with protocol reverse engineering, other work related to protocol message format extraction, and dynamic taint analysis applications.

Protocol Reverse-engineering: Successful protocol reversing projects have so far relied on manual techniques, which are slow and costly [4, 1, 3, 7, 2]. Our work pro-

Keyword	Apache	Savant	Miniweb
GET	Yes	Yes	Yes
Host	Yes	NS	Yes
User-Agent	NS	Yes	NS
Accept	Yes	Yes	NS
Accept-Language	Accept	Yes	NS
Accept-Encoding	Accept-	Accept-	NS
Accept-Charset	Accept-	Accept-	NS
Keep-Alive	NS	NS	NS
Connection	Yes	Yes	Yes

Table 4: Keywords present in a HTTP GET query sent to all three HTTP servers and whether they were properly extracted by our system.

Server	Additional keywords found
Apache	'HTTP/' (1); 'e' (1) ; 'Keep-Alive' (1)
Savant	'HTTP/1.' (1); 'Keep-Alive' (1); ':' (4)
Miniweb	N/A

Table 5: Additional keywords in the HTTP GET query.

vides new automatic techniques that can be used to reduce the cost and time associated with these projects.

Lim et al [24] addressed the problem of automatically extracting the format from files and application data output by a program using binaries. Their approach needs the user to identify the output functions and their corresponding parameters. This information is rarely available. Our approach differs in that we do not require any a priori knowledge about the program, only the program binary.

Reverse engineering a protocol strictly from network traces was first addressed in the Protocol Informatics Project [9] that used sequence alignment algorithms. Recently, Cui et al [17] have also proposed a method to derive the message format strictly from application data. Our approach leverages the availability of a protocol implementation, and monitors the program's processing of the network input.

Additional work on protocol message format: There has been additional work that can be used in the protocol reverse-engineering problem. Kannan et al [22] studied how to extract the application-level structure in application data. Their work can be used to find multiple connections belonging to the same protocol session.

Application dialogue replayers [18, 23, 30], aim to replay an original protocol session involving two hosts, to a third host that did not participate in the original session. These tools need to update user or session dependant information in the protocol session. Thus, they may effectively extract a partial session description.

Ma et al [25] use a network-based approach for automatically identifying traffic that uses the same application-layer protocol, without relying on the port number. Their approach extracts a partial session description from the first 64 bytes of the session data. In addition, protocol analyzers have been widely used in network security [8, 6]. Since many protocols exist and their specification is sometimes complex, there have been languages and parsers proposed for simplifying the specification of network protocols [10, 16, 33].

Dynamic Taint Analysis: Previous work has used dynamic taint analysis to tackle problems such as: exploit detection [13, 15, 28, 35], worm containment [14, 34], signature generation [29], and cross-site scripting detection [36]. We propose to use dynamic taint analysis as a technique to understand how a program processes the received application data.

9 Discussion

In this section we discuss the limitations of our approach and how we plan to address them in the future.

Input messages: One fundamental limitation is that we can only obtain the format from the messages given to our analysis. If some messages never appear, we know nothing about them. We plan to incorporate static analysis to our dynamic analysis techniques to deal with this limitation.

Other field format attributes: Currently, our field description only captures a few field attributes. Other attributes such as the field data type (e.g., integer/string), or the field encoding (e.g., big-endian / litte-endian or ASCII / EBDIC / Unicode) are currently not extracted. Also, some field properties such as whether a field is floating, that is, if it can appear in any order in the field sequence, that require analyzing multiple message formats simultaneoulsy are left up to the analyst. Finally, our analysis works on byte granularity. Thus, currently we are not able to distinguish fields shorter than one byte, such as flags fields.

Field semantics: Our system provides limited description about how fields are used. We identify direction fields but we do not identify other field uses such as timestamps, checksums or addresses. We expect to be able to extract more complete semantic information about fields by using symbolic execution to understand the way the program handles them.

Message boundaries: In this paper, we have focus on finding the field boundaries in a message, assuming a synchronous protocol, which helps us to identify the message boundaries. But sessions can contain multiple messages, so we need to identify the message boundaries as well. We hope to address the problem of identifying message boundaries in future work.

Robustness against obfuscation: Although we have tried to keep our analysis as general as possible, currently, our techniques are not fully resistant against obfuscation. Thus, a protocol architect determined to hide her protocol message format might be able to do so. We plan to study techniques more robust against obfuscation in future work.

10 Conclusion

In this paper we have proposed a new approach for protocol reverse engineering by using dynamic analysis of program binaries implementing a protocol. Compared to previous work that uses only network traces [17], our approach can extract more accurate information because the program binary contains richer protocol semantics.

Our approach, *shadowing*, is based on the intuition that the way that an implementation of the protocol processes the received application data reveals a wealth of data about the protocol message format. To extract the message format from the different messages that comprise a protocol we have developed new techniques for identifying difficult-to-find protocol elements such as direction fields, separators, multi-byte fixed-length fields and keywords. Our techniques are more general than previously proposed ones, and allow us to extract more refined message formats.

We have implemented our approach in a system called *Polyglot* and evaluated it over real world implementations of five different protocols: DNS, HTTP, IRC, Samba and ICQ. Our results show accurate message format with minimal differences compared to the manually crafted formats in a state-of-the-art protocol analyzer. The minimal differences we find are usually due to different implementations handling fields in different ways. Finding such differences between implementations is an added benefit, as they are important for problems such as fingerprint generation, fuzzing, and error detection.

11 Acknowledgements

We would like to thank all members of the BitBlaze group for helpful discussions. We would also like to thank Christopher Kruegel, Weidong Cui, and the anonymous reviewers for their help to improve this paper.

References

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

[2] Icqlib: The ICQ Library. http://kicq.sourceforge.net/icqlib.shtml.

[3] Libyahoo2: A C Library for Yahoo! Messenger. http://libyahoo2.sourceforge.net.

[4] MSN Messenger Protocol. http://www.hypothetic.org/docs/msn/index.php.

[5] Qemu: Open Source Processor Emulator. http://fabrice.bellard.free.fr/qemu/.

[6] Tcpdump. http://www.tcpdump.org/.

[7] The UnOfficial AIM/OSCAR Protocol Specification. http://www.oilcan.org/oscar/.

[8] Wireshark, Network Protocol Analyzer. http://www.wireshark.org.

[9] M. A. Beddoe. Network Protocol Analysis Using Bioinformatics Algorithms. http://www.baselineresearch.net/PI/.

[10] N. Borisov, D. J. Brumley, H. J. Wang, and C. Guo. Generic Application-Level Protocol Analyzer and Its Language. *Network and Distributed System Security Symposium,* San Diego, CA, February 2007.

[11] 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. *USENIX Security Symposium,* Boston, MA, August 2007.

[12] J. Caballero, S. Venkataraman, P. Poosankam, M. G. Kang, D. Song, and A. Blum. FiG: Automatic Fingerprint Generation. *Network and Distributed System Security Symposium,* San Diego, CA, February 2007.

[13] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding Data Lifetime Via Whole System Simulation. *USENIX Security Symposium,* San Diego, CA, August 2004.

[14] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-End Containment of Internet Worms. *Symposium on Operating Systems Principles,* Brighton, United Kingdom, October 2005.

[15] J. R. Crandall, S. F. Wu, and F. T. Chong. Minos: Architectural Support for Protecting Control Data. *ACM Transactions on Architecture and Code Optimization,* December 2006.

[16] D. Crocker and P. Overell. Augmented BNF for Syntax Specifications: ABNF. RFC 4234 (Draft Standard), 4234, October 2005.

[17] W. Cui, J. Kannan, and H. J. Wang. Discoverer: Automatic Protocol Description Generation from Network Traces. *USENIX Security Symposium,* Boston, MA, August 2007.

[18] W. Cui, V. Paxson, N. C. Weaver, and R. H. Katz. Protocol-Independent Adaptive Replay of Application Dialog. *Network and Distributed System Security Symposium,* San Diego, CA, February 2006.

[19] H. Dreger, A. Feldmann, M. Mai, V. Paxson, and R. Sommer. Dynamic Application-Layer Protocol Analysis for Network Intrusion Detection. *USENIX Security Symposium,* Vancouver, Canada, July 2006.

[20] C. D. Grosso, G. Antoniol, M. D. Penta, P. Galinier, and E. Merlo. Improving Network Applications Security: A New Heuristic to Generate Stress Testing

Data. *Genetic and Evolutionary Computation Conference,* June 2005.

[21] P. Haffner, S. Sen, O. Spatscheck, and D. Wang. ACAS: Automated Construction of Application Signatures. *ACM SIGCOMM, Workshop on Mining network data,* Philadelphia, PA, October 2005.

[22] J. Kannan, J. Jung, V. Paxson, and C. E. Koksal. Semi-Automated Discovery of Application Session Structure. *Internet Measurement Conference,* Rio de Janeiro, Brazil, October 2006.

[23] C. Leita, K. Mermoud, and M. Dacier. ScriptGen: An Automated Script Generation Tool for Honeyd. *Annual Computer Security Applications Conference,* Tucson, AZ, December 2005.

[24] J. Lim, T. Reps, and B. Liblit. Extracting Output Formats from Executables. *Working Conference on Reverse Engineering,* Benevento, Italy, October 2006.

[25] J. Ma, K. Levchenko, C. Kreibich, S. Savage, and G. M. Voelker. Unexpected Means of Protocol Inference. *Internet Measurement Conference,* Rio de Janeiro, Brazil, October 2006.

[26] P. McMinn, M. Harman, D. Binkley, and P. Tonella. The Species Per Path Approach to SearchBased Test Data Generation. *International Symposium on Software Testing and Analysis,* July 2006.

[27] P. V. Mockapetris. Domain Names - Implementation and Specification. RFC 1035 (Standard), IETF Request for Comments 1035, November 1987.

[28] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. *Network and Distributed System Security Symposium,* San Diego, CA, February 2005.

[29] J. Newsome, D. Brumley, and D. Song. Vulnerability-Specific Execution Filtering for Exploit Prevention on Commodity Software. *Network and Distributed System Security Symposium*, San Diego, CA, February 2006.

[30] J. Newsome, D. Brumley, J. Franklin, and D. Song. Replayer: Automatic Protocol Replay By Binary Analysis. *ACM Conference on Computer and Communications Security,* Alexandria, VA, October 2006.

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

[32] R. Pang, M. Allman, M. Bennett, J. Lee, V. Paxson, and B. Tierney. A First Look At Modern Enterprise Traffic. *Internet Measurement Conference*, Berkeley, CA, October 2005.

[33] R. Pang, V. Paxson, R. Sommer, and L. Peterson. Binpac: A Yacc for Writing Application Protocol Parsers. *Internet Measurement Conference*, Rio de Janeiro, Brazil, October 2006.

[34] G. Portokalidis, A. Slowinska, and H. Bos. Argos: An Emulator for Fingerprinting Zero-Day Attacks for Advertised Honeypots with Automatic Signature Generation. *ACM SIGOPS Operating Systems Review,* 40(4), October 2006.

[35] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure Program Execution Via Dynamic Information Flow Tracking. *International Conference on Architectural Support for Programming Languages and Operating Systems,* Boston, MA, October 2004.

[36] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. *Network and Distributed System Security Symposium*, San Diego, CA, February 2007.

[37] H. Yin, D. Song, E. Manuel, C. Kruegel, and E. Kirda. Panorama: Capturing System-Wide Information Flow for Malware Detection and Analysis. *ACM Conference on Computer and Communications Security*, Alexandria, VA, October 2007.

A Monitoring the Execution

We implement the execution monitor using an emulator [5]. Executing a program inside an emulator allows us to monitor the internal execution of the program and the input/output operations it performs. The emulator has been enhanced to support *dynamic taint analysis* [13, 14, 15, 28, 34, 35]. In dynamic taint analyisis, input data that is considered important is marked (i.e., tainted) when it is received and any instruction that operates on some of the tainted data (e.g., moving it to another location or performing an arithmetic or logical operation), propagates the taint information to the destination operand.

For our purposes, we taint any information received by the program from the network. Specifically, each input byte is assigned a *taint record* that contains a flag to indicate that it came from the network, some connection identifier, and the position of that byte in the application data. Each location, including memory, register and disk is assigned a *shadow memory*, which is used to store the taint records of the input bytes that the data in that location was generated from. As the program moves the input tainted data to new locations, and performs operations on it, the shadow memory for the destination location is updated with the taint records of the input bytes it was generated from. This taint propagation includes all instructions performed by the program or any library that the program uses, including system libraries and dynamically loaded libraries such as dll's.

In addition to the taint data, the emulator also collects information about each instruction, including the content of the operands at the time the instruction is executed. This data is written into the execution trace, which contains all instructions executed by the program, the data they operated on, and the associated taint information.

B Additional Results

B.1 ICQ Login

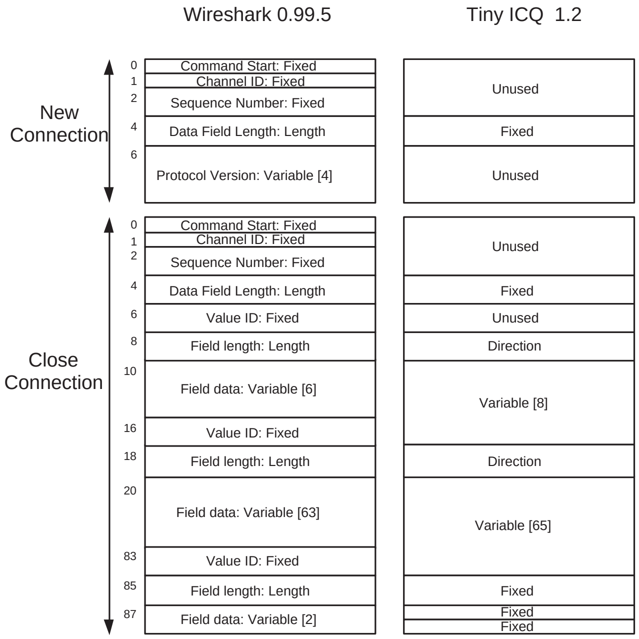

Figure 7: Message format extracted from an ICQ login session compared to the one from Wireshark.

B.2 DNS Query with Pointer

B.3 IRC keywords

Tables 6 and 7 show the results from the keyword extraction module for an IRC login request. Note that the *PONG* keyword was not present in the JoinMe trace, because it is a reply to a *PING* request that is only sent based on the server's configuration.

Keyword	Beware	JoinMe	Unreal
NICK	Yes	Yes	Yes
USER	Yes	Yes	Yes
PONG	Yes	N/A	Yes

Table 6: Keywords present in a IRC login request and whether they were properly extracted by our system.

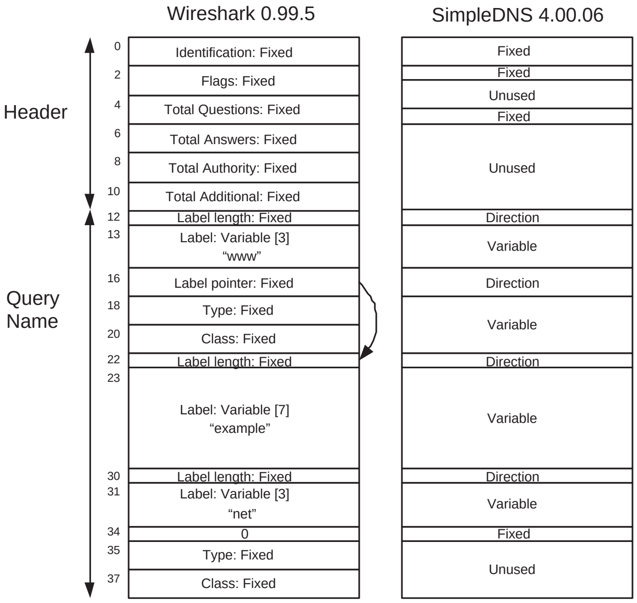

Figure 8: Message format extracted from an DNS query containing a pointer field. This type of query is only supported by the SimpleDNS server.

Server	Additional keywords found
Beware	':' (1); ':1301071548' (1)
JoinMe	' ' (2); ' :' (1)
Unreal	'e' (1); ':' (1); ':A' (1); 'PROTOCTL' (1); 'NAMESX' (1)

Table 7: Additional keywords found in the IRC login request.