Bidirectional Protocol Reverse Engineering: Message Format Extraction and Field Semantics Inference

Juan Caballero
Pongsin Poosankam
Christian Kreibich
Dawn Song

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2009-57
http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-57.html

May 5, 2009

Bidirectional Protocol Reverse Engineering: Message Format Extraction and Field Semantics Inference

Juan Caballero[†*], Pongsin Poosankam[†*], Christian Kreibich[‡], Dawn Song[*]
[*]UC Berkeley [†]CMU [‡]ICSI

Abstract

Automatic protocol reverse-engineering is important for many security applications, including the analysis and defense against botnets. Understanding such C&C protocols is crucial for anticipating a botnet's repertoire of nefarious activity and to enable active botnet infiltration. Frequently, messages sent and received by a bot have to be rewritten in order to contain malicious activity and to provide the botmaster with an illusion of successful and unhampered operation. To enable such rewriting, we need detailed information about the intent and structure of the messages in *both directions* of the communication despite the fact that we generally only have access to the implementation of one endpoint, namely the bot binary. Current techniques cannot enable such rewriting. In this paper, we propose techniques to extract the format of the protocol messages *sent* by an application that implements a protocol specification, and to infer the field semantics for messages both *sent* and *received* by the application. Our techniques enable applications such as rewriting the C&C messages for active botnet infiltration. We implement our techniques into Dispatcher, a tool to extract the message format and field semantics of both received and sent messages. We use Dispatcher to analyze MegaD, a prevalent spam botnet employing a hitherto undocumented C&C protocol, and show that the protocol information extracted by Dispatcher can be used to rewrite the messages sent upstream to the botmaster.

1 Introduction

Automatic protocol reverse-engineering techniques enable extracting the protocol specification of unknown or undocumented application-level protocols [19, 22, 25, 26, 35, 36, 38, 48]. A detailed protocol specification can enhance many security applications such as fuzzing [22], application fingerprinting [18], deep packet inspection [29], or signature-based filtering [27].

One important application for automatic protocol reverse engineering is the analysis and infiltration of botnets. Botnets, large networks of infected computers under control of an attacker, are one of the dominant threats in the Internet today. They enable a wide variety of abusive or fraudulent activities, such as spamming, phishing, click-fraud, and distributed denial-of-service (DDoS) attacks [12, 28, 32]. At the heart of a botnet is its C&C protocol, which enables a bot to locate relevant rendezvous points in the network and provides the botmaster with a means to coordinate malicious activity in the bot population. Automatic protocol reverse-engineering can be used for understanding the C&C protocol used by a botnet, revealing a wealth of information about the capabilities of its bots and the overall intent of the botnet.

In addition to understanding its C&C protocol, an analyst may also be interested in interacting actively with the botnet. Previous work analyzed the economics of the Storm botnet by rewriting the commands sent to the bots [33]. Other times, an analyst may want to rewrite messages sent upstream by the bots, such as when a site's containment policy requires the analyst to make bots lie about their capabilities and achievements. For example, the analyst may want to rewrite a capability report sent by the bot to make the botmaster believe that the bot can send email while all the outgoing SMTP connections by the bot are blocked, or that the bot is connected to the Internet using a high-speed LAN when in reality it is funneling traffic through a low-throughput connection.

To successfully rewrite a C&C message, an analyst first needs to understand the goal of the message, its field structure, and the location of fields carrying relevant information to rewrite. While older botnets build their C&C protocol on top of IRC, many newer botnets use customized or proprietary protocols [2, 20, 31].

Analyzing such C&C protocols is challenging. Manual protocol reverse-engineering of such protocols is time-consuming and error-prone. Furthermore, previous automatic protocol reverse engineering techniques have limi-

tations that prevent them from enabling rewriting of such protocols. Techniques that use network traffic as input [25, 26, 35, 36] are easily hampered by obfuscation or encryption. Techniques that rely on observing how a communication end point (client or server) processes a received input [19, 22, 38, 48] present two major limitations. First, given a program they can only extract information about one side of the dialog, i.e., the *received* messages [22, 38]. To obtain a complete understanding of the protocol they require access to both sides' implementation of the dialog. Unfortunately, when studying a botnet analysts often have access only to the bot side of the communication. This is true for other applications such as instant-messaging solutions where the clients are freely available but the servers are not. Second, current binary-based techniques do not address extracting the semantic information from the protocol messages. Semantic information is fundamental for understanding the intent of a message, and therefore to identify what parts of a dialog to rewrite. For text-based protocols an analyst can sometimes infer such information from the content of the messages, but with binary-based protocols such approach is often not possible.

In this paper we present novel techniques to extract the message format for messages *sent* by an application, which enable extracting the protocol message format from just one side of the communication. New techniques are needed because current techniques to extract the message format of *received* messages rely on tainting the network input and monitoring how the tainted data is used by the program. Most data in sent messages does not come from the tainted network input. Instead, we use the following intuition: programs store fields in memory buffers and construct the messages to be sent by combining those buffers together. Thus, the structure of the buffer holding the sent message represents the inverse of the structure of the sent message. We also present novel techniques to infer the field semantics in messages *sent* and *received* by an application. Our type-inference-based techniques leverage the rich semantic information that is already available in the program by monitoring how data in the received messages is used at places where the semantics are known, and how the sent messages are built from data with known semantics. In addition, we propose modifications to a recently proposed technique to identify the buffers holding the unencrypted *received* message [47], so that it also identifies the buffers holding the unencrypted *sent* message.

We implement our techniques into Dispatcher, a tool to extract the message format and field semantics of both received and sent messages. We use Dispatcher to analyze the C&C protocol used by MegaD, one of the most prevalent spam botnets in use today [9]. To the best of our knowledge, MegaD's proprietary, encrypted, binary C&C protocol has not been previously published and thus presented an ideal test case for our system. We show that the C&C information extracted by Dispatcher can be used to rewrite the MegaD C&C messages. In addition, we use four open protocols: HTTP, FTP, ICQ, and DNS to compare the message format automatically extracted by Dispatcher with the one extracted by Wireshark [14], a state-of-the-art protocol parser that contains manually written protocol grammars.

In summary our contributions are the following:

- We propose novel techniques to extract the format of the protocol messages *sent* by an application that implements a protocol specification. Previous work could only extract the format of the *received* messages. Our techniques enable extracting the complete protocol format even when only one side of the communication is available.
- We present techniques to infer the field semantics for messages *sent* and *received* by an application. Our type-inference-based techniques leverage the wealth of semantic information available in the program.
- We design and develop Dispatcher, a tool that implements our techniques and automatically extracts the message format and associated semantics from both sides of a protocol. We use Dispatcher to analyze MegaD, a prevalent spam botnet, which uses an encrypted binary C&C protocol previously not understood.
- We show that the protocol information that Dispatcher extracts can be used to rewrite the responses that a MegaD bot sends to the commands received from the botmaster, therefore enabling active botnet infiltration.

2 Overview & Problem Definition

In this section we define the problems addressed in the paper and give an overview of our approach.

Scope. The goal of automatic protocol reverse-engineering is to extract the *protocol format*, which captures the structure of all messages that comprise the protocol [19, 25, 26, 35, 38, 48], and the *protocol state machine*, which captures the sequences of messages that represent valid sessions of the protocol [22, 36]. Extracting

the protocol format usually comprises two steps. First, given a set of input protocol messages extract the *message format* of each message. Second, given the set of message formats, identify optional, repetitive and alternative fields, and infer the protocol format, which encompasses the multiple message types that comprise the protocol. The protocol format can be represented as a regular expression [48] or a BNF grammar [27].

This paper deals only with the first step of the protocol format extraction, extracting the message format for a given message, which is a pre-requisite for extracting both the protocol format and the protocol state-machine.

Message format. The message format is captured in the *message field tree*, a hierarchical tree in which each node represents a field in the message[1]. A child node represents a subfield of its parent, and thus corresponds to a subrange of the parent field in the message. The root node represents the complete message, the internal nodes represent *composed* fields[2] and the leaf nodes represent the smallest semantic units in the message[3]. Each node contains an attribute list, where each attribute captures properties about the field such as the field range (the start and end positions in the given message), or whether the field has fixed-length or variable-length, as well as inter-field dependencies such as a field representing the length of another target field or being a checksum of multiple target fields in the tree. Figure 1 shows the message field tree for a C&C message used by MegaD to communicate back to the C&C server information about the bot's host. The root node represents the message, which is 58 bytes long. There are two composed fields: the payload, which is the encrypted part of the message, and the host information, which contains leaf fields representing data about the host such as the CPU identifier and the IP address. The attributes capture that the *MSG_length* field is the length of the payload and the *Length* field is the length of the *Host info* field.

Field semantics. One important property of a field is its semantics, i.e, the type of data that the field contains. Typical field semantics are lengths, timestamps, checksums, hostnames, and filenames. Inferring the field semantics is fundamental to understand what a message does and to identify interesting parts of a dialog to rewrite. The field semantics are captured in the message field tree as an attribute for each field and can be used to label the fields. For example, in Figure 1 the semantics inference states that the range [54:57] contains an IP address and

[1]Called protocol field tree in [38].

[2]Called complex fields in [48], and hierarchical fields in [38].

[3]Called finest-grained fields in [38].

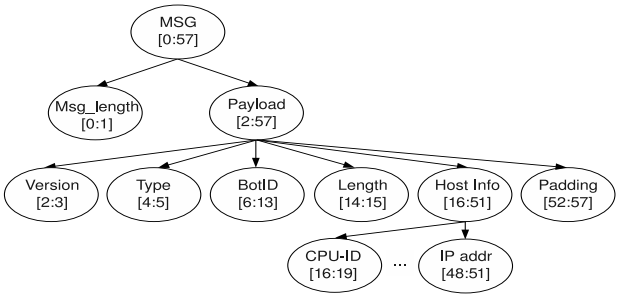

Figure 1: Message field tree for the MegaD Host-Information message.

range [6:13] contains some data previously received over the network. We use this information to label the corresponding fields *BotID* and *IP addr*.

Problem definition. In this paper we address two problems: 1) extracting the message field tree for the messages *sent* by the application, and 2) inferring field semantics, that is, annotating the nodes in the message field tree, for both *received* and *sent* messages, with a field semantics attribute.

Approach. We define the *output buffer* to be the buffer that contains the message about to be sent at the time that the function that sends data over the network is invoked. As a special case, for encrypted protocols, the output buffer is the buffer that contains the unencrypted data at the time the encryption routine is invoked. To extract the message format for *sent* messages we use the following intuition: programs store fields in memory buffers and construct the messages to be sent by combining those buffers together. Thus, the structure of the output buffer represents the inverse of the structure of the sent message. We propose *buffer deconstruction*, a technique to build the message field tree of a sent message by analyzing how the *output buffer* is constructed from other memory buffers in the program. We present our message format extraction techniques for sent messages in Section 4 and our handling of encrypted protocols in Section 5.

To infer the field semantics, we use type-inference-based techniques that leverage the observation that many functions and instructions used by programs contain known semantic information that can be leveraged for field semantics inference. When a field in the received message propagates to the parameters of those functions or instructions (i.e., semantic sinks), we can infer its semantics. When the output of those functions or instructions (i.e., semantic sources) propagates to some field in the output buffer, we can infer its semantics.

We have developed Dispatcher, a tool that enables analyzing both sides of the communication of an unknown protocol, even when an analyst has access only to the application on one side of the dialog. Dispatcher integrates previously proposed techniques to extract the message format of received messages [19, 38, 48], as well as our novel techniques to extract the message format of sent messages, and to infer field semantics in both received and sent messages. We show that the information extracted by Dispatcher enables rewriting MegaD's C&C messages.

Obtaining an execution trace. The input to our message format extraction and field semantics inference techniques are execution traces taken by monitoring the program while it is involved in some network dialog using the unknown protocol. To monitor the program we use a custom analysis environment, which implements dynamic taint tracking [21, 23, 40, 45] and produces instruction-level execution traces that contain all instructions executed, the content of the operands and the associated taint information. To analyze the protocol used by malware samples (e.g., the C&C protocol of a botnet), we need to run the malware sample in a specialized analysis network with custom containment policies [3, 46].

An execution trace contains the processing of multiple messages sent and received by the program during the network dialog. We split the execution trace for the dialog into smaller execution traces for the individual messages by monitoring the program's use of networking functions that read or write data from sockets. We split the execution trace into two traces every time that the program makes a successful call to write data to a socket (e.g., *send*) and every time that the program makes a successful call to read data from a socket (e.g., *recv*), except if the parameter defining the maximum number of bytes to read is tainted. In this case, the read data is considered part of the previous message and the trace is not split. This handles the case when a program first reads the length of the payload and then reads the variable-length payload using the received length value.

Handling obfuscation. The MegaD binary we analyze uses obfuscation techniques such as binary packing and inlining unnecesary instructions, which are designed to thwart static analysis. But, as far as we can tell, it does not implement techniques designed to thwart dynamic analysis such as detecting virtualized or emulated environments. Thus, our techniques run fine on MegaD. However, we expect malware to adapt and have designed our techniques to capture fundamental properties so that they are as resilient as possible to obfuscation. Nevertheless, the techniques proposed in this paper are not specific to malware analysis and can be used to analyze any unknown or undocumented protocols.

3 Field Semantics Inference

In this section we present our technique to identify the field semantics of both received and sent messages[4].

The intuition behind our type-inference-based techniques is that many functions and instructions used by programs contain rich semantic information. We can leverage the existing semantic information to infer field semantics by monitoring if the received network data is used at a point where the semantics are known, or if the data to be sent on the network has been derived from some data with known semantics. Such *inference* is very general and can be used to identify a broad spectrum of field semantics including timestamps, filenames, hostnames, ports, IP addresses, and many other. The semantic information of those functions and instructions is publicly available in the prototype, which describes the goal of the function or instruction, as well as the semantics of its inputs and outputs. Function prototypes can be found, among others, at the Microsoft Developer Network [10] or the standard C library [7]. For instructions, one can refer to the system's manufacturer's manuals [1, 6].

Techniques. For *received* messages, Dispatcher uses taint propagation to monitor if a sequence of bytes from the received message is used in the *parameters* of some selected function calls and instructions, for which the system has been provided with the function's prototype. The sequence of bytes in the received message can then be associated with the semantics for the argument, as defined in the prototype. For example, when a program calls the *connect* function Dispatcher uses the function's prototype to check if any of the parameters in the stack is tainted. The function's prototype tells us that the first parameter is the socket descriptor, the second one is an address structure that contains the IP address and port of the host to connect to, and the third one is the length of the address structure. If the memory locations that correspond to the IP address to connect to in the address structure are tainted from 4 bytes in the input, then Dispatcher can infer that those 4 bytes in the input message (identified from the offset in the taint information) form a field, which contains an IP address to connect to. Similarly, if the memory locations

[4]Our semantics inference techniques were first published as a technical report [17]. They are more general than simultaneous work that identifies cookies and filenames from execution traces [48], and predate other work that also identifies such fields [27].

Field Semantics	Received	Sent
Cookies	yes	yes
IP addresses	yes	yes
Error codes	no	yes
File data	no	yes
File information	no	yes
Filenames	yes	yes
Hash / Checksum	yes	yes
Hostnames	yes	yes
Host information	no	yes
Keyboard input	no	yes
Keywords	yes	yes
Length	yes	yes
Padding	yes	no
Ports	yes	yes
Registry data	no	yes
Sleep timers	yes	no
Stored data	yes	no
Timestamps	no	yes

Table 1: Field semantics identified by Dispatcher for both received and sent messages.

that correspond to the port to connect to have been derived from 2 bytes in the input message, it can identify the position of the port field in the input message.

For *sent* messages, Dispatcher taints the output of some selected functions and instructions using a unique source identifier and offset pair. For each tainted sequence of bytes in the output buffer, Dispatcher identifies from which taint source the sequence of bytes was derived. The semantics for the taint source (return values) are given by the function's or instruction's prototype, and can be associated to the sequence of bytes. For example, if a program uses the *rdtsc* x86 instruction, the instruction's prototype [6] describes the instruction semantics, in particular that it takes no input and returns a 64-bit output representing the current value of the processors time-stamp counter, which is placed in registers EDX:EAX. Thus, at the time of execution when the program uses *rdtsc*, Dispatcher taints the EDX and EAX registers with some unique source identifier and offset. The source identifier uniquely identifies the taint source to be from *rdtsc*, and the offsets identify each byte in the *rdtsc* stream (offsets 0 through 7 for the first call to *rdtsc*).

A special case of the last technique is the *cookie* inference. A cookie represents data from a received network message that propagates to the output buffer (e.g., session identifiers). Thus, a cookie is simultaneously identified in the received and sent messages.

Implementation. To identify field semantics Dispatcher uses an input set of function and instruction prototypes. By default, Dispatcher includes over one hundred functions and a few instructions for which we have already added the prototypes by searching online repositories. To identify new field semantics and their corresponding functions, we examine the external functions called by the program in the execution trace. Table 1 shows the field semantics that Dispatcher can infer from received and sent messages using the predefined functions. In the table, stored data represents data that the program receives over the network and *writes* to the filesystem or the Windows registry, as opposed to data *read* from the filesystem or the Windows registry. We refer the reader to Appendix B for examples of functions and instructions used to identify each of the field semantics in Table 1.

4 Extracting the message format of sent messages

The message field tree captures the hierarchical field structure of the message as well as the field properties encoded in attributes. To extract the message field tree of a sent message we first reverse-engineer the structure of the output message and output a message field tree with no field attributes. Then, we propose specific techniques to identify the field attributes such as how to identify the field boundary (fixed-length, delimiter, length field) and the keywords present in each field.

A field is a sequence of consecutive bytes in a message with some meaning. A memory buffer is a sequence of consecutive bytes in memory that stores data with some meaning. To reverse-engineer the structure of the output message we cannot use current techniques to extract the message format of *received* messages because they rely on tainting the network input and monitoring how the tainted data is used by the program. Most data in sent messages does not come from the tainted network input. Instead, we use the following intuition: programs store fields in memory buffers and construct the messages to be sent by combining those buffers together. Thus, the structure of the output buffer represents the inverse of the message field tree of the sent message. We propose *buffer deconstruction*, a technique to build the message field tree of a sent message by analyzing how the *output buffer* is constructed from other memory buffers in the program. Figure 2 shows the deconstruction of the output buffer holding the message in Figure 1. Note the similarity between Figure 1 and the upside-down version of Figure 2.

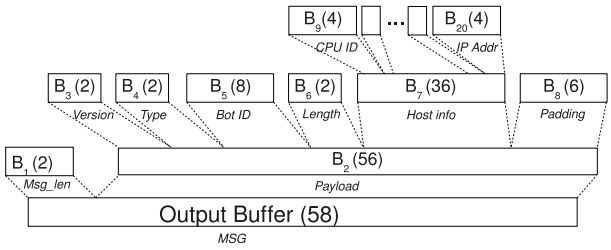

Figure 2: Buffer deconstruction for the MegaD message in Figure 1. Each box is a memory buffer starting at address B_x with the byte length in brackets. Note the similarity with the upside-down version of Figure 1.

Extracting the message format of sent messages is a three-step process. In the *preparation* step, Dispatcher makes a forward pass over the execution trace to extract information about the loops that were executed, the liveness of buffers in the stack, and the callstack information at each point in the execution trace. It also builds an index of the execution trace to enable random access to any instruction. We present the preparation in Section 4.1. The core of the message format extraction is the *buffer deconstruction* step, which is a recursive process in which one memory buffer is deconstructed at a time by extracting the sequence of memory buffers that comprise it. The process is started with the output buffer and recurses until there are no more buffers to deconstruct. Since the structure of the output buffer is the inverse of the message field tree for the sent message, then every memory buffer that forms the output buffer, and recursively the memory buffers that form them, corresponds to a field in the message field tree. For example, deconstructing the output buffer in Figure 2 returns a sequence of two buffers, a 2-byte buffer starting at offset zero in the output buffer (B_1) and a 56-byte buffer starting at offset 2 in the output buffer (B_2). Correspondingly a field with range [0:1] and another one with range [2:57] are added to the no-attributes message field tree. Thus, the buffer deconstruction builds the no-attributes message field tree as it recurses into the output buffer structure. We present the buffer deconstruction in Section 4.2. Finally, the *field attribute inference* identifies length fields, delimiters, keywords, arrays and variable-length fields and adds the information into attributes for the corresponding fields in the message field tree. We present the field attribute inference in Section 4.3.

4.1 Preparation

During the *preparation*, Dispatcher makes a forward pass over the execution trace collecting information needed by the buffer deconstruction as well as the attribute inference.

Loop analysis. During the forward pass, Dispatcher extracts information about each loop present in the execution trace. To identify the loops in the execution trace, Dispatcher supports two different detection methods: static and dynamic. The static method extracts the addresses of the loop head and exit conditions statically from the binary before the forward pass starts, and uses that information during the forward pass to identify the points where any of those loops appears in the trace. The dynamic method does not require any static processing and extracts the loops directly during the forward pass by monitoring instructions that appear multiple times in the same function. Both methods are complimentary. While using static information is more precise at identifying the loop exit conditions, it also requires analyzing all the modules (executable plus dynamically link libraries) used by the application, may miss loops that contain indirection, and cannot be applied if the unpacked binary is not available, such as in the case of MegaD. On the other hand, the dynamic method is less accurate at identifying the loop exit conditions, but requires no setup and can be used in all our samples including MegaD.

Callstack Analysis. During the forward pass, Dispatcher replicates the function stack of the program by monitoring the function calls and returns. The output of the callstack analysis is a function that given an instruction number returns the innermost function that contained that instruction at that point of the execution.

Buffer Liveness Analysis. During the execution trace capture, Dispatcher monitors the heap allocation and free functions used by the program. For each heap allocation it provides the instruction number in the trace, the buffer start and the size of the buffer. For each heap free, it specifies the instruction number in the trace, and the start address of the buffer being freed. During the forward pass, Dispatcher monitors the stack pointer at the function entry and return points, extracting information about which memory locations in the stack are freed when the function returns. This information is used by Dispatcher to determine whether two different writes to the same memory address, correspond to the same memory buffer, since memory locations in the stack (and occasionally in the heap) may be reused for different buffers.

4.2 Buffer Deconstruction

Buffer deconstruction is a recursive process. In each iteration it deconstructs a given memory buffer into the se-

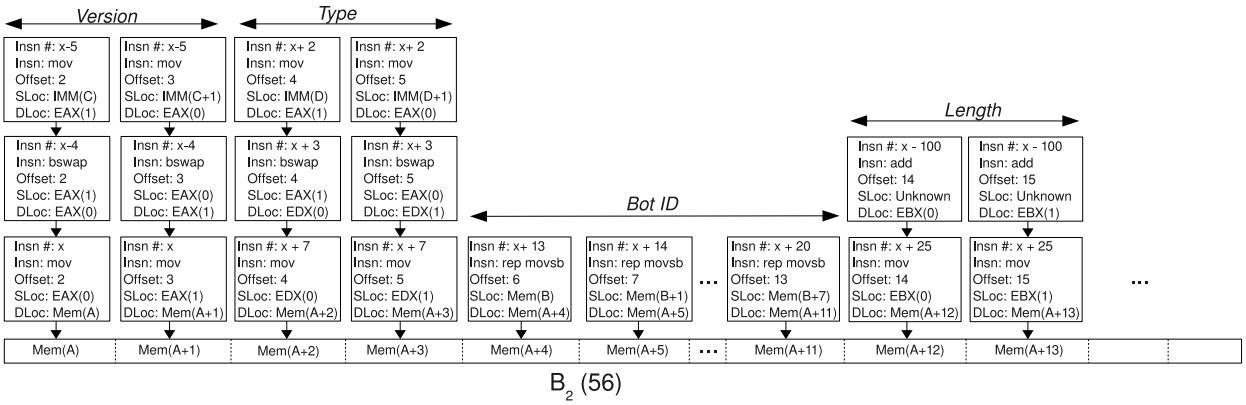

Figure 3: Dependency chain for B_2 in Figure 2. The start address of B_2 is A.

quence of other memory buffers that comprise it. The process starts with the output buffer and recurses until there are no more buffers to deconstruct. It has two parts. First, for each location (i.e., byte) in the given buffer we build a *dependency chain*. Then, using the dependency chains and the information collected in the preparation step, we extract the structure of the given buffer. The input to each buffer deconstruction iteration is a buffer defined by its start address in memory and its length, and the instruction number in the trace where the buffer was last written. The start address and length of the output buffer are obtained from the parameters of the function that sends the data over the network (or the encryption function). The instruction number to start the analysis is the instruction number for the first instruction in the send (or encrypt) function. In the remainder of this section we introduce what locations and dependency chains are and present how they are used to deconstruct the output buffer.

Program locations. We define a *program location* to be a one-byte-long storage unit in the program's state. We consider four types of locations: *memory locations*, *register locations*, *immediate locations*, and *constant locations*, and focus on the address of those locations, rather than on its content. Each memory byte is a memory location indexed by its address. Each byte in a register is a register location, for example, there are 4 locations in EAX: EAX(0) or AL, EAX(1) or AH, EAX(2), and EAX(3). An immediate location corresponds to a byte from an immediate in the code section of some module, indexed by the offset of the byte with respect to the beginning of the module. Constant locations represent the output of some instructions that have constant output. For example, one common instruction is to xor one register against itself (e.g., *xor %eax, %eax*). The output of such instructions is always a zero value in that register. Dispatcher recognizes a number of such instructions and makes each byte of its output a constant location.

Dependency chains. A dependency chain for a program location is the sequence of *write operations* that produced the value of the location at a certain point in the program. A write operation comprises the instruction number at which the write occurred, the location that was written, the source location, and the offset of the written location with respect to the beginning of the output buffer. Figure 3 shows the dependency chains for the B_2 buffer (the one that holds the encrypted payload) in Figure 2. In the figure, each box represents a write operation, and each sequence of vertical boxes represents the dependency chain for one location in the buffer.

The dependency chain is computed in a backwards pass starting at the given instruction number. We stop building the dependency chain at the first write operation for which the source location is: 1) an immediate location, 2) a constant location, 3) a memory location, or 4) an unknown location.

If the source location is part of an immediate or part of the output from some constant output instruction, then there are no more dependencies and the chain is complete. This is the case for the first four bytes of B_2 in Figure 3. The reason to stop at a source memory location is that we want to understand how a memory buffer has been constructed from other memory buffers. After extracting the structure of the given buffer, Dispatcher recurses on the buffers that form it. For example, in Figure 3 the dependency chains for locations $A+4$ through $A+11$ contains only one write operation because the source of the write operation is another memory location. When building the dependency chains, Dispatcher only handles

a small subset of x86 instructions which simply move data around, without modifying it. This subset includes move instructions (*mov,movs*), move with zero-extend instructions (*movz*), push and pop instructions, string stores (*stos*), plus instructions that are used to convert data from network to host order and vice versa such as exchange instructions (*xchg*), swap instructions (*bswap*), or right shifts that shift entire bytes (e.g., *shr $0x8,%eax*). When a write operation is performed by any other instruction, the source is considered unknown and the dependency chain stops. Often, it is enough to stop the dependency chain at such instructions, because the program is at that point performing some operation on the field (e.g., arithmetic operation) as opposed to just moving the content around. Since programs operate on leaf fields, not on composed fields, then at that point of the chain we have already recursed up to the corresponding leaf field in the message field tree. For example, in Figure 3 the dependency chains for the last two bytes stop at the same *add* instruction. Thus, both source locations are unknown. Note that those locations correspond to the length field in Figure 1. The fact that the program is increasing the length value indicates that the dependency chain has already reached a leaf field.

Extracting the buffer structure. For a buffer location, we call the source location of the last element in its dependency chain, the *source* of the buffer location. We say that two source locations belong to the same source buffer if they are contiguous memory locations (in either ascending or descending order) and the liveness information states that none of those locations has been freed between their corresponding write operations. If the source locations are not in memory (e.g., register, immediate, constant or unknown location), they belong to the same buffer if they were written by the same instruction (i.e, same instruction number).

For example, in Figure 3 the source locations for memory locations $A+4$ and $A+5$ are contiguous (*Mem(B)* and *Mem(B+1)*). The source locations for memory locations A and $A+1$ are also contiguous (*IMM(C)* and *IMM(C+1)*) because they are written by the same instruction ($x-5$).

To extract the structure for the given buffer (B_2) Dispatcher iterates on the buffer locations from the buffer start (*Mem(A)*) to the end (*Mem(A+55)*). For each buffer location, Dispatcher checks whether the source of the current buffer location belongs to the same source buffer as the source of the previous buffer location. If they do not, then it has found a boundary in the structure of the buffer. The structure of the given buffer is output as a sequence of ranges that form it, where each range states whether it corresponds to a source memory buffer. Dispatcher finds 6 ranges in B_2. The last two ranges correspond to the Host Info field and the padding in Figure 1. The other four are shown in Figure 3. They are marked with arrows at the top of the figure. Since only the third range originates from another memory buffer, that is the only buffer that Dispatcher will recurse on to reconstruct.

Attribute	**Value**
Field Range	Start offset and length in message
Field Boundary	Fixed, Length, Delimiter
Field Semantics	A value from Table 1
Field Keywords	List of keywords in field

Table 2: Field attributes used in the message field tree.

Once the buffer structure has been extracted, Dispatcher uses the correspondence between buffers and fields in the analyzed message, adding one field to the message field tree per range in the buffer structure using the offsets relative to the output buffer. In Figure 3 it adds four new fields that correspond to the *Version*, *Type*, *Bot ID*, and *Length* in Figure 1.

4.3 Field Attributes Inference

The message field tree built during the buffer deconstruction step represents the hierarchical structure of the output message but it does not contain information about inter-field relationships such as if a field represents the length of another target field. Such additional information is captured by the field attributes in the message field tree.

Table 2 presents the field attributes that we identify in this paper. The field range captures the position of the field in the message. The field boundary captures how an application determines where the field ends. Fields can be fixed-length (*Fixed*), variable-length using a length field (*Length*). or variable-length using a delimiter (*Delimiter*)[5]. The field semantics are the values in Table 1. The field keywords attribute contains a list of all the protocol constants that appear in the field and their position.

The field attributes in Table 2 are similar to the ones that previous work extracts for received messages [19,48]. But, previous techniques that work on received messages do not work on sent messages because they rely on monitoring how the data received over the network is processed, when for sent messages we can only observe how the sent messages are built. Our techniques are new, but share common intuitions with previous techniques because they both try to capture the fundamental properties

[5]Also called separator in [19].

of the different protocol elements. In fact, some attribute values are more difficult to extract for sent messages than for received messages. For example, many fields that a protocol specification would define as variable-length may encode some fixed-length data in a specific implementation. For example the *Server* header is variable-length based on the HTTP specification. However a given HTTP server implementation may have hardcoded the *Server* string in the binary. Thus, for the implementation the field is fixed-length and it becomes difficult to infer otherwise. Leveraging the availability of multiple implementations of the same protocol could help in such cases. We plan to study this in future work.

Keywords. Keywords are constants that appear in network messages. To identify constants in the output buffer, Dispatcher taints the memory region that contains the module (and DLL's shipped with the main binary) with a specific taint origin, effectively tainting both immediates in the code section as well as data stored in the data section. Locations in the output buffer tainted from this origin are considered keywords.

Length fields. Dispatcher uses three different techniques to identify length fields. The intuition behind the techniques is that length fields can be computed by either incrementing a counter as the program iterates on the field, or by subtracting pointers to the beginning and end of the buffer. The intuition behind the first two techniques is that those arithmetic operations translate into an unknown source at the end of the dependency chains for the buffer locations corresponding to the length field. When a dependency chain ends in an unknown source, Dispatcher checks whether the instruction that performs the write is inside a known function the computes the length of a string (e.g., *strlen*), or it is a subtraction where the operands are pointers to the beginning and end of the buffer. The third technique tries to identify counter increments that do not correspond to well-known string length functions. For each buffer it uses the loop information to identify if most writes to the buffer[6] belong to the same loop. If they do, then it uses the techniques in [44] to extract the loop induction variables. For each induction variable it computes the dependency chain and checks whether it intersects the dependency chains from any output buffer locations that precede the locations written in the loop (since a length field always has to precede its target field). Any intersecting location is part of the length field for the field processed in the loop.

[6]Many memory move functions are optimized to move 4 bytes at a time in one loop and use separate instructions or loops to move the remaining bytes.

Delimiters. Delimiters are constants used by protocols to mark the boundary of variable-length fields. Thus, it is difficult to differentiate a delimiter from any another constant in the output message. To identify delimiters, Dispatcher looks for constants that appear multiple times in the same message or appear at the end of multiple messages in the same session (three appearances are required). Constants can be identified by checking the offsets of the taint information for keyword identification. If the delimiters come from the data section, they can also be identified by checking whether the source address of all instances of the constant comes from the same buffer.

Variable-length fields. Dispatcher marks fields that precede a delimiter, and target fields for previously identified length fields as variable-length fields. It also marks as variable-length fields, fields that have been derived from semantic sources that are known to have variable length such as file data. All other fields are marked as fixed-length.

Arrays. The intuition behind identifying arrays of records is that they are written in loops, one record at a time. Thus, Dispatcher uses the loop information extracted during preparation to identify loops that write multiple consecutive fields. Then, it adds to the message field tree one *Array* field with the range being the combined range of all the consecutive fields written in the loop, and one *Record* field per range of bytes written in each iteration of the loop.

5 Handling encrypted messages

Our protocol reverse engineering techniques, as well as previous ones, work on unencrypted data. Thus, when reverse-engineering encrypted protocols we need to address two problems: for received messages, we need to identify the buffers holding the unencrypted data at the point that the decryption has finished (since buffers may only hold the decrypted data for a brief period of time). For sent messages, we need to identify the buffers holding the unencrypted data at the point that the encryption is about to begin. Once the buffers holding the unencrypted data have been identified, protocol reverse engineering techniques can be applied on them, rather than on the messages received or about to be sent on the wire.

Recent work has looked at the problem of reverse-engineering the format of received encrypted messages [39, 47]. Since the application needs to decrypt the data before using it, those approaches monitor the application's processing of the encrypted message and attempt

to locate the buffers that contain the decrypted data at the point that the decryption has finished. Those approaches do not address the problem of finding the buffers holding the unencrypted data before it is encrypted, which is also required in our case.

In this work we first tried to extend the technique presented in ReFormat [47] to identify the buffers holding the unencrypted data before the encryption. However, we found that the technique in ReFormat could not identify the buffers holding the decrypted data. The problem is that ReFormat tries to identify a single boundary between the decryption and the normal protocol processing. In MegaD multiple such boundaries exist. As shown in Figure 1 MegaD messages comprise two bytes with the message length, followed by the encrypted payload. After checking the message length, a MegaD bot will decrypt 8 bytes from the encrypted payload and process them, then move to the next 8 bytes and process them, and so on. In addition, some messages in MegaD also use compression and the decryption and decompression operations are interleaved. Thus, there is no single program point where all data in a message is available unencrypted and uncompressed. We believe that a general technique has to identify every *instance* of encryption, hashing, compression, and obfuscation, which we generally term *encoding functions*.

Identifying the encoding functions. To address this limitation and to enable locating the buffers holding the unencrypted data before encryption, we have simplified the technique in ReFormat by removing the cumulative metric, the use of tainted data, and the concept of leaf functions. The technique uses the intuition in [47] that encoding functions contain an inordinate number of arithmetic and bitwise operations. It works as follows. Dispatcher makes a forward pass over the input execution trace replicating the callstack of the application by monitoring the call and return instructions. For each function it computes the ratio between the number of arithmetic and bitwise operations over the total number of instructions in the function. The ratio includes only the instructions that belong to the function. It does not include the instructions in the functions called from inside that function. The ratio is computed for each appearance of the function in the trace.

Any function that executes a minimum number of instructions and has a ratio larger than a pre-defined threshold is flagged by Dispatcher as an instance of a encoding function. In our experiments, the threshold is set to 0.55 and the minimum number of instructions is 20. In our MegaD execution traces, this simple technique identifies all instances of 3 unique functions: the decryption routine, the encryption routine, and a routine that deobfuscates the encryption and decryption keys that are hidden in the binary before calling the encryption or decryption routines. In addition, in the traces that process messages with compressed data, Dispatcher flags a fourth function that corresponds to the *inflate* function in the *zlib* library, which is statically linked into the MegaD binary. Our evaluation results in Section 6.3 show the false positive rate of the technique to be 0.002%.

Identifying the buffers. To identify the buffers holding the unencrypted data before encryption we compute the *read set* for the encryption routine, the set of locations read inside the encryption routine before being written. The read set for the encryption routine includes the buffers holding the unencrypted data, in addition to the encryption key and some hardcoded tables used by the routine. We can differentiate the buffers holding the unencrypted data because their content varies between multiple instances of the same function. To identify the buffers holding the unencrypted data after decryption we compute the *write set* for the decryption routine, the set of locations written inside the decryption routine and read later in the trace.

6 Evaluation

In this section we evaluate our techniques on the MegaD C&C protocol, as well as a number of open protocols.

6.1 Evaluation on MegaD

MegaD uses a proprietary, encrypted, binary protocol previously not understood. Our MegaD evaluation has two parts. First, we describe the information obtained by Dispatcher on the C&C protocol used by MegaD. Then, we show how the information extracted by Dispatcher can be used to rewrite a dialog between the bot and the C&C server.

MegaD C&C Protocol. The MegaD C&C protocol uses TCP as the transport protocol. It uses port 443 (assigned for HTTPS) but the messages are encrypted with a proprietary algorithm rather than using the SSL algorithms. Our network traces show our MegaD bot communicating with three entities: the *C&C server* that the bot periodically probes for new commands; the *SMTP test server*, an SMTP server whose hostname is provided by the C&C server and to which the bot connects to test for spam sending capabilities; and the *spam server*, whose IP address and listening port are sent by the C&C server to the bot

so that the bot can download all spam-related information such as the spam template or the email addresses to spam. The communication with the C&C server and the spam server uses the encrypted C&C protocol, while the communication with the SMTP test server uses unencrypted SMTP. The communication model is pull-based. The bot periodically probes the botmaster by sending a request message. The botmaster replies with two messages: one with authentication information, and the other one with a command. The bot performs the requested action and sends a response to the botmaster with its results.

Message format. Our MegaD C&C traces contain 14 different messages (7 on each direction of the dialog). Using Dispatcher, we have extracted the message field tree for messages on both directions, as well as the associated field semantics. All 14 messages follow the structure shown in Figure 1 with a 2-byte message length followed by an encrypted payload. The payload, once decrypted, contains a 2-byte field that we term version as it is always a keyword of value 0x100, followed by a 2-byte message type field. The structure of the remaining payload depends on the message type. To summarize the protocol format we have used the output of Dispatcher to write a BinPac grammar [41] that comprises all 14 messages. Field semantics are added as comments to the grammar. Appendix A presents an abridged version of the grammar.

To the best of our knowledge, we are the first to document MegaD's C&C protocol in detail. Thus, we lack ground truth to evaluate our grammar. To verify the grammar's accuracy, we use another execution trace that contains a different instance of one of the analyzed dialogs. We dump the content of all unencrypted messages and try to parse the messages using our grammar. For this, we were provided by other researchers with a stand-alone version of the BinPac parser included in Bro [42]. Using our grammar, the parser successfully parses all MegaD C&C messages in the new dialog. In addition, the parser throws an error when given messages that do not follow the MegaD grammar.

Attribute detection. The 14 MegaD messages contain no delimiters or arrays. They contain two variable-length fields that use length fields to mark their boundaries: the compressed spam-related information (i.e., template and addresses) received from the spam server, and the host information field in Figure 1. Both the length fields and variable-length fields are correctly detected by Dispatcher. The only attributes that Dispatcher misses are the message length fields on sent messages because they are computed using complex pointer arithmetic that Dispatcher cannot reason about.

Field semantics. Dispatcher identifies 11 different field semantics over the 14 messages: IP addresses, ports, hostnames, length, sleep timers, error codes, keywords, cookies, stored data, padding and host information. There are only two fields in the MegaD grammar for which Dispatcher does not identify their semantics. Both of them happen in received messages: one of them is the message type, which we identify by looking for fields that are compared against multiple constants in the execution and for which the message format varies depending on its value. The other one corresponds to an integer whose value is checked by the program but apparently not used further. Note that we identify some fields in sent messages as keywords because they come from immediates and constants in the data section. We cannot identify exactly what they represent because we do not see how they are used by the C&C server.

Rewriting a MegaD dialog. To show how our grammar enables live rewriting, we run a live bot inside our analysis environment, which is located in a network that filters all outgoing SMTP connections for containment purposes. In a first dialog, the C&C server sends the command to the bot ordering to test for spam capability using a given Spam test server. The analysis network blocks the SMTP connection causing the bot to send an error message back to the C&C server, to communicate that it cannot send spam. No more spam-related messages are received by the bot. Then, we start a new dialog where at the time the bot calls the encrypt function to encrypt the error message, we stop the execution, rewrite the encryption buffer with the message that indicates success, and let the execution continue[7]. After the rewriting the bot keeps receiving the spam-related messages, including the spam template and the addresses to spam, despite the fact that it cannot send any spam messages. Note that simply replaying the message that indicates success from a previous dialog into the new dialog does not work because the success message includes a cookie value that the C&C selects and that can change between dialogs.

6.2 Evaluation on Open Protocols

In this section we evaluate our techniques on four open protocols: HTTP , DNS, FTP, and ICQ. For this, we compare the output of Dispatcher with the output by Wireshark 1.0.5 [14] when processing 12 messages belonging to those four protocols. For each protocol we select a representative application that implements the protocol:

[7]The size of both messages is the same once padding is accounted for, thus we can reuse the buffer allocated by the bot.

		Wireshark		Dispatcher		Errors			
Protocol	**Message Type**	$\|L_W\|$	$\|C_W\|$	$\|L_D\|$	$\|C_D\|$	$\|E(L_W)\|$	$\|E(L_D)\|$	$\|E(C_W)\|$	$\|E(C_D)\|$
HTTP	GET reply	11	1	22	0	11	1	0	1
	POST reply	11	1	22	0	11	1	0	1
DNS	A reply	27	4	28	0	1	0	0	4
FTP	Welcome0	2	1	3	1	1	0	0	0
	Welcome1	2	1	3	1	1	0	0	0
	Welcome2	2	1	3	1	1	0	0	0
	USER reply	2	1	3	1	1	1	0	0
	PASS reply	2	1	2	0	1	1	0	1
	SYST reply	2	1	2	0	1	1	0	1
ICQ	New connection	5	0	5	0	0	0	0	0
	AIM Sign-on	11	3	15	3	5	0	0	0
	AIM Logon	46	15	46	15	0	0	0	0
Total		123	30	154	22	34	5	0	8

Table 3: Comparison of the message field tree for sent messages extracted by Dispatcher and Wireshark

Apache-2.2.1 for HTTP, Bind-9.6.0 for DNS, Filezilla-0.9.31 for FTP, and Pidgin-2.5.5 for ICQ. Note that regardless of the application being a client (Pidgin) or a server (Bind, Apache, Filezilla), for this part of the evaluation we focus on sent messages.

Message format. Wireshark is a network protocol analyzer, which contains manually-generated grammars (called dissectors) for many network protocols. Although Wireshark is a mature and widely-used tool, its dissectors have been manually generated and therefore are not completely error-free. To compare the accuracy of the message format automatically extracted by Dispatcher with the manually generated ones included in Wireshark, we analyze the message field tree output by both tools and manually compare them to the protocol specification. Thus, we can classify the differences between both tools to be either Dispatcher or Wireshark errors (or both).

We name the set of leaf fields and composed fields in the message field tree output by Wireshark as L_W and C_W respectively. Then, L_D and C_D are the corresponding sets for Dispatcher. Table 3 shows the evaluation results. For each protocol and message it first shows the number of leaf fields and composed fields in the message field tree output by both tools $|L_W|$, $|C_W|$, $|L_D|$, and $|C_D|$. Then, it presents the manual classification of its errors, where $|E(L_W)|$ and $|E(L_D)|$ represent the number of errors on leaf fields in the message field tree output by Wireshark and Dispatcher respectively. Similarly, $|E(C_W)|$ and $|E(C_D)|$ represent the number of errors on composed fields.

The results show that Dispatcher outperforms Wireshark when identifying leaf fields. This surprising result is due to the inconsistencies between the different dissectors in Wireshark when identifying delimiters. Some dissectors do not add the delimiter fields to the message field tree, others concatenate them to the variable-length field for which they mark the boundary, while others treat them as separate fields. After checking the protocol specifications, we believe that delimiters should be treated as their own fields in all dissectors. The results also show that Wireshark outperforms Dispatcher when identifying composed fields. This is due to the program not using loops to write the arrays because the number of elements in the array is known or is small enough that the compiler has unrolled the loops.

Overall, Dispatcher outperformed Wireshark for the given messages. Note that, we do not claim that Dispatcher is generally more accurate than Wireshark since we are only evaluating a limited number of protocols and messages. But, the results show that the accuracy of the message format automatically extracted by Dispatcher can rival the accuracy of the manually generated one used by Wireshark.

Errors on leaf fields. Here we detail the errors on leaf fields that we have assigned to Dispatcher. The error in the HTTP GET reply message is in the *Status-Line*. The HTTP/1.1 specification [30] states that its format is: *Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF*, but both Dispatcher and Wireshark consider the Status-Code, the delimiter, and the Reason-Phrase to belong to the same field. The FTP specification [43] states that a reply message comprises a completion code followed by a text string. The error in the FTP USER reply message is due to the fact that the server echoes back the username to the client and Dispatcher identifies the username being echoed back as an additional

cookie field. The other FTP replies have the same type of error: the response code is merged with the text string because the program keeps the whole message (except the delimiter) in a single buffer in the data section. As mentioned earlier the errors on composed fields are due to the program being analyzed not using loops to write the arrays. This can happen because the number of elements in the array is a priori known or is small enough that the compiler has unrolled the loops. For example in the DNS reply the four errors correspond to the *Queries*, *Answers*, *Authoritative*, and *Additional* sections in the message, which Bind processes separately and therefore cannot be identified by Dispatcher.

These errors highlight the fact that the message field tree extracted by Dispatcher is limited to the quality of the protocol implementation in the binary, and may differ from the protocol specification even when analyzing mature implementations.

Attribute detection. The 12 messages contain 14 length fields, 43 delimiters, 57 variable-length fields, and 3 arrays. Dispatcher misses 8 length fields because their value is hard-coded in the program. Thus, their target variable-length fields are considered fixed-length. Out of the 43 delimiters Dispatcher only misses one, which corresponds to a null byte marking the end of a cookie string that was considered part of the string. Dispatcher correctly identifies all other variable-length fields. Out of 3 arrays, Dispatcher misses one formed by *Queries*, *Answers*, *Authoritative*, and *Additional* sections in the DNS reply, which Bind processes separately and therefore cannot be identified by Dispatcher.

Field semantics. Dispatcher correctly identifies all semantic information in the sent messages, except the 3 pointers in the DNS reply, used by the DNS compression method, which are computed using pointer arithmetic that Dispatcher cannot reason about.

6.3 Detecting Encoding Functions

To evaluate the detection of encoding functions presented in Section 5 we perform the following experiment. We obtain 20 execution traces from multiple programs that handle network data. Five of these traces process encrypted and compressed functions, four of them are from MegaD sessions and the other one is from Apache while handling an HTTPS session. MegaD using its own encryption algorithm and the *zlib* library for compression and Apache uses SSL with AES and SHA-1[8]. The remaining 15 execution traces are from a variety of programs including browsers (Internet Explorer 7, Safari 3.1, and Google Chrome 1.0), network servers (Bind,Atphttpd) and services embedded in Windows (RPC, MSSQL).

Dispatcher flags any function instances in the execution traces with at least 20 instructions and a ratio of arithmetic and bitwise instructions greater than 0.55 as encoding functions. The results are shown in Table 4. The 20 execution traces contain over 3.5 million functions calls from 22,379 unique functions. Dispatcher flags 0.14% of the function instances as encoding functions. We manually classify the unique functions flagged by Dispatcher as true positives or false positives, using the function names and associated debugging information. We conservatively classify all instances of functions flagged by Dispatcher, for which we don't have any information as false positives. Dispatcher correctly identifies all encoding functions in the MegaD and Apache-SSL traces. There are a total 87 false positives from 9 unique functions. Out of those 9 unique functions we have been able to identify two: `memchr` and `comctl32.dll::TrueSaturateBits`. All instances of the other 7 are conservatively classified as false positives. Based on these results, our technique correctly identifies all known encoding functions and has a false positive rate of 0.002%.

7 Related Work

Protocol reverse-engineering projects have existed for a long time to enable interoperability of open solutions with proprietary protocols. Those projects relied on manual techniques, which are slow and costly [4, 5, 8, 11, 13]. Automatic protocol reverse engineering techniques can be used, among other applications, to reduce the cost and time associated with these projects.

Automatic protocol reverse-engineering. Automatic protocol reverse engineering techniques can be divided into those that extract the field structure of a single message [19, 25, 38], those that analyze multiple messages to extract the protocol format [15, 27, 48], and those that infer the protocol state-machine [22, 36]. They can also be classified into techniques that use as input network traffic [15, 25, 36] and techniques that take as input execution traces that capture how a program processes a received input [19, 22, 27, 38, 48].

Techniques that takes as input network data [15, 25, 36] face the issue of limited semantic information in network traces, and cannot address encrypted or obfuscated pro-

[8]TLS-DHE-RSA with AES-CBC-256-SHA-1

Number of traces	Number of functions	True Positives	False Positives	False Positive Rate
20	3,569,773 (22,379)	4,874 (21)	87 (9)	0.002%

Table 4: Evaluation of the detection of encoding functions. Values in parentheses represent the numbers of unique instances. False positives are computed based on manual verification.

tocols. Techniques to extract the message field tree are a prerequisite for techniques that extract the protocol format [27,48] and the protocol state-machine [22] from execution traces. Current approaches that extract the message field tree of a given message have focused on extracting the format of messages *received* by an application. To obtain a complete understanding of the protocol they require access to both sides of the dialog. Our techniques allow to extract the message field tree for *sent* messages, thus enabling the study of both sides of a communication from a single binary.

Lim et al [37] use inter-procedural static analysis to extract the format from files and application data output by a program. Their approach requires the user to input the prototype of the functions that write data to the output buffer. This information is often not available, e.g., when the functions used to write data are not exported by the program. They require sophisticated analysis to deal with indirection and cannot handle packed binaries such as MegaD. Their work does not address semantics inference. Our approach differs in that we do not require any a priori knowledge about the program, and we use a dynamic binary analysis approach that can effectively deal with indirection and packed binaries.

State-machine inference. Protocol reverse-engineering also includes inferring the protocol's state-machine. ScriptGen [36] infers the protocol state-machine from network data. Due to the lack of semantics in network data it is difficult for ScriptGen to determine whether two network messages are two instances of the same message type. Prospex [22] addresses this issue by leveraging information extracted during program execution such as the message field tree and the functions called by the program upon message reception.

Replaying network sessions. Previous work has addressed the problem of replaying previously captured network sessions [26, 35, 36]. Such systems perform limited protocol reverse-engineering on network traces only to the extent necessary for replay. Their focus is to identify the dynamic fields, i.e., fields that change value between sessions, such as cookies, length fields or IP addresses.

Identifying application sessions. There has been additional work that can be used in the protocol reverse-engineering problem. Kannan et al [34] 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.

Encoding the protocol information. Previous work has proposed languages to describe protocol specifications [16, 24, 41]. Such languages are useful to store the results from protocol reverse engineering techniques, enabling the construction of generic protocol parsers.

8 Conclusion

Automatic protocol reverse-engineering is important for many security applications, including the analysis and infiltration of botnets. Prior techniques cannot enable rewriting of C&C messages needed for infiltration because they cannot analyze encrypted protocols used by newer botnets, they do not extract information about the semantics of the protocol, or they require access to both peers in a protocol dialog for a complete view of the protocol. In this paper we have addressed those limitations.

We have proposed techniques to extract the message format of *sent* messages. Our techniques leverage the intuition that the structure of the output buffer represents the inverse of the structure of the sent message. Thus, we introduce *buffer deconstruction*, a technique that extracts the structure of a message being sent by reconstructing how the output buffer has been built from other memory buffers in the program. In addition, we have proposed techniques for inferring field semantics, a prerequisite for rewriting C&C messages for botnet infiltration. Our type-inference-based techniques leverage the rich semantic information that is already available in the program by monitoring how data in the received messages is used at places where the semantics are known, and how the sent messages are built from data with known semantics.

We have implemented our techniques as well as previous approaches into Dispatcher, a tool that enables the analysis of protocol dialogs even when only one of the peers involved in the dialog is available. We have used Dispatcher to analyze the previously undocumented C&C protocol of MegaD, a prevalent spam botnet. We have shown that the information output by Dispatcher enables botnet infiltration by rewriting the C&C messages.

9 Acknowledgements

We are grateful to Robin Sommer for providing us with a stand-alone version of BinPac, and to Stephen McCamant for his valuable comments.

This material is based upon work partially supported by the National Science Foundation under Grants No. 0311808, No. 0448452, No. 0627511, and CCF-0424422, and by the Air Force Office of Scientific Research under MURI Grant No. 22178970-4170. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Air Force Office of Scientific Research, or the National Science Foundation.

References

[1] AMD64 architecture tech docs. `http://www.amd.com/us-en/Processors/DevelopWithAMD/0,,30_2252_875_7044,00.html`.

[2] An analysis of Conficker's logic and rendezvous points. `http://mtc.sri.com/Conficker/`.

[3] DETERlab testbed. `http://www.isi.edu/deter/`.

[4] How Samba was written. `http://samba.org/ftp/tridge/misc/french_cafe.txt`.

[5] icqlib: The ICQ library. `http://kicq.sourceforge.net/icqlib.shtml`.

[6] Intel64 and IA-32 architectures software developer's manuals. `http://www.intel.com/products/processor/manuals/`.

[7] The ISO/IEC 9899:1999 C programming language standard. `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf`.

[8] Libyahoo2: A C library for Yahoo! Messenger. `http://libyahoo2.sourceforge.net`.

[9] Marshal8e6 security threats: Email and web threats. `http://www.marshal.com/newsimages/trace/Marshal8e6_TRACE_Report_Jan2009.pdf`.

[10] Microsoft developer network. `http://msdn.microsoft.com`.

[11] MSN messenger protocol. `http://www.hypothetic.org/docs/msn/index.php`.

[12] Spotlight on bots: The world's most un-wanted bots. `http://nortontoday.symantec.com/features/spotlight_on_bots.php`.

[13] The unofficial AIM/OSCAR protocol specification. `http://www.oilcan.org/oscar/`.

[14] Wireshark. `http://www.wireshark.org/`.

[15] M. A. Beddoe. Network protocol analysis using bioinformatics algorithms. http://www.baselineresearch.net/PI/.

[16] N. Borisov, D. J. Brumley, H. J. Wang, and C. Guo. Generic application-level protocol analyzer and its language. In *Network and Distributed System Security Symposium*, San Diego, CA, February 2007.

[17] J. Caballero and D. Song. Rosetta: Extracting protocol semantics using binary analysis with applications to protocol replay and nat rewriting. Technical Report CMU-CyLab-07-014, Cylab, Carnegie Mellon University, October 2007.

[18] J. Caballero, S. Venkataraman, P. Poosankam, M. G. Kang, D. Song, and A. Blum. Fig: Automatic fingerprint generation. In *Network and Distributed System Security Symposium*, San Diego, CA, February 2007.

[19] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *ACM Conference on Computer and Communications Security*, Alexandria, VA, October 2007.

[20] K. Chiang and L. Lloyd. A case study of the Rustock rootkit and spam bot. In *Workshop on Hot Topics in Understanding Botnets*, Cambridge, MA, April 2007.

[21] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *USENIX Security Symposium*, San Diego, CA, August 2004.

[22] P. M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda. Prospex: Protocol specification extraction. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2009.

[23] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of internet worms. In *Symposium on Operating Systems Principles*, Brighton, United Kingdom, October 2005.

[24] D. Crocker and P. Overell. Augmented BNF for syntax specifications: ABNF. RFC 4234 (Draft Standard), October 2005. `http://www.ietf.org/rfc/rfc4234.txt`.

[25] W. Cui, J. Kannan, and H. J. Wang. Discoverer: Automatic protocol description generation from network traces. In *USENIX Security Symposium*, Boston, MA, August 2007.

[26] W. Cui, V. Paxson, N. C. Weaver, and R. H. Katz. Protocol-independent adaptive replay of application dialog. In *USENIX Security Symposium*, Boston, MA, August 2007.

[27] W. Cui, M. Peinado, K. Chen, H. J. Wang, and L. Irun-Briz. Tupni: Automatic reverse engineering of input formats. In *ACM Conference on Computer and Communications Security*, Alexandria, VA, October 2008.

[28] N. Daswani, M. Stoppelman, and the Google Click Quality & Security Teams. The anatomy of clickbot.a. In *Workshop on Hot Topics in Understanding Botnets*, Cambridge, MA, April 2007.

[29] H. Dreger, A. Feldmann, M. Mai, V. Paxson, and R. Sommer. Dynamic application-layer protocol analysis for network intrusion detection. In *USENIX Security Symposium*, Vancouver, Canada, July 2006.

[30] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. RFC 2068 (Proposed Standard), January 1997. Obsoleted by RFC 2616.

[31] J. B. Grizzard, V. Sharma, C. Nunnery, and B. B. Kang. Peer-to-peer botnets: Overview and case study. In *Workshop on Hot Topics in Understanding Botnets*, Cambridge, MA, April 2007.

[32] J. P. John, A. Moshchuk, S. D. Gribble, and A. Krishnamurthy. Studying spamming botnets using Botlab. In *Symposium on Networked System Design and Implementation*, Boston, MA, April 2009.

[33] C. Kanich, C. Kreibich, K. Levchenko, B. Enright, G. M. Voelker, V. Paxson, and S. Savage. Spamalytics: An empirical analysis of spam marketing conversion. In *ACM Conference on Computer and Communications Security*, Alexandria, VA, October 2008.

[34] J. Kannan, J. Jung, V. Paxson, and C. E. Koksal. Semi-automated discovery of application session structure. In *Internet Measurement Conference*, Rio de Janeiro, Brazil, October 2006.

[35] C. Leita, M. Dacier, and F. Massicotte. Automatic handling of protocol dependencies and reaction to 0-day attacks with ScriptGen based honeypots. In *International Symposium on Recent Advances in Intrusion Detection*, Hamburg, Germany, September 2006.

[36] C. Leita, K. Mermoud, and M. Dacier. ScriptGen: An automated script generation tool for honeyd. In *Annual Computer Security Applications Conference*, Tucson, AZ, December 2005.

[37] J. Lim, T. Reps, and B. Liblit. Extracting output formats from executables. In *Working Conference on Reverse Engineering*, Benevento, Italy, October 2006.

[38] Z. Lin, X. Jiang, D. Xu, and X. Zhang. Automatic protocol format reverse engineering through context-aware monitored execution. In *Network and Distributed System Security Symposium*, San Diego, CA, February 2008.

[39] N. Lutz. Towards revealing attacker's intent by automatically decrypting network traffic. Master's thesis, ETH, Zürich, Switzerland, July 2008.

[40] 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*, San Diego, CA, February 2005.

[41] R. Pang, V. Paxson, R. Sommer, and L. Peterson. binpac: A yacc for writing application protocol parsers. In *Internet Measurement Conference*, Rio de Janeiro, Brazil, October 2006.

[42] V. Paxson. Bro: A system for detecting network intruders in real-time. *Computer Networks*, 31(23–24), 1999.

[43] J. Postel and J. Reynolds. File transfer protocol. RFC 959 (Standard), October 1985. Updated by RFCs 2228, 2640, 2773, 3659.

[44] P. Saxena, P. Poosankam, S. McCamant, and D. Song. Loop-extended symbolic execution on binary programs. In *International Symposium on Software Testing and Analysis*, Chicago, IL, July 2009.

[45] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, October 2004.

[46] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M. Voelker, and S. Savage. Scalability, fidelity, and containment in the Potemkin virtual honeyfarm. In *Symposium on Operating Systems Principles*, Brighton, United Kingdom, October 2005.

[47] Z. Wang, X. Jiang, W. Cui, and X. Wang. ReFormat: Automatic reverse engineering of encrypted messages. Technical Report NCSU-TR-2008-26, Norh Carolina State University, December 2008.

[48] G. Wondracek, P. M. Comparetti, C. Kruegel, and E. Kirda. Automatic network protocol analysis. In *Network and Distributed System Security Symposium*, San Diego, CA, February 2008.

A MegaD BinPac grammar

```
type MegaD_Message(is_inbound: bool) = record {
  msg_len : uint16;
  encrypted_payload(is_inbound):
    bytestring &length = 8 * msg_len;
} &byteorder = bigendian;

type encrypted_payload(is_inbound: bool) = record {
  version : uint16; # Constant (0x0100 or 0x0001)
  mtype : uint16;
  data : MegaD_data(is_inbound, mtype);
};

# Message types seen in our traces
type MegaD_data(is_inbound: bool, msg_type: uint16) =
  case msg_type of {
    0x00 -> m00 : msg_0;
    0x01 -> m01 : msg_1;
    0x0e -> m0e : empty_msg;
    0x15 -> m15 : empty_msg;
    0x16 -> m16 : msg_0x16;
    0x18 -> m18 : empty_msg;
    0x1c -> m1c : msg_0x1c(is_inbound);
    0x1d -> m1d : msg_0x1d;
    0x21 -> m21 : msg_0x21;
    0x22 -> m22 : msg_0x22;
    0x23 -> m23 : msg_0x23;
    0x24 -> m24 : msg_0x24;
    0x25 -> m25 : msg_0x25;
    default -> unknown : bytestring &restofdata;
};

# Direction: outbound (CC)
# MegaD supports two submessages for type zero
type msg_0 = record {
  fld_00 : uint8; # <unknown>
  fld_01 : MegaD_msg0(fld_00);
};

type MegaD_msg0(msg0_type: uint8) =
  case msg0_type of {
    0x00 -> m00 : msg_0_init;
    0x01 -> m01 : msg_0_idle;
    default -> unknown : bytestring &restofdata;
};

type msg_0_init = record {
  fld_00 : bytestring &length=16; # Constant(0)
  fld_01 : uint32; # Constant (0xd)
  fld_02 : uint32; # Constant (0x26)
  fld_03 : uint32; # IP address
  pad : bytestring &restofdata;
};

type msg_0_idle = record {
  fld_00 : bytestring &length=8; # Bot ID
  fld_01 : uint32; # Constant(0)
  pad : bytestring &restofdata;
};

# Direction: inbound (CC)
type msg_1 = record {
  fld_00 : bytestring &length=16; # Stored data
  fld_01 : uint32; # Sleep Timer(sec)
  fld_02 : bytestring &length=8; # Bot ID
};

# Direction: inbound (CC)
type empty_msg = record {
  pad : bytestring &restofdata;
};

type host_info = record {
  fld_00 : uint32; # Cpu identifier
  fld_01 : uint32; # Tick difference
  fld_02 : uint32; # Tick counter
  fld_03 : uint16; # OS major version
  fld_04 : uint16; # OS minor version
  fld_05 : uint16; # OS build number
  fld_06 : uint16; # Service pack major
  fld_07 : uint16; # Service pack minor
  fld_08 : uint32; # Physical memory(KB)
  fld_09 : uint32; # Available memory(KB)
  fld_10 : uint16; # Internet conn. type
  fld_11 : uint32; # IP address
};

# Direction: outbound (CC)
type msg_0x16 = record {
  fld_00 : bytestring &length=8; # Bot ID
  fld_01 : uint16; # Length(host_info)
  fld_02 : host_info; # Host information
  pad : bytestring &restofdata; # Padding
};

# Direction: both directions (Spam Server)
type msg_0x1c(is_inbound) =
  case is_inbound of {
    true  -> m1c_in : msg_0x1c_inbound;
    false -> m1c_out : msg_0x1c_outbound;
};

type msg_0x1c_inbound = record {
  fld_00 : uint32; # <unknown>
  fld_01 : uint32; # Size for memset
  fld_02 : uint32; # Size of fld_03
  fld_03 : bytestring &length = fld_02 # Compressed
  pad : bytestring &restofdata;
};

type msg_0x1c_outbound = record {
```

```
  fld_00 : bytestring &length = 16; # Cookie
  fld_01 : uint32; # Constant(0)
};

# Direction: outbound (Spam Server)
type msg_0x1d = record {
  fld_00 : bytestring &length = 16; # Cookie
  fld_01 : uint32; # Constant(0)
};

# Direction: inbound (CC)
type msg_0x21 = record {
  fld_00 : uint32; # <unknown>
  fld_01 : uint16; # Port
  fld_02 : uint8[] &until($element == 0); # Hostname
  pad : bytestring &restofdata; # Padding
};

# Direction: outbound (CC)
type msg_0x22 = record {
  fld_00 : bytestring &length=8; # Bot ID
  pad : bytestring &restofdata; # Padding
};

# Direction: outbound (CC)
type msg_0x23 = record {
  fld_00 : uint32; # Error code
  fld_01 : bytestring &length=8; # Bot ID
};

# Direction: inbound (CC)
type msg_0x24 = record {
  fld_00 : uint32; # IP address
  fld_01 : uint16; # Port
  pad : bytestring &restofdata; # Padding
};

# Direction: outbound (CC)
type msg_0x25 = record {
  fld_00 : bytestring &length=8; # Bot ID
  pad : bytestring &restofdata; # Padding
};
```

B Field Semantics

This appendix we provide some examples of functions used to identify the field semantics described in Table 1.

Cookies. Cookies represent data from a received network message that propagates to a sent message (e.g., session identifiers). Thus, a cookie is simultaneously identified in the received and sent messages. Note that, once a cookie has been identified we can check if it appears in later messages (both received and sent) in the dialog.

IP addresses. Dispatcher identifies IP addresses in received messages by monitoring if the parameters of some functions used to establish network connections (e.g., *connect*) or perform DNS reverse lookups (e.g., *getnameinfo*) have been derived from the received messages. Dispatcher identifies IP addresses in sent messages by tainting the output of functions that return local information (e.g., *gethostbyname*), remote information (e.g., *getpeername*), or functions that check the name of connected sockets (e.g., *getsockname*).

Error codes. Some programs report back unexpected errors using error codes. Dispatcher identifies error codes in sent messages by tainting the output of functions that report error conditions (e.g., *RtlGetLastWin32Error*).

File data. File data is data read from local files. Dispatcher can identify file data in sent messages by tainting the output of functions that read from file (e.g., *read*) or functions that map files directly into memory (e.g., *MapViewOfFile*). A special case of file data is user-specified *configuration data* such as the number of times to retry a connection. Dispatcher can mark file data as configuration data when provided with the list of files that contain the configuration information for the program.

File information. File information is file metadata such as the size of a file or the last modification date. Dispatcher identifies file information in sent messages by tainting the output of functions that query for file properties (e.g., *NtQueryInformationFile*).

Filenames. Filenames are a special case of file information. Dispatcher can identify filenames in received messages by analyzing if the parameters of functions used to open files (e.g., *open*) or used to get file properties (e.g., *NtQueryInformationFile*) have been derived from data previously received over the network. It can identify filenames in sent messages by tainting the output of functions that list the files in a directory (e.g., *NtQueryDirectoryFile*).

Hash / Checksum. We call both hash and checksum fields *verification fields* because they are often used to check if the data has been modified during transmission. Dispatcher identifies verification functions using the technique to identify encoding functions presented in Section 5. If the output of a encoding function is compared against a range of bytes received over the network, then that range is marked as a verification field in the received message. If the output of a encoding function appears on a sent message, then it is either a verification field or an encrypted/obfuscated field. Dispatcher can use the scope (the range of bytes in the sent message) to distinguish between a verification field and an encrypted/obfuscated field, since verification fields are usually shorter.

Hostnames. Hostnames can identify remote hosts as well as the local host. Dispatcher can identify hostnames in received messages by checking if the parameters of functions that start network connections (e.g., *connect*) are derived from received messages and in sent messages by tainting the output of functions that return local host information (e.g., *gethostname*).

Host information. We call any hardware or software properties of the host, *host information*. For example, when MegaD builds the message in Figure 1, it queries the operating system for information about the processor type, the operating system version, the memory status of the host or the type of connection to the Internet, all of which are examples of host information fields. Dispatcher identifies host information fields in sent messages by tainting the output of a variety of functions such as *GetVersionExA*, or *GlobalMemoryStatus*.

Keyboard input. Protocol messages often include data provided by the user via the keyboard, such as the filename in a FTP download, the domain name in a DNS query or the user name and password in an ICQ login session. Dispatcher identifies keyboard input in sent messages by tainting any data input by the user using the keyboard.

Keywords. Dispatcher identifies keyworkds in received messages using the techniques proposed in Polyglot [19] and in sent messages by tainting the memory region that contains a given module, as explained in Section 4.3.

Length. Dispatcher identifies length fields in received messages using previously proposed techniques [19, 48] and in sent messages using the techniques described in Section 4.3. Message length fields are a special type of length fields, that represent the length of a message on the wire. Dispatcher can identify message length fields in received messages by monitoring if some bytes in the received message are compared against the output of the function calls to read data from the socket (e.g, *read*, *recv*).

Padding. Dispatcher identifies padding in received messages by looking for tainted bytes that are not used by the program (only moved around) and that are present at the end of variable-length fields or at the end of the message. Dispatcher considers a padding field to be at most 5 bytes (64-bit alignment).

Ports. Ports are usually used altogether with IP addresses or hostnames to define an end point for a connection. Dispatcher identifies ports in received messages by analyzing how the parameters of functions used by the program to start new connections (e.g., *connect*) and bind new listening ports (e.g., *bind*) have been derived from a previously received message. Dispatcher identifies ports in sent messages by tainting the output of functions that check the name of connected sockets (e.g., *getsockname*).

Registry data. Registry data is any data stored in the Windows registry. Dispatcher identifies registry data in sent messages by tainting the output of functions that read data from the Windows registry (e.g., *NtQueryValueKey*).

Sleep timers. Sleep timers are timers used to indicate to a host that it should delay execution for a certain amount of time. Dispatcher identifies sleep timers in received messages by monitoring if the parameters to functions that delay execution (e.g., *sleep*) have been derived from data received over the network.

Stored data. Stored data is data received over the network that the program saves into permanent storage, so that it is kept even if a reboot happens. It includes data written to disk and the Windows registry. Dispatcher can identify stored data by monitoring if data received over the network is used as parameters for functions that write data to file (e.g., *write*) or the Windows registry (e.g., *NtSetValueKey*).

Timestamps. Timestamps are fields that contain time data. Dispatcher identifies timestamps in sent messages by tainting the output of functions that request the local or system time (e.g., *GetLocalTime*, *GetSystemTime*).