Automatic Protocol Reverse-Engineering: Message Format Extraction and Field Semantics Inference

Juan Caballero[a,*], Dawn Song[b]

[a] *IMDEA Software Institute, Madrid, Spain*
[b] *University of California, Berkeley, CA, USA*

Abstract

Understanding the command-and-control (C&C) protocol used by a botnet is crucial for anticipating its repertoire of nefarious activity. However, the C&C protocols of botnets, similar to many other application layer protocols, are undocumented. Automatic protocol reverse-engineering techniques enable understanding undocumented protocols and are important for many security applications, including the analysis and defense against botnets. For example, they enable active botnet infiltration, where a security analyst rewrites messages sent and received by a bot in order to contain malicious activity and to provide the botmaster with an illusion of successful and unhampered operation.

In this work, we propose a novel approach to automatic protocol reverse engineering based on dynamic program binary analysis. Compared to previous work that examines the network traffic, we leverage the availability of a program that implements the protocol. Our approach extracts more accurate and complete protocol information and enables the analysis of encrypted protocols. Our automatic protocol reverse-engineering techniques extract the message format and field semantics of protocol messages *sent* and *received* by an application that implements an unknown protocol specification. We implement our techniques into a tool called Dispatcher and use it to analyze the previously undocumented C&C protocol of MegaD, a spam botnet that at its peak produced one third of the spam on the Internet.

Keywords: protocol reverse-engineering, active botnet infiltration, message format extraction, field semantics inference, command-and-control protocol

1. Introduction

Protocol reverse-engineering techniques extract the specification of unknown or undocumented network protocols and file formats. Protocol reverse-engineering techniques are needed because many protocols and file formats, especially at the application layer, are closed (i.e., have no publicly available specification). For example, malware often uses undocumented network protocols such as the command-and-control (C&C) protocols used by botnets to synchronize their actions and report back on the nefarious activities. Commercial off-the-shelf applications also use a myriad of undocumented protocols and file formats. Closed network protocols include Skype's protocol [1]; protocols used by instant messaging clients such as AOL's ICQ [2], Yahoo!'s Messenger [3], and Microsoft's MSN Messenger [4]; and update protocols used by antivirus tools and browsers. Closed file formats include the DWG format used by Autodesk's AutoCAD software [5] and the PSD format used by Adobe's Photoshop software [6].

A detailed protocol specification can enable or enhance many security applications. For example, in this work we enable active botnet infiltration by extracting the specification of the C&C protocol used by the MegaD spam botnet and use it for deep packet inspection and rewriting of the C&C communication. Protocol specifications are also the input for generic protocol parsers used in network monitoring [7, 8] and can be used to build protocol-aware fuzzers

*Corresponding author
Email addresses: juan.caballero@imdea.org (Juan Caballero), dawnsong@cs.berkeley.edu (Dawn Song)

that explore deeper execution paths than random fuzzers can [9], as well as to generate accurate fingerprints required by fingerprinting tools that remotely distinguish among implementations of the same specification [10].

Currently, protocol reverse-engineering is mostly a time-consuming and error-prone manual task. Protocol reverse-engineering projects such as the ones targeting the MSN Messenger and SMB protocols from Microsoft [11, 12] [1], the Yahoo! Messenger protocol [14], or the OSCAR and ICQ protocols from AOL [15, 16], have all been long term efforts lasting 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, automatic protocol reverse-engineering techniques are needed.

Previous work on automatic protocol reverse-engineering proposes techniques that take as input network data [17, 18, 19]. Those techniques face the issue of limited protocol information available in network traces and cannot address encrypted protocols. To address those limitations, we present a new approach for automatic protocol reverse-engineering, which leverages the availability of a program that implements the protocol. Our approach uses dynamic program binary analysis techniques and is based on the intuition that monitoring how the program parses and constructs protocol messages reveals a wealth of information about the message structure and its semantics.

Compared to network traces, program binaries contain richer protocol information because they represent the implementation of the protocol, which is the most detailed description of the protocol in absence of the specification. Understanding the protocol implementation can be beneficial even for protocols with a publicly available specification, because implementations often deviate from the specification. In addition, for encrypted protocols, the program binary knows the cryptographic information required to decrypt and encrypt protocol data. Thus, we can wait until the program decrypts the received network data to start our analysis and stop it before the program encrypts the network data to be sent in response, thus revealing the structure and semantics of the underlying protocol.

Our work in context. This work comprises research published in two conference articles. The first article appeared in the proceedings of the 14th ACM Conference on Computer and Communications Security (CCS 2007). It presented a system called *Polyglot* [20], which implemented the first approach for automatic protocol reverse-engineering using dynamic binary analysis. Polyglot uses the intuition that monitoring the execution of a program that implements the protocol reveals a wealth of information about the protocol. Polyglot extracts only the message format of a received message. The second article appeared in the proceedings of the 16th ACM Conference on Computer and Communications Security (CCS 2009). It presented a system called *Dispatcher* [21], which in addition to the techniques introduced in Polyglot, implemented techniques to extract the message format for a sent message. It also implemented semantics inference techniques for both sent and received messages, which we had previously introduced in a Technical Report in 2007 [22].

After the publication of Polyglot, other research groups published automatic protocol reverse-engineering techniques that used dynamic binary analysis for extracting the protocol grammar [23, 24, 25] and the protocol state-machine [26]. The works that focus on protocol grammar extraction use the approach we introduced in Polyglot of monitoring the execution of a program that implements the protocol. Their techniques target two issues: 1) they consider the message format to be hierarchical [23, 24, 25], rather than flat as considered in Polyglot, and 2) they extend the problem scope from extracting the message format as done in Polyglot, to extracting the protocol grammar by combining information from multiple messages [23, 25]. In Dispatcher we still focus only on message format extraction because it is a pre-requisite for both protocol grammar and state-machine extraction, but we consider the hierarchical structure of the protocol messages. In this work, we present a unified view of the techniques introduced in Polyglot and Dispatcher that considers the hierarchical structure of protocol messages. We also unify the protocol nomenclature used across the different protocol reverse-engineering works.

2. Overview & Problem Definition

In this section we introduce automatic protocol reverse-engineering and its goals, describe the scope of the problem we address, introduce common protocol elements and terminology, formally define the problem, and provide an

[1] Microsoft has since publicly released the specification of both protocols as part of their Open Specification initiative [13]

overview of our approach.

2.1. Automatic Protocol Reverse-Engineering

The goal of automatic protocol reverse-engineering is given an undocumented protocol or file format to extract the *protocol grammar*, which captures the structure of all messages that comprise the protocol, and the *protocol state machine*, which captures the sequences of messages that represent valid sessions of the protocol. In this work we focus on reversing application layer protocols because those comprise the majority of all protocols and are more likely to be undocumented. In addition, we consider file formats a simple instance of a protocol where there are no sessions and each file corresponds to a single message.

Extracting the protocol grammar usually comprises two steps. First, given a set of input messages, extract the *message format* of each individual message. Second, combine the message format from multiple messages of the same type to identify complex message properties such as field alternation and optional fields. In this work we address the first step of protocol grammar extraction: extracting the message format for a given message. Extracting the message format is a pre-requisite for extracting both the protocol grammar and the protocol state-machine. The message format captures the field structure and the field semantics of the message, which we describe next.

Message format. The message format has two components: the *message field tree* and a *field attribute list* for each node in the tree. The message field tree[2] is a hierarchical tree structure where each node represents a field in the message and is tagged with a *[start:end]* range of offsets where the field appears in the message, where offset zero is the first byte in the message. A child node represents a subfield of its parent, and thus corresponds to a subrange of the parent field in the message. The children of a parent have non-overlapping ranges and are ordered using the lowest offset in their range. The root node represents the complete message, the internal nodes represent *records*[3], and the leaf nodes represent *leaf fields*[4], the smallest semantic units in a protocol. Note that leaf fields are sometimes referred to simply as fields. In this work, when we refer to fields in plural, we mean any node in the message field tree, which includes both records and leaf fields.

In addition to the range, a node contains a field attribute list, where each attribute captures a property of the field. Table 1 shows the field attributes that we consider in this work. The field boundary attribute captures how the recipient locates the boundary between the end of this field and the beginning of the next field in the message. For fixed-length fields the receiver can find the boundary using the constant field length value, which is known a priori. For variable-length fields the receiver can use a *delimiter*, i.e., a constant value that marks the end of the field, or a *length field*. The field dependencies attributes captures inter-field relationships such as this field being the length of another field or this field being the checksum of multiple other fields in the message. The field semantics attribute captures the type of data that a field carries. We explain the different protocol elements in more detail in the next section.

Note that the message field tree with the associated field ranges perfectly describes the structure of a given message. However, without the attribute list we cannot generalize anything learned from this message to other instances of the same message type (e.g., from one HTTP GET request to another).

Field semantics. One important field attribute is the field semantics, i.e, the type of data that the field contains. Typical field semantics include timestamps, hostnames, IP addresses, ports, and filenames. Field semantics are fundamental to understand what a message does and are important for both text and binary protocols. For example, an ASCII-encoded integer in a text-based protocol can represent among others a length, a port number, a sleep timer, or a checksum value. Field semantics are critical for many applications, e.g., they are needed in active botnet infiltration to identify interesting fields in a message to rewrite.

HTTP running example. Figure 1, captures the message field tree for an HTTP request. The HTTP request message is shown on the upper left corner and the box on the upper right corner shows the attribute list for one of the nodes. The root node in Figure 1 represents the complete HTTP request, which is 68 bytes long. There are four records:

[2] Also called protocol field tree [24].
[3] Also called hierarchical fields [24, 21] and complex fields [23].
[4] Also called finest-grained fields [24].

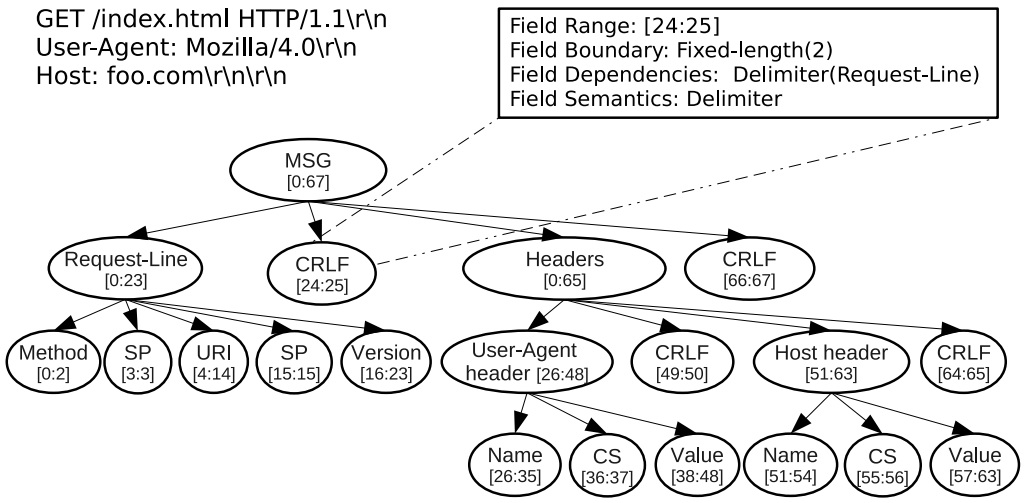

Figure 1: Message field tree for the HTTP request on the upper left corner. The upper right corner box shows the attribute list for one of the delimiters.

Attribute	**Value**
Field Range	Start and end offsets in message
Field Boundary	Fixed-length(l), Variable-length($Length$), Variable-length($Delimiter$)
Field Dependencies	Length(x_i), Delimiter(x_i), Checksum($x_i, \ldots, x_j$)
Field Semantics	The type of data the field carries. A value from Table 3

Table 1: Field attributes used in this work. Each attribute captures a property of the field.

the *Headers*, the *Request-Line*, the *User-Agent* header, and the *Host* header. HTTP mostly uses delimiters to mark the end of the variable-length fields. The field attribute list for the CRLF field shown in the figure, shows in the field semantics attribute that the *CRLF* field is a delimiter and in the field dependencies field that its target is the *Request-Line*. The HTTP specification is publicly available [27] and Figure 2 shows a partial HTTP grammar, taken from the specification, that covers most production rules related to our example HTTP request.

2.2. Protocol Elements

In this section we describe some elements commonly used in protocols and how they are represented in the message field tree and field attribute list.

Message. We define a message to be the Protocol Data Unit (PDU) of the application layer, where a PDU is the information that is delivered as a unit among peer entities in a networking layer. Table 2 shows the PDUs for different networking layers. We separate the application layer PDU with a horizontal line because there is no defined PDU for layers above the Transport layer [28].

Fixed-length and variable-length fields. Each field, regardless if a record or a leaf field, is either fixed-length or variable-length. The length value of a fixed-length field is static, i.e., it does not change across multiple instances of the same field. The length value for a fixed-length field is part of the protocol specification and known a priori to the implementations of the protocol. In contrast, the length of a variable-length field is dynamic, i.e., it can change across multiple instances of the same field. Protocol specifications need to describe how an implementation identifies the length or the boundary of a variable-length field. The main protocol elements used for this task are length fields and delimiters, which we describe next. The field boundary attribute captures whether a field is fixed-length or variable-length and for the latter whether it uses a delimiter or a length field. In our HTTP running example, all fields except

```
HTTP-message = Request | Response
Request = generic-message
Response = generic-message
generic-message = start-line
                  *(message-header CRLF)
                  CRLF
                  [ message-body ]
start-line = Request-Line | Status-Line
message-header = field-name ":" [ field-value ]
```

Figure 2: Partial HTTP protocol grammar from RFC 2616 [27].

Layer	**PDU**
Physical Layer	bit
Data Link Layer	frame
Network Layer	packet
Transport Layer	segment
Application Layer	message

Table 2: Protocol Data Units (PDUs) for the different networking layers.

the delimiters themselves are variable-length, while in our MegaD running example only the *MSG*, *Host-Info*, and *Padding* fields are variable-length, the rest are fixed-length.

Length fields. A length field captures the size of a *target* variable-length field, which can be a record or a leaf field. A length field always precedes its target field in a message, but it does not need to be its immediate predecessor. The length field can use different units. For example, in Figure 7 the *Msg-Length* field encodes the total length of the message in 64-bit units, but the *Length* field encodes the length of the *Host-Info* record in bytes. The value of the length field is often the output of a formula that may use the real length of the target field plus or minus some known constant. For example, a record may have three child fields: a fixed-length type field, a fixed-length length field, and a variable-length payload. In this case the length field can capture the payload length or the record length which includes the fixed-length of both the type field and the length field itself. The field dependencies attribute captures whether a field is a length field and what the target variable-length field is. The field boundary attribute captures for a target variable-length field whether its boundary is located using a particular length field.

Delimiters. A delimiter[5] is a constant used to mark the boundary of a target variable-length field. Delimiters are fields themselves and are always the successor of the target variable-length field they delimit. Delimiters are part of the protocol specification and known to the implementations of the protocol. Delimiters can be used in binary, text or mixed protocols, e.g., delimiters are used in HTTP, which is mainly a text protocol, and also in SMB, which is mainly a binary protocol. They can be formed by one or more bytes. For example, in Figure 1, the Carriage-Return plus Line-Feed two-byte sequence (CRLF) is used as a delimiter to mark the end of the start-line and the different message headers [27], while SMB uses a single null byte to separate dialect strings inside a *Negotiate Protocol Request* [29]. Protocols can have multiple delimiters. For example, in Figure 1, in addition to the CRLF delimiter to separate headers, there is also the space (SP) delimiter that marks the end of the *Method* and *URI* fields, as well as the semicolon plus space (CS) delimiter that separates the *Name* from the *Value* in each header field. As shown in the field attribute list in Figure 1, the field dependencies attribute captures whether a field is a delimiter and which field is its target. The field boundary attribute captures for a target variable-length field whether its boundary is located using a particular delimiter.

Field Sequences. Field sequences, or sequences for short, are lists of consecutive fields with the same type. Sequences

[5]In our early protocol reverse-engineering work [20] we referred to delimiters as separators. Since then, we have adopted the term delimiter because it has been more commonly used in follow-up work.

are used in file formats such as WMF, AVI or MPEG, and also in network protocols such as HTTP. A sequence is always variable-length, regardless if the fields that form the sequence are fixed-length or variable-length. The end of a sequence is marked using a delimiter or a special length field called a counter field. For example, in Figure 1, the *Headers* field is a sequence and an empty line (CRLF) delimiter is used to mark its end. Note that an array is a special case of a sequence where each field in the sequence has fixed-length. A sequence is simply a record in the message field tree with a field semantics attribute value that indicates so. All children of a sequence are of the same type.

Keywords. Keywords are protocol constants that appear in the protocol messages. Keywords are part of the protocol specification and known a priori to the implementations. Not all protocol constants are keywords, since there are protocol constants that never appear in a message, such as the maximum length of a field. Keywords can appear in binary, text, and mixed protocols and can be strings or numbers. For example, in Figure 1 the "GET", "HTTP", "User-Agent", and "Host" strings are all keywords, while in Figure 7 the version number is also a keyword. The field semantics attribute captures whether a field carries a keyword. A field can carry different keywords in different instances of the field. For example, the *Method* field in Figure 1 carries the "GET" keyword but in other HTTP requests it could carry the "POST" or "HEAD" keywords. Note that according to this definition delimiters are also keywords. We differentiate delimiters from other keywords because of their particular use.

Dynamic fields. Dynamic fields have been defined by previous work to be fields whose value may change across different protocol dialogs [30]. According to that definition almost any field in a protocol is dynamic. There are very few fields in protocols whose value never changes because they can encode very little information. In this work, we define dynamic fields to be fields that carry protocol-independent information, which means fields that never carry a protocol keyword.

2.3. Problem Definition

In this work we develop protocol reverse-engineering techniques to address two problems: *message format extraction* and *field semantics inference*. Message format extraction is the problem of extracting the message field tree for one message of the protocol. It can be applied to a message *received* by the application, as well as to a message *sent* by the application in response to a previously received message. Field semantics inference is the problem of given a message field tree, tagging each field in the tree with a semantics attribute specifying the type of data the field carries.

The input to our message format extraction and field semantics inference techniques is execution traces taken by monitoring an application that implements the protocol, while it is involved in a network dialog using the unknown protocol. The execution traces can be obtained by monitoring a live dialog, where the application communicates with another entity somewhere on the Internet, or an offline dialog, where we replay a previously captured dialog from the unknown protocol to the application. In both cases the application runs inside an *execution monitor* that tracks the program execution.

Our execution monitor is built on top of TEMU [31], a dynamic analysis platform that enables user-defined, instruction-level execution monitoring and instrumentation. TEMU is in turn implemented on top of the QEMU open-source whole-system emulator [32]. TEMU can run any PE/ELF program binary on an unmodified guest operating system (x86 Windows or Linux) inside another host operating system (Linux on x86). TEMU provides 3 main blocks of functionality over QEMU: an API to build custom plugins to monitor the execution of the guest machine, a taint tracking module, and an introspection module that enables the plugins to read some guest OS information such as process and thread identifiers. We have implemented a plugin for TEMU called *Tracecap* [33] that outputs an *execution trace*, containing all executed instructions and the contents of each instruction's operands, and an *allocation log* with information about the allocation/deallocation operations invoked by the program during the run. It can also output *snapshots* of the state of a process (memory and register contents) at a point in the execution.

Taking an execution trace of a program slows the execution 40 times on average [33]. While expensive, this is ameliorated by the fact that our execution traces only capture the execution between the time when the program receives the message and the time when the program outputs its response. Since the execution traces are short, their analysis typically runs in a few minutes, which is fine since there are no stringent performace requirements.

2.4. Approach

We design message format extraction and protocol inference techniques for both messages received and sent by an application. Thus, our approach can analyze both sides of the communication of an unknown protocol, even when an analyst has access only to the application implementing one side of the dialog. This is important because there are scenarios where access to applications that implement both sides of a dialog is difficult, such as reversing a botnet's C&C protocol where the binary of the C&C server is rarely available or reversing a proprietary instant-messaging protocol (e.g., Yahoo's YMSG or Microsoft's MSNP) where client implementations are publicly available, but server implementations are not.

To extract the format of *received* messages we use the following intuition: by monitoring how a program parses a received message we can learn the message format because in order to access the information in the leaf fields, the program first needs to find those fields by extracting the hierarchical structure of the message. By monitoring the parsing process, we can learn what the program already knows, e.g., the length of fixed-length fields and the values used as delimiters, as well as what the program has to discover, e.g., the boundaries of the variable-length fields. We present our message format extraction techniques for received messages in Section 3.

To extract the format of *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. 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 contains the unencrypted data at the time the encryption routine is invoked. Our intuition is that 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.

Our techniques to extract the message format differ for received and sent messages. For received messages, our techniques focus on how the program parses the message and leverage taint propagation [34, 35, 36, 37], a data-flow technique that allows us to follow how the received message is handled throughout the parsing. For sent messages, our techniques focus on how the program builds the message from its individual fields and leverage buffer deconstruction, which analyzes how the different memory buffers are used to fill the output buffer. Note that we do not leverage taint propagation for extracting the message field tree of sent messages, because only a fraction of all possible sources of taint information during message creation (e.g., output of system calls and data sections in the program) is actually used to build the sent message.

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 is used to derive the arguments 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) are used to derive some field in the output buffer, we can infer its semantics. We present our field semantic inference techniques for both received and sent messages in Section 5.

One limitation of our message format extraction and field inference techniques is that they work at the byte level. Thus, they currently cannot handle fields smaller than 8-bit, such as the QR (query or response) bit or the 4-bit Opcode (kind of query) in a DNS request [38]. While our techniques can be extended to operate at the bit-level with some engineering effort, an end-to-end solution requires the building blocks we use e.g., taint propagation, to also operate at the bit level.

Encrypted protocols. To handle encrypted protocols such as MegaD's C&C protocol, we use the intuition that the program binary knows the cryptographic information (e.g., cryptographic routines and keys) required to decrypt and encrypt the protocol messages. Thus, we can wait until the program decrypts the received message to start our analysis and stop the analysis before the program encrypts the message to be sent in response. Compared to previous work, we propose extensions to a recently proposed technique to identify the buffers holding the unencrypted received message [39]. Our extensions generalize the technique to support implementations where there is no single boundary between decryption and protocol processing, and to identify the buffers holding the unencrypted sent message. We present our handling of encrypted protocols in Section 6.

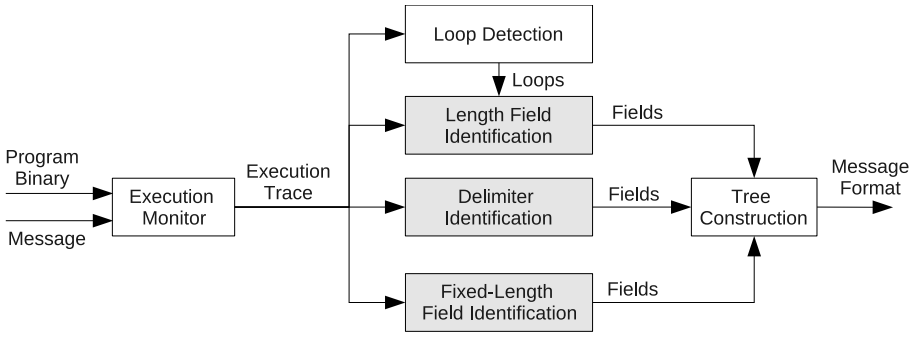

Figure 3: Message format extraction for received messages.

Per-message execution traces. An execution trace may contain the processing of multiple messages sent and received by the application during the network dialog. To separately analyze each message we need to split the execution trace into per-message traces. This is challenging when two consecutive messages are sent on the same direction of the communication. For example, MegaD uses a TCP-based C&C protocol. In a C&C connection the bot sends a request to the C&C server and receives one or more consecutive responses with the same format. At that point the question is whether to consider the response from the server a single message in which case there is a single message field tree where the child of the root node corresponds to a sequence with two child records, or to consider the response as two messages, in which case there are two separate message field trees.

Some work defines a message to be all data received by a peer before a response is sent, i.e., before the application calls the function that writes data to the socket [26]. This makes the response from MegaD's C&C server to be a single message. In this work we use a different definition of what a message is and 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 when the argument 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 of a program reading a field conveying the length of the message payload and using this value to read the payload itself.

Loop detection. Our format extraction techniques leverage a *loop detection module* that extracts the loops present in an execution trace. The loop detection module supports two different detection methods: static and dynamic. The static method first extracts loop information (e.g., the addresses of the loop head and the loop exit conditions) from control flow graphs. Then, it uses the loop information in a pass over the execution trace to detect the loops present in the execution. The dynamic method does not require any static processing and extracts the loops from a single pass on the execution trace, using techniques that detect loop backedges as instructions that appear multiple times in the same function [40]. The output of both loop detection methods is a list of loops present in the execution trace. The information for each loop includes the position of the loop in the execution trace as well as information about all the iterations of a loop (i.e., loops in an execution trace are unrolled). Both methods have pros and cons. The static method is often more accurate because it can precisely identify loop entry and exit points, but it requires analyzing all the modules (i.e., program executable and dynamically link libraries it uses, including operating system libraries) used by the application, may miss loops that contain indirection, and cannot be applied if the unpacked binary is not available. On the other hand, the dynamic method cannot detect loops that do not complete an iteration, needs heuristics to identify loop exit conditions, but requires no setup and can be used on any execution trace.

3. Message Format Extraction for a Received Message

The input for extracting the message format for a received message is an execution trace of the program while it parses the protocol message that we want to extract the format for. Here, the execution monitor taints each byte of the received message with a different taint offset where offset zero corresponds to the first byte in the message. The

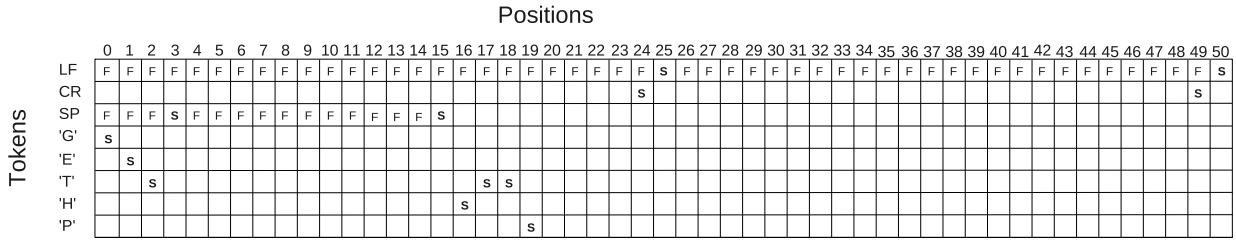

Figure 4: Partial token table for the HTTP request in Figure 1.

execution trace contains for each instruction operand whether the operand is tainted. If tainted, it contains the taint offsets for each byte in the operand. We refer to the taint offsets as *positions* in the received message. The output of this process is the message format as a message field tree with an attribute list for each node. This message field tree has no semantics attribute. In Section 5 we show how to extract the values for the field semantics attribute.

Figure 3 illustrates the message format extraction process for a received message. It shows that the execution trace is the input to three modules: delimiter identification, length identification, and fixed-length field identification. The delimiter and length identification modules focus on variable-length fields that use delimiters and length fields to mark their boundaries. In addition to the execution trace, the length identification module also takes as input the loop information provided by the loop detection module. We present the delimiter identification module in Section 3.1 and the length identification module in Section 3.2. The fixed-length identification focuses on fixed-length fields and is presented in Section 3.3. The fields identified by those three modules are added to the message field tree by the tree construction module.

3.1. Identifying Delimiters

In Section 2.2 we defined a delimiter to be a constant used to mark the boundary of a target variable-length field. Delimiters are part of the protocol specification and known to the programs that implement the protocol. The intuition to identify delimiters is that when parsing a received message, programs search for the delimiter by comparing the different bytes in the message against the delimiter. When a successful comparison happens, the program knows it has found a delimiter and therefore the boundary of the target variable-length field that precedes the delimiter.

For example, a Web server that receives the HTTP request in Figure 1 knows that the Carriage-Return plus Line-Feed (CRLF) sequence is the delimiter used to mark the end of the *Request-Line* field. The server compares the bytes in the request from the beginning (position zero) until finding the CRLF value at positions [24:25]. At that point the program knows that the range of the *Request-Line* is [0:23]. Similarly, the program knows that the space character (SP) is the delimiter used to mark the end of the *Method* and *URI* fields inside the *Request-Line*. Thus, the server compares the bytes in the *Request-Line* range with the space character until it finds it at position 3. At that point, it knows that the *Method* field comprises range [0:2]. Then, it continues scanning for the next occurrence of the space character, which is found at position 15. At that point it knows that the *URI* field comprises the range [4:14] and that the remainder of the *Request-Line* (range [16:23]) has to correspond to the *Version* field.

In a nutshell, our delimiter identification technique scans the execution trace looking for comparison operations between bytes from the received message (i.e., tainted bytes) and constant values (i.e., untainted bytes). For each comparison operation found, it stores for each tainted byte involved in the comparison, the position of the tainted byte, the constant value it was compared against, and the result of the comparison (i.e., success or failed). Then, it searches for tokens (i.e., byte-long constants) that are compared against multiple consecutive positions in the input message.

The detailed process comprises 4 steps: 1) generate a *token table* that summarizes all comparisons between tainted and untainted data in the execution trace, 2) use the token table to identify byte-long delimiters, 3) extend byte-long delimiters into multi-byte delimiters, and 4) add the delimiters and their target-fields to the message field tree. We describe these steps below.

Our delimiter identification technique has two important properties. First, it makes no assumptions about the

constants used as delimiters. Instead, it identifies delimiters by the way they are used. Second, it does not assume that the program searches for the delimiter in an ascending position order. All byte-comparisons between tainted and untainted data are recorded in the token table in the first step, before delimiters are identified. Thus, it does not matter the order in which the comparisons are done by the program, which is important because some programs like the Apache Web server scan backwards to find delimiters.

Generating the token table. The token table summarizes all comparisons between tainted and untainted data in the execution trace. Each row in the token table represents a token (i.e., a byte-long constant value) that at some point in the execution was compared against tainted data. Thus, the token table can have at most 256 rows. Each column represents a position in the received message from zero to the message size minus one. Each entry in the table represents whether the comparison between the token and the position in the received message was successful (S) or failed (F). Figure 4 shows a partial token table for the comparisons that a Web server performs on the first 51 bytes of the HTTP request in Figure 1. For brevity, we limit the table to only 8 tokens.

To populate the token table, the delimiter identification module scans the execution trace for comparison operations that involve tainted and untainted data. It breaks each comparison operation into one-byte comparisons and for each one-byte comparison it extracts the position of the tainted byte, the token it was compared against, and the result of the comparison. It adds a new entry with this information into the token table. Only equality comparisons are added to the table and comparison operations include not only compare (`cmp`) instructions but also other operations that compilers use to compare operands such as string comparison (`scas`) instructions or `test` instructions with identical operands (used to cheaply compare if an operand has zero value).

Extracting byte-long delimiters. To extract byte-long delimiters the delimiter identification module scans each row of the token table in ascending position order to find all sequences of consecutive positions that were compared against the token. A new sequence is started every time the current position is not consecutive with the previous one and every time a successful comparison is found. The reason to break a sequence at a successful comparison is that a successful comparison marks the presence of the delimiter and thus the end of the field it delimits. Once the list of all sequences for a token has been extracted, any sequence shorter than 3 positions is removed to avoid including spurious comparisons. We call each sequence of consecutive positions a *scope* and the output of this step is a list of byte-long delimiters with the associated scopes where the delimiter was used.

For example, from the token table in Figure 4 this step outputs two one-byte delimiters: the Line-Feed (LF) token with scopes [0:25] and [26:50], and the space (SP) token with scopes [0:3] and [4:15]. This shows that each one-byte delimiter can have multiple scopes and that different one-byte delimiters may have overlapping scopes since the scopes for the SP token are a subrange of the scope for the LF token. Thus, the delimiter scope hierarchy captures the hierarchical relationship between the *Request-Line* and the *Method* and *URI* fields. Note that two one-byte delimiters cannot have identical scopes since we require a successful comparison to mark the end of a scope.

Extending delimiters. When a delimiter consists of multiple bytes, e.g., the CRLF delimiter, the program can use different ways to find it such as searching for the complete delimiter or only searching for one byte in the delimiter and when it finds it, checking if the remaining bytes in the delimiter are also present. For multi-byte delimiters, the previous step identifies only one byte in the delimiter or all the bytes but as independent byte-long delimiters. For example, the token table in Figure 4 corresponds to a Web server that scans for the LF character and once found, it checks if the CR character is present in the previous position. In this case, the previous step identifies only the LF token as a one-byte delimiter.

In this last step, we try to extend each one-byte delimiter by analyzing the comparisons at the positions before and after all occurrences of the delimiter, i.e., the comparisons at the predecessor and successor positions for the last position in each scope. If the token table shows a successful comparison with the same token for all predecessor positions, we extend the delimiter with that token. If the token table shows a successful comparison with the same token for all successor positions, we extend the delimiter with that token and increase all scopes by one. The process recurses on each delimiter that was extended, until no more delimiters are extended. At that point, any duplicate scopes for a delimiter are removed. The output of this step is a list of multi-byte delimiters with the scopes where they are used.

For example, the one-byte LF delimiter identified in the previous step has scopes [0:25] and [26:50]. This step first

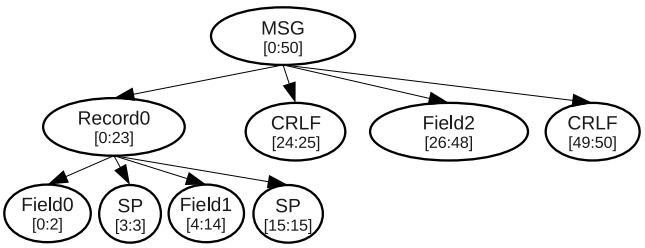

Figure 5: Partial message field tree generated by inserting the fields derived by identifying delimiters using the token table in Figure 4 into an empty tree.

checks the successor positions 26 and 51, finding no successful comparisons with the same token at those positions. Then, it checks the predecessor positions 24 and 49, finding that they all have a successful comparison against the CR token. Thus, the one-byte LF delimiter is extended to be a two-byte CRLF delimiter with identical scopes. The same process for the one-byte SP delimiter produces no extensions and the output of this step is two delimiters: CRLF with scopes [0:25] and [26:50], and SP with scopes [0:3] and [4:15].

Adding the delimiters and target fields to the message field tree. Once the delimiters have been identified, each scope is used to create two fields: a delimiter field with a range that covers the bytes in the delimiter and a variable-length field that covers the remainder of the scope. Both fields are added to the message field tree. For example, the [0:25] scope for the CRLF delimiter produces a delimiter field with range [24:25] and variable-length field with range [0:23]. Note that, the operation that inserts new nodes into the message field tree uses the field ranges to determine the correct position of the field in the tree. For example, Figure 5 shows the message field tree after inserting the fields derived by the delimiter identification process using the token table in Figure 4 into an empty tree. Note that the message field tree has a gap at depth 2 for the range [16:23], which corresponds to the *Version* field in Figure 1. Once the length and fixed-length field identification terminates, the tree construction module fills the gaps with fields.

3.2. Identifying Length Fields

The intuition behind our techniques for length 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 length field, it needs to use some value derived from the length field to advance the pointer to the end of the field. Thus, we identify length fields when they are *used* to increment the value of a pointer to the tainted data. For example, in Figure 6 we identify the length field at positions 12-13 when it is used to access positions 18-20.

We consider two possibilities to determine whether a field is being used as a length field: 1) the program computes the value of the pointer increment from the length 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.

Incrementing the pointer using arithmetic operations. For the first case, the program performs an indirect memory access where the effective address has been computed from some tainted data. Thus, when we find an indirect memory access that: 1) accesses a tainted memory location, and 2) where the effective address has been computed from tainted data (i.e., the base or index registers used to compute the address were tainted), we mark all the consecutive positions used to compute the effective address as part of a length field. In addition, we mark the smallest position in the effective address as the end of the target field. For example, in Figure 6 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 length field with length 2, and position 18 as the end of the target field. If a length 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 6 directs to position 18, and it appears again in an indirect memory access to position 27, we still consider the end of the target field to be position 18.

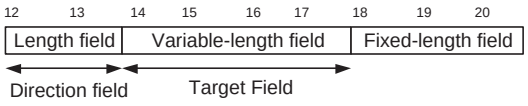

Figure 6: Length field example.

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. The length identification module uses the loop information provided by the loop detection module to identify loops in the trace that have a tainted exit condition. 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 length 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 length field.

Variable-length fields. Length fields are used to locate the end of a variable-length target field. To determine the start of the target variable-length field, without assuming any field encoding, we use the following approach. Length fields need to appear before their target field, so they can be used to skip it. Most often, as mentioned in [30] they precede the target field in the field sequence. After we locate a length field, we consider that the sequence of bytes between the last position belonging to the length field and the end of the target field, corresponds to a variable-length field. For example, in Figure 6, 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 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 length fields by looking at pointer increments and thus, it is independent of the encoding used in the length 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 [18, 30]. Thus, those techniques would miss length fields if they use other encodings, which do not belong to the set of pre-defined encodings being tested.

3.3. Identifying Fixed-Length Fields

In Sections 3.1 and 3.2 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. The intuition behind our fixed-length field identification technique is that fields are semantic units and programs take decisions based on the value of a field as a whole. Thus, when a field comprises multiple bytes, those bytes need to be used together in arithmetic operations, comparisons or other tasks. 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 length fields, using the techniques explained in Section 3.2. If no length 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

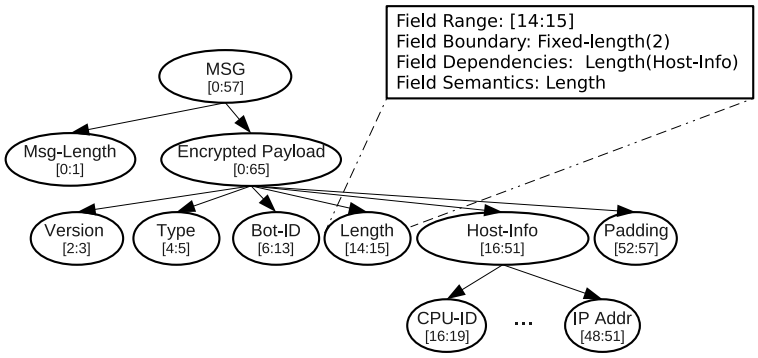

Figure 7: Message field tree for the MegaD Host-Information message.

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 delimiters and the length fields.

Even with this limitation, our approach is an improvement over previous work [18], 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.

4. Message Format Extraction for a Sent Message

The input for extracting the message format for a sent message is an execution trace of the program while it constructs the response to a given message. The output of this process is the message format as a message field tree with an attribute list for each node. This message field tree has no semantics attribute. In Section 5 we show how to extract the values for the field semantics attribute.

Our techniques to extract the message format for sent messages do not leverage taint propagation in the same way than the techniques for received messages do. For sent messages our techniques mostly work backwards (i.e., bottom-to-top) on the execution trace, while taint propagation is a forward (i.e., top-to-bottom) technique. Here, we leverage taint propagation in a different way by tainting the memory regions where the program under analysis and all dynamic libraries (DLLs) shipped with the program are loaded. Intuitively, protocol constants known to the program are stored in the data sections of those modules. Taint propagation allows us to track how those constants are used to build the sent message. This is needed to identify delimiters and keywords, which are constants as explained in Section 2.2.

MegaD running example. The MegaD botnet is one of the most prevalent spam botnets in use at the time of writing [41, 42]. MegaD uses an encrypted, binary (under the encryption), previously undocumented C&C protocol. Figure 7, corresponds to a message constructed by a MegaD bot to communicate back to the C&C server information about the bot's host. We use the message in Figure 7 as a running example throughout this section. The message is 58 bytes long and is partially encrypted. The *Msg-Length* field represents the total length of the message in 4-bit units and is unencrypted. The *Encrypted Payload* record corresponds to the encrypted part of the message. The other record contains the host information such as the CPU identifier and the IP address of the host.

Approach overview. The process of extracting the message format of a sent message is illustrated in Figure 8. It comprises three steps. The *preparation* step consists of a forward pass over the execution trace to extract information about the execution. We present the preparation step in Section 4.1. The core of the process is the *buffer deconstruction* step. The intuition behind buffer deconstruction is that the message field tree for the sent message is the inverse of the structure of the output buffer, which holds the message when is about to be sent on the network. Thus, deconstructing the output buffer into the memory buffers that were used to fill it with data reveals the message field tree of the sent message. This happens because programs store fields in memory buffers and construct the messages to be sent by combining those buffers together. Figure 9 shows the deconstruction of the output buffer holding the message in

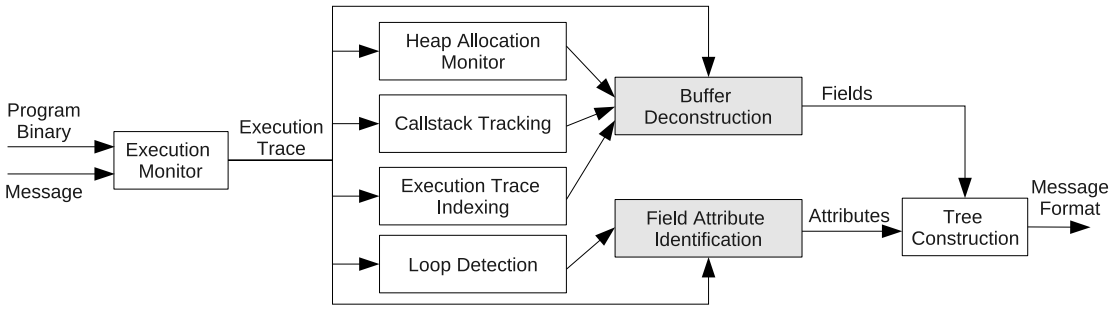

Figure 8: Message format extraction for sent messages.

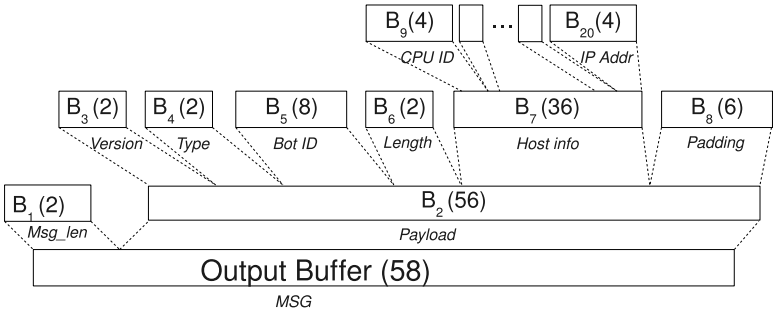

Figure 9: Buffer deconstruction for the MegaD message in Figure 7. Each box is a memory buffer starting at address B_x with the byte length in brackets. Note the similarity with the upside-down version of Figure 7.

Figure 7. Note how Figure 9 is the upside-down version of Figure 7. Buffer deconstruction is implemented as a backward pass over an execution trace. It outputs a message field tree with an empty field attribute list for each node (except the field range). We present buffer deconstruction in Section 4.2. Finally, the *field attribute inference* step identifies length fields, delimiters, field sequences, variable-length fields, and fixed-length fields. The information on those protocol elements is used to fill the field attributes in the message field tree. We present field attribute inference in Section 4.3.

4.1. Preparation

During preparation, a forward pass over the execution trace is made collecting information needed by buffer deconstruction and field attribute inference. Preparation uses four external modules: the execution trace indexing module, the call stack tracking module, the loop detection module, and the heap allocation monitor. It uses the trace indexing module to build a trace index that enables random access to the execution trace, needed by buffer deconstruction to scan the execution trace backwards. It uses the call stack tracking module to produce a function that given a instruction in the trace returns the function nesting when the instruction was executed, also needed by buffer deconstruction. It uses the loop detection module to extract information about the loops in the execution trace, needed by field attribute inference. Buffer deconstruction also needs information on whether two different writes to the same memory address correspond to the same memory buffer, since memory locations in the stack and the heap may be reused for different buffers. Buffer liveness information is gathered during preparation using the heap allocation monitor for heap buffers, and using the call stack tracking module to extract information about which memory locations in the stack are freed when the function returns.

4.2. Buffer Deconstruction

Buffer deconstruction is a recursive process. In each iteration it deconstructs a given memory buffer into a list of other memory buffers that were used to fill it with data. The process starts with the output buffer and recurses

until there are no more buffers to deconstruct. Each memory buffer that forms the output buffer (and, recursively, the memory buffers that form them) corresponds to a field in the message field tree. At the end of each iteration, for each memory buffer used to construct the current buffer, a field is added into the message field tree. For example, the output buffer in Figure 9 holds the message in Figure 7 before it is sent over the network. Deconstructing this output buffer returns a sequence of two buffers that were used to fill it with data: 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 message field tree. These two fields correspond to the *Msg-Length* and the *Encrypted Payload* fields in Figure 7.

Note that buffer deconstruction works at the binary level where a memory buffer is just a sequence of consecutive memory locations that were allocated in the same execution context. Thus, when any variable (e.g., an integer) is moved into memory (e.g., passed by value in a function call) it becomes a memory buffer. Buffer deconstruction has two parts. First, for each byte in the given buffer it builds a *dependency chain*. Then, using the dependency chains and the information collected in the preparation step, it deconstructs the given buffer. The input to each buffer deconstruction iteration is a buffer defined by its start address in memory, 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 arguments of the function that sends the data over the network (or the encryption function). The instruction number to start the analysis corresponds to the first instruction in the send (or encrypt) function. In the remainder of this section we introduce what program 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 (i.e., bytes) in the 32-bit EAX register: the lowest byte is EAX(0) and corresponds to the AL register, EAX(1) corresponds to the AH register, and EAX(2) and EAX(3) correspond to the higher two bytes in the register. 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*), which clears the 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 destination location (i..e, the location that was written), the source location (i.e., the location that was read), and the offset of the written location with respect to the beginning of the output buffer. Figure 10 shows the dependency chains for the B_2 buffer (the one that holds the encrypted payload) in Figure 9. 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 backward 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. We describe these four stop conditions next.

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 10. 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 deconstructing the given buffer, Dispatcher recurses on the buffers that form it. For example, in Figure 10 the dependency chains for locations *Mem(A+4)* through *Mem(A+11)* contains only one write operation because the source location is another memory location. Dispatcher will then create a new dependency chain for buffer *Mem(B)* through *Mem(B+7)*. 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*), and instructions

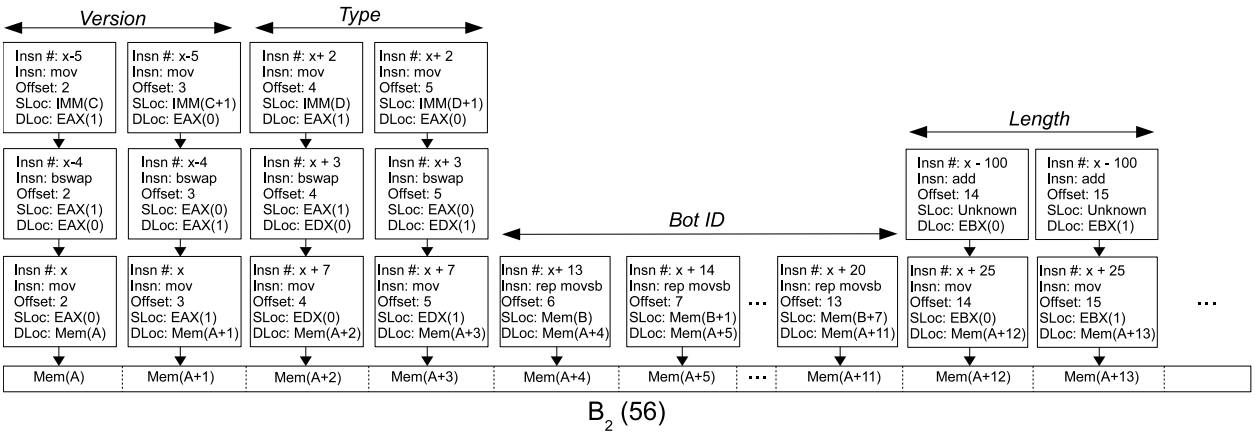

Figure 10: Dependency chain for B_2 in Figure 9. The start address of B_2 is A.

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., an arithmetic operation) as opposed to just moving the content around. Since programs operate on leaf fields, not on records, 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 10 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 7. 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. We call the source location of the last element in the dependency chain of a buffer location its *source*. 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).

To extract the structure for the given buffer Dispatcher iterates on the buffer locations from the buffer start to the buffer end. 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.

For example, in Figure 10 the source locations for *Mem(A+4)* and *Mem(A+5)* are contiguous locations *Mem(B)* and *Mem(B+1)* but the source locations for *Mem(A+11)* and *Mem(A+12)* are not contiguous. Thus, Dispatcher marks location *Mem(A+12)* as the beginning of a new range. Dispatcher finds 6 ranges in B_2. The first four are shown in Figure 10 and 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. The last two ranges correspond to the *Host Info* and *Padding* fields in Figure 7 and are not shown in Figure 10.

Once the buffer structure has been extracted, Dispatcher uses the correspondence between buffers and fields in the analyzed message to add one field to the message field tree per range in the buffer structure using the offsets relative to the output buffer. In Figure 10 it adds four new fields that correspond to the *Version*, *Type*, *Bot ID*, and *Length* in Figure 7. Note that buffer deconstruction focuses on the source and tail of the dependency chain, ignoring the possibly multiple instructions that may move a byte of data across different registers before writing it to a memory location. There are two reasons why we ignore those internal instructions in the chain. One is that registers are only temporary storage locations, the other one is that general-purpose registers have a maximum length (i.e., 4 bytes in a

32-bit architecture) that is smaller than the size of many variable-length fields. Thus, if those intermediate instructions where accounted for, the technique would split large fields into multiple smaller fields.

4.3. Field Attributes Inference

The message field tree built by buffer deconstruction captures the hierarchical structure of the output message, but does not contain field attributes other than the field range. Field attributes convene information that can be generalized from this message to other messages of the same type such as if a field is fixed-length or variable-length or inter-field relationships such as if a field represents the length of another target variable-length field. Similar to the need for buffer deconstruction, new field attribute inference techniques are also needed for sent messages. Next, we propose field attribute inference techniques designed to identify different protocol elements in sent messages. These techniques differ but share common intuitions with the techniques used for received messages: both try to capture fundamental properties of the protocol elements.

Length fields. We use three different techniques to identify length fields in sent messages. The intuition behind the techniques is that length fields can be computed either by incrementing a counter as the program iterates on the field, or by subtracting pointers to the beginning and the 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 that computes the length of a string (e.g., *strlen*) or is a subtraction of 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 [43] 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 precedes its target field). Any intersecting location is part of the length field for the field processed in the loop.

Delimiters. Delimiters are constants and it is difficult to differentiate them from other constants in the sent message. The technique to identify delimiters 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 are identified using the taint information introduced by tainting the memory regions containing the program and DLLs shipped it. 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. Fields that precede a delimiter and target fields for previously identified length fields are marked as variable-length fields. Fields derived from semantic sources that are known to be variable-length such as file data are also marked as variable-length. All other fields are marked as fixed-length. Note that some fields that a protocol specification would define as variable-length may encode always the same 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 hard-coded the *Server* string in the binary, making the field fixed-length for this implementation. Leveraging the availability of multiple implementations of the same protocol could help identify such cases.

Field sequences. The intuition behind identifying field sequences is that they are written in loops, one field at a time. The technique to identify sequences searches for loops that write multiple consecutive fields. For each loop, it adds to the message field tree one record field with the range being the combined range of all the consecutive fields written in the loop and with a *Sequence* field semantics value. It also adds one field per range of bytes written in each iteration of 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.

5. Field Semantics Inference

In this section we present our techniques to identify the field semantics of both received and sent messages. The intuition behind our type-inference-based techniques is that many functions and instructions used by programs contain rich semantic information. We can leverage this information to infer field semantics by monitoring if received network data is used at a point where the semantics are known (i.e., semantics sinks), or if data to be sent to the network has been derived from data with known semantics (i.e., semantics sources). Such semantics 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 others. The semantic information of those functions and instructions is publicly available in their prototypes, which describe their goal as well as the semantics of its inputs and outputs. Function prototypes can be found, for example, at the Microsoft Developer Network [44] or the standard C library documentation [45]. For instructions, one can refer to the system manufacturers' manuals.

Techniques. For *received* messages, Dispatcher uses taint propagation to monitor if a sequence of bytes from the received message is used in the arguments 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 of the arguments 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 arguments on the stack is tainted. The function's prototype tells us that the first argument 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 four bytes in the input, then Dispatcher can infer that those four bytes in the input message (identified by the offset in the taint information) form a field that contains an IP address to connect to. Similarly, if the memory locations that correspond to the port to connect to have been derived from two 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 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 of 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* instruction, we can leverage the knowledge that it takes no input and returns a 64-bit output representing the current value of the processor's time-stamp counter, which is placed in registers EDX:EAX [46]. Thus, at the time of execution when the program uses *rdtsc*, Dispatcher taints the EDX and EAX registers with a unique source identifier and offset pair. This pair uniquely labels the taint source to be from *rdtsc*, and the offsets identify each byte in the *rdtsc* stream (offsets 0 through 7 for the first use).

A special case of this technique is *cookie* inference. A cookie represents data from a received message that propagates unchanged 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 3 shows the field semantics that Dispatcher can infer from received and sent messages using the predefined functions.

Keywords. An important field semantic is keywords. Keywords are protocol constants that appear in network messages and are known a priori to the implementation. They are useful to create protocol signatures to detect services running on non-standard ports and mapping traffic to applications [47, 48]. Our intuition to identify keywords in received messages is that similar to delimiters, the program compares the keywords against the received application data. Dispatcher locates the keywords in the received message by analyzing the *successful comparisons* between tainted and untainted data, using comparison operations as the semantics sinks. The technique comprises two steps. The first step is identical to the first step in the delimiter identification technique presented in Section 3.1, that is, to populate the token table. The second step differs in that it focuses on the successful comparisons, rather than all the

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
Sequences	no	yes
Registry data	no	yes
Sleep timers	yes	no
Stored data	yes	no
Timestamps	no	yes

Table 3: Field semantics identified by Dispatcher for both received and sent messages. Stored data represents data received over the network and *written* to the filesystem or the Windows registry, as opposed to data *read* from those sources.

comparisons. It consists of scanning in ascending position order the columns in the token table. For each position, if we find a successful comparison, then we concatenate the token that was compared to the position to the current keyword. If no successful comparison is found at the current position, we store the current keyword and start a new keyword. We also break the current keyword and start a new one if the keyword crosses a field boundary as defined by the message field tree. This technique 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. For example, using the token table shown in Figure 4, Dispatcher identifies two HTTP keywords: "GET" at positions [0:2] and "HTTP" at positions [16:19].

To identify keywords in sent messages, Dispatcher taints the memory region that contains the module (and DLLs shipped with the main binary) with a specific taint source, 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 source are considered keywords.

6. Handling Encryption

Our protocol reverse-engineering techniques work on unencrypted data. Thus, when reversing encrypted protocols we need to address two problems. First, 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. Second, 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.

Recent work has looked at the problem of reverse-engineering the format of received encrypted messages [39, 49]. Since the application needs to decrypt the data before using it, those approaches monitor the application's processing of the encrypted message and locate the buffers that contain the decrypted data at the point that the decryption has finished by identifying the output buffers of functions with a high ratio of arithmetic and bitwise instructions. 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 present extensions to the technique presented in ReFormat [39], which flags encoding functions as functions with a high ratio of arithmetic and bitwise instructions.

Next, we describe our two extensions to the technique presented in ReFormat [39]. First, ReFormat can only handle

applications where there exists a single boundary between decryption and normal protocol processing. However, multiple such boundaries may exist. As shown in Figure 7 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. Consequently, we extend the technique to identify every *instance* of encryption, hashing, compression, and obfuscation, which we generally term *encoding functions*. Second, ReFormat was not designed to identify the buffers holding the decoded (unencrypted) data before encoding (encryption). Thus, we extend the technique to also cover this case. We present the generalized technique next.

Identifying encoding functions. To identify every instance of an encoding function we have simplified the process in ReFormat by removing the cumulative ratio of arithmetic and bitwise instructions for the whole trace (since we are interested in the ratio for each function), the use of tainted data, and the concept of leaf functions. The extended technique applies the intuition in ReFormat that the decryption process contains an inordinate number of arithmetic and bitwise operations to encoding functions. It makes a forward pass over the input execution trace using the call stack tracking module. For each function instance, 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 function's own instructions. It does not include instructions belonging to any called functions. Any function instance that executes a minimum number of instructions and has a ratio larger than a pre-defined threshold is flagged as an instance of an encoding function. The minimum number of instructions is needed because the ratio is not meaningful for functions that execute few instructions. In our experiments we set the minimum number of instructions to 20. We have experimentally set the threshold to 0.55 by training with a number of known encoding functions and selecting a threshold that minimizes the number of false negatives. We evaluate the technique in Section 7.3.

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, the encryption key, and any hard-coded 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.

7. Evaluation

In this section we evaluate our techniques on the previously undocumented C&C protocol used by the MegaD botnet, as well as a number of open protocols. MegaD is a prevalent spamming botnet first observed in 2007 and credited at its peak with responsibility for sending a third of the world's spam[50]. We use MegaD's proprietary and encrypted C&C protocol as a real-world test of our techniques. We use the open protocols to evaluate our techniques against a known ground truth.

7.1. Evaluation on MegaD

MegaD uses a proprietary, encrypted, binary protocol that has not been previously analyzed. Our MegaD evaluation has two parts. We first describe the information obtained by Dispatcher on the C&C protocol used by MegaD, and then show how the information extracted by Dispatcher can be used to rewrite a C&C dialog.

MegaD C&C Protocol. The MegaD C&C protocol uses TCP for transport on either port 80 or 443[7]. It employs a proprietary encryption algorithm instead of the SSL routines for HTTPS commonly used on port 443. Some MegaD bots use port 80 and others use 443 but the encryption and protocol grammar are identical regardless of the port.

[7]Malware often uses TCP ports 80 and 443 for their communication because those ports are often open in firewalls to enable Web browsing.

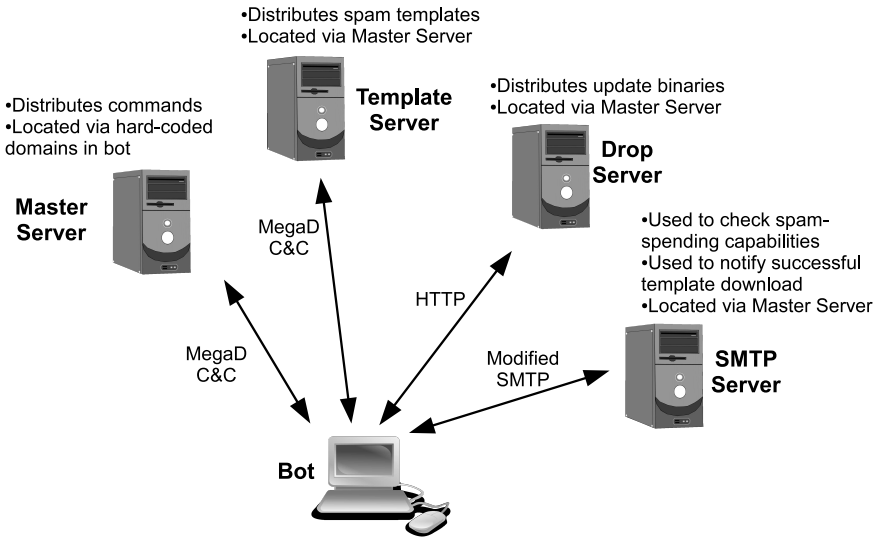

Figure 11: The four server types that a MegaD bot communicates with. The figure shows for each server the communication protocol used between the bot and the server, the main use of the server, and how the bot locates the server.

A MegaD bot communicates with four types of C&C servers: *Master Servers (MS)*, *Drop Servers (DS)*, *Template Servers (TS)*, and *SMTP Servers (SS)*. The four server types are illustrated in Figure 11. The botmaster uses the master servers to distribute commands to the bots. Bots locate a master server using a rendezvous algorithm, based on domain names hard-coded in the bot binaries. A bot employs pull-based communication using MegaD's C&C protocol to periodically probe the master server with a request message to which the server replies with a sequence of messages carrying authentication information and a command. The bot performs the requested action and returns the results to the master server. Drop servers distribute new binaries. A bot locates a drop server by receiving a message from its master server containing a URL specifying a file to download through HTTP. Template servers distribute the spam templates that bots use to construct spam. A bot locates a template server via a message from the master server specifying the address and port to contact. Again, communication proceeds in a pull-based fashion using MegaD's custom C&C protocol. SMTP servers play two distinct roles. First, bots check their spam-sending capabilities by sending them a test email using the standard SMTP protocol. A bot locates the SMTP server for this testing via a message from the master server specifying the server's hostname. Second, bots notify an SMTP server after downloading a new spam template and prior to commencing to spam. A bot locates the SMTP server used for template download notification via a control parameter in the spam template. The notification uses a modified SMTP protocol. Instead of sending the usual SMTP "HELO <hostname>" message, the bot sends a special "HELO 1" message and closes the connection.

Message format. We capture two MegaD C&C network traces by running the binary in a contained environment that forwards C&C traffic but blocks any other traffic from the bot (e.g., spam traffic). Our MegaD C&C traces contain 15 different messages (7 received and 8 sent by the bot). Using Dispatcher, we have extracted the message field tree for messages on both directions, as well as the associated field semantics. All 15 messages follow the structure shown in Figure 7 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 or 0x1, followed by a 2-byte message type field. The structure of the remaining payload depends on the message type. To summarize the protocol grammar we have used the output of Dispatcher to write a BinPac grammar [8] that comprises all 15 messages. Field semantics are added as comments to the grammar.

To the best of our knowledge, we are the first to document the C&C protocol employed by MegaD. 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

Program	Version	Protocol	Type	Guest OS
Apache [52]	2.2.1	HTTP	Server	Windows XP
BIND [53]	9.6.0	DNS	Server	Windows XP
FileZilla [54]	0.9.31	FTP	Server	Windows XP
Pidgin [55]	2.5.5	ICQ	Client	Windows XP
Sambad [29]	3.0.24	SMB	Server	Linux Fedora Core 5
TinyICQ [56]	1.2	ICQ	Client	Windows XP

Table 4: Different programs used in our evaluation on open protocols.

parse the messages using our grammar. For this, we employed a stand-alone version of the BinPac parser included in Bro [51]. 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.

Field attribute inference. The 15 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 7. 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. In particular, the message length is computed by subtracting the pointers to the end and beginning of the message, but then this result goes through a sequence of arithmetic and bitwise instructions that encodes the final number of bytes in value in the field.

Field semantics. Dispatcher identifies 11 different field semantics over the 15 messages: IP addresses, ports, host-names, 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 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 MegaD 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[8]. 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 may change between dialogs.

7.2. Evaluation on Open Protocols

In this section we evaluate our techniques on five open protocols: DNS, FTP, HTTP, ICQ, and SMB. To this end, we compare the output of Dispatcher with that of Wireshark [57] when processing 17 messages belonging to those 5 protocols. For each protocol we select at least one application that implements it, which we present in Table 4. For

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

		Wireshark		Dispatcher		Errors			
Protocol	**Message Type**	$\|L_W\|$	$\|H_W\|$	$\|L_D\|$	$\|H_D\|$	$\|E_W^L\|$	$\|E_D^L\|$	$\|E_W^H\|$	$\|E_D^H\|$
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 5: Comparison of the message field tree for sent messages extracted by Dispatcher and Wireshark 1.0.5. The ICQ client used is Pidgin. L_W and L_D are the set of leaf fields output by Wireshark and Dispatcher respectively, while H_W and H_D are the sets of record (hierarchical) fields. E_W^L and E_D^L denote the set of errors in leaf field output by Wireshark and Dispatcher, while E_W^H and E_D^H denote the set of errors in record fields.

each protocol, we select a set of protocol messages. For HTTP we evaluate how the Apache server [52] processes a `HTTP GET` request for the file "index.html" and the reply generated by the server. For DNS we evaluate an A query to resolve the IP address of the domain "test.example.com" sent to the BIND name server [53] and its corresponding reply. For FTP we analyze the sequence of messages sent by the FileZilla server [54] in response to a connection, as well as the messages sent when the username and password are received. For ICQ we analyze the messages in a login connection sent by the Pidgin client tool [55] and the responses from the server interpreted by the TinyICQ client tool [56]. For SMB, we analyze a `Negotiate Protocol Request` received by the Sambad open source server [29].

Message format. Wireshark is a network protocol analyzer containing manually written grammars (called dissectors) for a large variety of network protocols. Although Wireshark is a mature and widely-used tool, its dissectors have been manually generated and therefore are not completely error-free. Wireshark dissectors parse a message into a message field tree. The internal message field tree is not output in a visual representation by Wireshark but is accessible through the library functions. To compare the accuracy of the message format automatically extracted by Dispatcher to the manually written 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 any differences between the output of both tools to be due to errors in Dispatcher, Wireshark, or both.

We denote the set of leaf fields and the set of records in the message field tree output by Wireshark as L_W (L stands for leaf) and H_W (H stands for hierarchical), respectively. L_D and H_D are the corresponding sets for Dispatcher. Table 5 shows the evaluation results for sent messages and Table 6 for received messages. For each protocol and message the tables show the number of leaf fields and records in the message field tree output by both tools as well as the result of the manual classification of its errors. Here, $|E_W^L|$ and $|E_D^L|$ represent the number of errors on leaf fields in the message field tree output by Wireshark and Dispatcher respectively. Similarly, $|E_W^H|$ and $|E_D^H|$ represent the number of errors on records.

The results show that Dispatcher outperforms Wireshark when identifying leaf fields. This result is mainly due to the inconsistencies between the different dissectors in Wireshark when identifying delimiters. Some dissectors do not add delimiters to the message field tree, some 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. Figure 12 illustrates some of the errors made by Wireshark. It shows the message field tree for a simple HTTP response output by Wireshark. The dotted nodes are missing nodes that Wireshark does not output, which include delimiters, the reason field and the children of the `Server header` field.

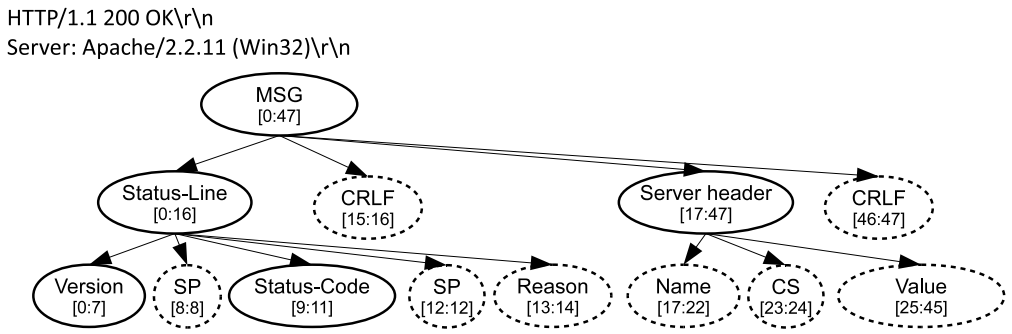

Figure 12: Message field tree for a simple HTTP response output by Wireshark. The dotted nodes are fields that Wireshark does not output.

		Wireshark		**Dispatcher**		**Errors**			
Protocol	**Message Type**	$\|L_W\|$	$\|H_W\|$	$\|L_D\|$	$\|H_D\|$	$\|E_W^L\|$	$\|E_D^L\|$	$\|E_W^H\|$	$\|E_D^H\|$
HTTP	GET request	13	2	40	10	27	2	8	0
DNS	A query	14	3	13	1	1	0	0	2
ICQ	New connection	38	11	36	11	0	2	0	0
	Close connection	13	3	10	3	0	3	0	0
SMB	Negotiate Protocol Request	48	16	39	11	9	6	0	5
Total		126	35	138	36	37	13	8	7

Table 6: Comparison of the message field tree for received messages extracted by Dispatcher and Wireshark 1.2.8. The ICQ client is TinyICQ.

The results also show that Wireshark outperforms Dispatcher when identifying records. For sent messages, 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 loop. For example, if an array has only two elements, the source-level loop that processes the field iterates only twice and the compiler may decide to unroll the two iterations at the binary-level. Thus, at the binary level there is no loop that handles both records in the array and Dispatcher will flag them as separate fields rather than as two records of an array. For received messages it is often due to the loop that processes the record being missed by the detection because it executed only one iteration[9].

The two main sources of errors for Dispatcher when analyzing sent messages are: consecutive fields that are stored as a single string in the program binary and arrays that are not written using a loop. An example of consecutive fields stored as a unit by the application is the error in the *Status-Line* record of the HTTP reply message. The HTTP/1.1 specification [27] states that its format is: *Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF*, but Dispatcher considers the *Status-Code*, the delimiter, and the *Reason-Phrase* to belong to the same field because all three fields are stored as a single string in the server's data section, which is copied as a whole into the sent message. An example of a program processing an array without a loop is the *BIND* server processing separately the *Queries*, *Answers*, *Authoritative*, and *Additional* sections in the DNS reply. This introduces four errors in the results because Dispatcher cannot identify that they form an array.

The two main sources of errors for Dispatcher when analyzing received messages are: fields smaller than one byte and unused fields. An example of fields smaller than one byte are the fields that comprise the flags records in the DNS and SMB messages. Since Dispatcher works at the byte level it currently does not identify fields smaller than one byte. Unused fields are fields that the program only moves without performing any other operation on them. When two consecutive unused fixed-length fields are found, Dispatcher groups them as a single field introducing an error.

[9]Here the tool uses the dynamic loop detection method, which can only detect loops that complete a full iteration, i.e., where the backedge is taken.

Number of traces	Number of functions	False Positives	False Positive Rate
20	3,467,458 (16,852)	87 (9)	0.002%

Table 7: Evaluation of the detection of encoding functions. Values in parentheses represent the numbers of unique instances. False positives are computed based on manual verification.

For example, in the SMB *Negotiate Protocol Request* message, the *Process ID High*, *Signature*, *Reserved*, *Tree ID*, and *Process ID* fields are all grouped together by Dispatcher into a single unused field. These errors in sent and received messages 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 specification.

Overall, Dispatcher and Wireshark achieve similar accuracy. Note that we do not claim that Dispatcher will always be as accurate as Wireshark, since we are only evaluating a limited number of protocols and messages. However, the results show that the accuracy of the message format automatically extracted by Dispatcher can rival that of Wireshark, without requiring a manually generated grammar.

Field Attribute Inference. The 17 messages contain 34 length fields, 73 delimiters, 133 variable-length fields, and 6 arrays. We have analyzed in detail the errors in the field attribute inference for sent messages. 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 in sent messages 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 in sent messages. Out of 3 arrays, Dispatcher misses one formed by the *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 semantics in Table 3 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.

7.3. Detecting Encoding Functions

To evaluate the detection of encoding functions presented in Section 6 we perform the following experiment. We obtain 25 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 uses its own encryption algorithm and the *zlib* library for compression and Apache uses SSL with AES and SHA-1[10]. The remaining 20 execution traces are from a variety of programs including three browsers processing the same plain HTTP response (Internet Explorer 7, Safari 3.1, and Google Chrome 1.0), a DNS server processing a received request (BIND), a Web server processing an HTTP GET request (AtpHttpd), the Microsoft SQL server processing a request for database information (MSSQL), as well as the RPC service embedded in Windows handling a directory listing. For all these 15 traces the inputs do not contain any checksums, encrypted or compressed data, so we believe they are free of encoding functions.

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. To evaluate false negatives, we run Dispatcher on the Apache-SSL trace. Dispatcher correctly identifies all encoding functions. To evaluate false positives, we run Dispatcher on the 20 traces that do not contain encoding functions. The results are shown in Table 7. The 20 execution traces contain over 3.4 million functions calls from over 16,852 unique functions. Dispatcher flags 87 function instances as encoding functions, belonging to nine unique functions. Using function names and debugging information, we have been able to identify two out of those nine functions: `memchr` and `comctl32.dll::TrueSaturateBits`. Based on these results, our technique correctly identifies all known encoding functions and has a false positive rate of 0.002%.

Next, we run Dispatcher on the four MegaD execution traces. Four unique encoding functions are identified.

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

Three of them appear in all four execution traces: the decryption routine, the encryption routine, and a key generation routine that generates the encryption and decryption keys from a seed value in the binary before calling the encryption or decryption routines. In addition, in one execution trace Dispatcher flags a fourth function that corresponds to the *inflate* function in the *zlib* library, which is statically linked into the MegaD binary.

8. Related Work

Protocol reverse-engineering projects that enable interoperability of open solutions with proprietary protocols have existed for a long time. Those projects relied on manual techniques, which are slow and costly [11, 12, 14, 15, 16]. Automatic protocol reverse-engineering techniques can be used, among other applications, to reduce the cost and time associated with these projects.

Protocol format. Early work on automatic protocol format reverse-engineering takes as input network traffic. Among these approaches, the Protocol Informatics project pioneers the use of sequence alignment algorithms [17] and Discoverer proposes a related, but improved, technique that first tokenizes messages into a sequence of binary and text tokens, then clusters similar token sequences and finally merges similar sequences using type-based sequence alignment [18]. Approaches based on network traffic are useful when a program that implements the protocol is not available. However, they cannot reverse encrypted protocols and are limited by the lack of protocol information in network traffic. Leveraging a program that implements the protocol significantly improves the reversed-engineered format.

Lim et al. [58] use static analysis on program binaries 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. Their static analysis approach requires sophisticated analysis to deal with indirection and cannot handle packed binaries such as MegaD. Also, they do not extract the format of received messages or infer field semantics.

In Polyglot, we propose a dynamic binary analysis approach for extracting the message format of received messages that does not require any a priori knowledge about the program or the protocol and can effectively deal with indirection and packed binaries [20]. Dynamic binary analysis techniques are also used in follow-up work that extracts the hierarchical message format [23, 24, 25, 59]. We detail those works next.

In Autoformat, the authors propose techniques to extract the message field tree of received messages and to identify field sequences [24]. Their technique groups together consecutive positions in the input message that are processed by the same function. However, a function may parse multiple fields, e.g., when parsing two consecutive fixed-length fields in a binary protocol. Their output message field tree captures the hierarchical structure of the message but contains no field attributes. Thus, it cannot be used to generalize across multiple instances of the same message. To identify field sequences their technique looks for input positions that with similar execution history, i.e., that have been processed by the same functions.

Wondracek et al. propose techniques to extract the message format of received messages and use hierarchical sequence alignment to identify optional fields, alternation, and sequences of identically-structured records [23]. Their message format captures the hierarchical field structure and contains field attributes such as length fields and delimiters, which can be used to generalize across messages. When identifying leaf fields, they break an input chunk that is not a delimiter, length field, or variable-length field, into individual bytes and thus may miss identifying multi-byte fixed-length fields.

In Tupni the authors propose techniques to identify field sequences and to aggregate information from multiple messages [25]. Their field sequence identification technique groups together input positions that are handled in the same loop iteration. Tupni identifies fields with the same type across different messages by comparing the set of instructions that operate on the fields. Then, it aligns fields based on their types using the technique in [18].

Lin and Zhang develop techniques to extract the input syntax tree of inputs with top-down or bottom-up grammars [59]. Their technique for top-down grammars has the same scope as the works above and assumes that program control dependence follows program parsing. However, many programs may not satisfy this assumption, e.g., they may backtrack to previously scanned fields or they may perform error checks that do not reveal input structure but modify the program's control dependence. Their input syntax tree represents the hierarchical structure of the input but

does not allow to generalize to other inputs, similar to a message field tree with no field attributes. They also propose a technique for inputs with bottom-up grammars, commonly used in programming languages.

In Dispatcher, we propose message format extraction techniques for sent messages and field semantics inference techniques for both received and sent messages. Compared to the above approaches, Dispatcher is able to extract the message format for received and sent messages from the same binary. This is important in scenarios where only the program that implements one side of the dialog is available such as when analyzing the C&C protocols used by botnets and instant messaging protocols. In addition, Dispatcher extracts fine-grained field semantics, which are important to understand what a message does, as well as for identifying fields with the same type across multiple messages.

Semantics inference. Aggregate Structure Identification (ASI) [60] proposes a static analysis approach for decomposing aggregate data structures in Cobol programs by leveraging the program's access patterns and the type information from system calls and well-known library functions. This work follows ASI in leveraging the type information from system calls and well-known library functions for type inference, but uses a dynamic data-flow approach and requires no access to the source code. In this work we only recover semantic information for a single buffer holding a received message or a message about to be sent on the network. Recently, Rewards [61] generalizes our dynamic semantics inference approach from well-known library functions to type the program's internal data structures.

State-machine. In addition to extracting the protocol grammar, protocol reverse-engineering also includes inferring the protocol's state-machine. ScriptGen infers the protocol state-machine from network data [62]. Due to the lack of protocol information in network data it is difficult for ScriptGen to determine whether two network messages are two instances of the same message type. This is needed when converting messages into alphabet symbols. ScriptGen outputs a state-machine that captures only previously-seen sessions, without generalization. Prospex uses execution trace similarity metrics to cluster messages of the same type so they can be assigned the same symbol from the alphabet [26]. Then, it extracts a tree that captures previously-seen sessions, labels the tree nodes using heuristics, and applies an algorithm to infer the minimal consistent DFA. Techniques to extract the message format like the ones presented in this work are a prerequisite for techniques that extract the protocol state-machine.

Protocol specification languages. Previous work has proposed languages to describe protocol specifications [7, 63, 8]. Such languages are useful to store the results from protocol reverse-engineering and enable the construction of generic protocol parsers. In this work, we use the BinPac language to represent our MegaD C&C protocol specification and the generic BinPac parser to analyze MegaD messages given that specification [8].

Other related work. Previous work has targeted protocol reverse-engineering for specific applications like protocol replay or inferring connections that belong to the same application session. RolePlayer [30] and ScriptGen [62, 19] address the problem of replaying previously captured network sessions. Such systems perform limited protocol reverse-engineering from network traffic 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. Our field semantics inference techniques leverage the richer semantics available in protocol implementations compared to network traffic, accurately extracting a wide range of field semantics for dynamic fields. Replayer uses dynamic binary analysis to replay complete program executions that correspond to network dialogs [64]. Previous work also addresses the related problem of identifying multiple connections that belong to the same application session from network traffic [65].

9. Conclusion

In this work, we have proposed a new approach for automatic protocol reverse-engineering that uses dynamic program binary analysis. Our approach takes as input execution traces obtained by running a program that implements the protocol, while it processes a received message and builds the corresponding response. Compared to previous approaches that take as input network traces, our approach infers more complete protocol information and can analyze encrypted protocols.

We have develop techniques to extract the message format and the field semantics of messages on both directions of the communication, even when only one endpoint's implementation of the protocol is available. Our message format extraction techniques identify the field structure of a message as well as hard-to-find protocol elements in the

message such as length fields, delimiters, variable-length fields, and multiple consecutive fixed-length fields. Our field semantics inference techniques identify a wealth of field semantics including filenames, IP addresses, timestamps, ports, and error codes. In addition, we have shown how to apply our techniques to encrypted protocols by identifying the buffers that hold the unencrypted received message after decryption and the unencrypted message to be sent before encryption.

We have implemented our techniques in a tool called Dispatcher and have used it to extract the grammar of the previously undocumented, encrypted, C&C protocol of MegaD, a prevalent spam botnet. We have shown how the protocol grammar enables active botnet infiltration by rewriting a message that the bot sends to the C&C server. Furthermore, we have evaluated our techniques on a variety of open protocols and compared Dispatcher's output with the output of Wireshark, a state-of-the-art protocol analyzer. Dispatcher achieves similar accuracy as Wireshark, without requiring access to the protocol grammar.

10. Acknowledgements

We would like to thank our co-authors Christian Kreibich, Zhenkai Liang, Pongsin Poosankam, and Heng Yin for their contributions to the two papers that form the basis of this work. We also thank the anonymous reviewers for their insightful comments. This research was partially supported by the National Science Foundation under Grants No. 0311808, No. 0433540, No. 0448452, No. 0627511, and CCF-0424422, by the Air Force Office of Scientific Research under MURI Grant No. 22178970-4170, by the Army Research Office under the Cyber-TA Research Grant No. W911NF-06-1-0316, and by CyLab at Carnegie Mellon under grant DAAD19-02-1-0389 from the Army Research Office. During the preparation of this manuscript, Juan Caballero was partially funded by the European Union through Grants FP7-ICT No. 256980 and FP7-PEOPLE-COFUND No. 229599 and by the Spanish Government through a Juan de la Cierva Fellowship. Opinions expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

References

[1] Skype, `http://www.skype.com`.

[2] icq, `http://www.icq.com`.

[3] Yahoo! messenger, `http://messenger.yahoo.com`.

[4] Microsoft, Windows live messenger, `http://messenger.msn.com`.

[5] Autodesk, `http://autodesk.com`.

[6] Adobe photoshop, `http://www.adobe.com/products/photoshop/`.

[7] N. Borisov, D. J. Brumley, H. J. Wang, C. Guo, Generic application-level protocol analyzer and its language, in: Proceedings of the Network and Distributed System Security Symposium, San Diego, CA, 2007.

[8] R. Pang, V. Paxson, R. Sommer, L. Peterson, binpac: A yacc for writing application protocol parsers, in: Proceedings of the Internet Measurement Conference, Rio de Janeiro, Brazil, 2006.

[9] peach fuzzing platform, `http://peachfuzzer.com/`.

[10] Nmap, `http://www.insecure.org`.

[11] M. Mintz, A. Sayers, Unofficial guide to the msn messenger protocol, `http://www.hypothetic.org/docs/msn/` (December 2003).

[12] A. Tridgell, How samba was written, `http://samba.org/ftp/tridge/misc/french_cafe.txt` (August 2003).

[13] Microsoft, Open specifications, http://www.microsoft.com/openspecifications.

[14] Libyahoo2: A c library for yahoo! messenger, http://libyahoo2.sourceforge.net.

[15] A. Fritzler, The unofficial aim/oscar protocol specification, http://www.oilcan.org/oscar/ (April 2008).

[16] icqlib: The icq library, http://freshmeat.net/projects/icqlib/.

[17] M. A. Beddoe, Network protocol analysis using bioinformatics algorithms, http://www.4tphi.net/ awalters/PI/PI.html.

[18] W. Cui, J. Kannan, H. J. Wang, Discoverer: Automatic protocol description generation from network traces, in: Proceedings of the USENIX Security Symposium, Boston, MA, 2007.

[19] C. Leita, M. Dacier, F. Massicotte, Automatic handling of protocol dependencies and reaction to 0-day attacks with scriptgen based honeypots, in: Proceedings of the International Symposium on Recent Advances in Intrusion Detection, Hamburg, Germany, 2006.

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

[21] J. Caballero, P. Poosankam, C. Kreibich, D. Song, Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering, in: Proceedings of the ACM Conference on Computer and Communications Security, Chicago, IL, 2009.

[22] J. Caballero, D. Song, Rosetta: Extracting protocol semantics using binary analysis with applications to protocol replay and nat rewriting, Tech. Rep. CMU-CyLab-07-014, Cylab, Carnegie Mellon University, Pittsburgh, PA (October 2007).

[23] G. Wondracek, P. M. Comparetti, C. Kruegel, E. Kirda, Automatic network protocol analysis, in: Proceedings of the Network and Distributed System Security Symposium, San Diego, CA, 2008.

[24] Z. Lin, X. Jiang, D. Xu, X. Zhang, Automatic protocol format reverse engineering through context-aware monitored execution, in: Proceedings of the Network and Distributed System Security Symposium, San Diego, CA, 2008.

[25] W. Cui, M. Peinado, K. Chen, H. J. Wang, L. Irun-Briz, tupni: Automatic reverse engineering of input formats, in: Proceedings of the ACM Conference on Computer and Communications Security, Alexandria, VA, 2008.

[26] P. M. Comparetti, G. Wondracek, C. Kruegel, E. Kirda, Prospex: Protocol specification extraction, in: Proceedings of the IEEE Symposium on Security and Privacy, Oakland, CA, 2009.

[27] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, Hypertext transfer protocol – http/1.1, RFC 2616 (Draft Standard) (June 1999).
URL http://www.ietf.org/rfc/rfc2616.txt

[28] W. R. Stevens, tcp/ip Illustrated, Volume 1: The Protocols, Addison-Wesley, 1994.

[29] samba, http://samba.org.

[30] W. Cui, V. Paxson, N. C. Weaver, R. H. Katz, Protocol-independent adaptive replay of application dialog, in: Proceedings of the Network and Distributed System Security Symposium, San Diego, CA, 2006.

[31] temu: The bitblaze dynamic analysis component, http://bitblaze.cs.berkeley.edu/temu.html.

[32] qemu: Open source processor emulator, http://wiki.qemu.org.

[33] J. Caballero, Grammar and model extraction for security applications using dynamic program binary analysis, Ph.D. thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA (September 2010).

[34] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, M. Rosenblum, Understanding data lifetime via whole system simulation, in: Proceedings of the USENIX Security Symposium, San Diego, CA, 2004.

[35] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, P. Barham, vigilante: End-to-end containment of internet worms, in: Proceedings of the Symposium on Operating Systems Principles, Brighton, United Kingdom, 2005.

[36] J. Newsome, D. Song, Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software, in: Proceedings of the Network and Distributed System Security Symposium, San Diego, CA, 2005.

[37] G. E. Suh, J. W. Lee, D. Zhang, S. Devadas, Secure program execution via dynamic information flow tracking, in: Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems, Boston, MA, 2004.

[38] P. Mockapetris, Domain names - implementation and specification, RFC 1035 (Standard) (November 1987). URL http://www.ietf.org/rfc/rfc1035.txt

[39] Z. Wang, X. Jiang, W. Cui, X. Wang, M. Grace, reformat: Automatic reverse engineering of encrypted messages, in: Proceedings of the European Symposium on Research in Computer Security, Saint-Malo, France, 2009.

[40] M. Kobayashi, Dynamic characteristics of loops, IEEE Transactions in Computers 33 (2). doi:http://dx.doi.org/10.1109/TC.1984.1676404.

[41] M86 Security Labs, Security threats: Email and web threats, http://www.marshal.com/newsimages/trace/Marshal8e6_TRACE_Report_Jan2009.pdf (January 2009).

[42] J. P. John, A. Moshchuk, S. D. Gribble, A. Krishnamurthy, Studying spamming botnets using botlab, in: Proceedings of the Symposium on Networked System Design and Implementation, Boston, MA, 2009.

[43] P. Saxena, P. Poosankam, S. McCamant, D. Song, Loop-extended symbolic execution on binary programs, in: Proceedings of the International Symposium on Software Testing and Analysis, Chicago, IL, 2009.

[44] Microsoft, msdn: The microsoft developer network, http://msdn.microsoft.com.

[45] ISO/IEC, The iso/iec 9899:1999 c programming language standard (May 2005).

[46] Intel, intel64 and ia-32 architectures software developer's manuals, http://www.intel.com/products/processor/manuals/.

[47] P. Haffner, S. Sen, O. Spatscheck, D. Wang, acas: Automated construction of application signatures, in: Proceedings of the ACM Workshop on Mining network data, Philadelphia, PA, 2005.

[48] J. Ma, K. Levchenko, C. Kreibich, S. Savage, G. M. Voelker, Unexpected means of protocol inference, in: Proceedings of the Internet Measurement Conference, Rio de Janeiro, Brazil, 2006.

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

[50] M86 Security Labs, Megad analysis, http://www.m86security.com/labs/spambotitem.asp?article=896 (March 2009).

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

[52] Apache web server, http://httpd.apache.org.

[53] Bind, http://www.isc.org/software/bind.

[54] filezilla, http://filezilla-project.org/.

[55] pidgin, http://www.pidgin.im/.

[56] tinyicq, http://www.downv.com/Windows/download-Tiny-ICQ-10064167.htm.

[57] Wireshark, http://www.wireshark.org/.

[58] J. Lim, T. Reps, B. Liblit, Extracting output formats from executables, in: Proceedings of the Working Conference on Reverse Engineering, Benevento, Italy, 2006.

[59] Z. Lin, X. Zhang, Deriving input syntactic structure from execution, in: Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering, Atlanta, GA, 2008.

[60] G. Ramalingam, J. Field, F. Tip, Aggregate structure identification and its application to program analysis, in: Proceedings of the Symposium on Principles of Programming Languages, San Antonio, TX, 1999.

[61] Z. Lin, X. Zhang, D. Xu, Automatic reverse engineering of data structures from binary execution, in: Proceedings of the Network and Distributed System Security Symposium, San Diego, CA, 2010.

[62] C. Leita, K. Mermoud, M. Dacier, scriptgen: An automated script generation tool for honeyd, in: Proceedings of the Annual Computer Security Applications Conference, Tucson, AZ, 2005.

[63] D. Crocker, P. Overell, Augmented bnf for syntax specifications: abnf, RFC 4234 (Draft Standard) (October 2005).

[64] J. Newsome, D. Brumley, J. Franklin, D. Song, Replayer: Automatic protocol replay by binary analysis, in: Proceedings of the ACM Conference on Computer and Communications Security, Alexandria, VA, 2006.

[65] J. Kannan, J. Jung, V. Paxson, C. E. Koksal, Semi-automated discovery of application session structure, in: Proceedings of the Internet Measurement Conference, Rio de Janeiro, Brazil, 2006.