Prospex: Protocol Specification Extraction

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

[*]*Vienna University of Technology*
{pmilani,gilbert}@seclab.tuwien.ac.at
[†]*University of California, Santa Barbara*
chris@cs.ucsb.edu
[‡]*Institute Eurecom*
engin.kirda@eurecom.fr

Abstract

Protocol reverse engineering is the process of extracting application-level specifications for network protocols. Such specifications are very useful in a number of security-related contexts, for example, to perform deep packet inspection and black-box fuzzing, or to quickly understand custom botnet command and control (C&C) channels. Since manual reverse engineering is a time-consuming and tedious process, a number of systems have been proposed that aim to automate this task. These systems either analyze network traffic directly or monitor the execution of the application that receives the protocol messages. While previous systems show that precise message formats can be extracted automatically, they do not provide a protocol specification. The reason is that they do not reverse engineer the protocol state machine.

In this paper, we focus on closing this gap by presenting a system that is capable of automatically inferring state machines. This greatly enhances the results of automatic protocol reverse engineering, while further reducing the need for human interaction. We extend previous work that focuses on behavior-based message format extraction, and introduce techniques for identifying and clustering different types of messages not only based on their structure, but also according to the impact of each message on server behavior. Moreover, we present an algorithm for extracting the state machine. We have applied our techniques to a number of real-world protocols, including the command and control protocol used by a malicious bot. Our results demonstrate that we are able to extract format specifications for different types of messages and meaningful protocol state machines. We use these protocol specifications to automatically generate input for a stateful fuzzer, allowing us to discover security vulnerabilities in real-world applications.

1. Introduction

Reverse engineering is the process of analyzing a device or a system to understand its structure and functionality.

In the context of network protocols, reverse engineering describes the process of deriving the application-level protocol specification of an unknown protocol. To this end, an analyst can monitor the exchange of messages over a network or observe how the communication end-points (such as client and server) process network input. The detailed knowledge of a protocol specification is important for addressing a number of security problems.

Given a protocol specification, it can be used to generate protocol fuzzers [1] that perform black-box vulnerability analysis of network applications. In fact, many vulnerabilities have been found in the past that resulted from programming errors in protocol parsing code [2]. Moreover, detailed protocol specifications are required by intrusion detection systems (e.g., Bro [3]) that perform deep packet inspections. Also, the ability to generate protocol specifications is useful for generic protocol analyzers that require protocol grammars as input (e.g., binpac [4] and GAPA [5]). Furthermore, protocol reverse engineering can help to identify (subtle) variations in the way that different applications implement the same protocol. These differences can be used for application fingerprinting [6] or to discover security vulnerabilities [7]. Finally, the analysis of malware is another important area where protocol reverse engineering can be applied. Botnets [8] increasingly make use of non-standard communication protocols [9], [10]. For a security analyst who attempts to understand and take down botnets, the ability to automatically reverse engineer the command and control protocol is clearly helpful.

In general, reverse engineering is largely a manual, tedious, and time-consuming process. To support a human analyst with this task, a number of automatic protocol reverse engineering techniques have been proposed. These techniques aim to automatically generate the specifications of an application-level protocol. Two possible input sources can be used to analyze a protocol: network traffic and an application that implements the protocol.

A number of approaches [11]–[14] have been presented that use network traffic as input. These systems typically analyze traces generated by recording the communication

between a client and a server. Then, heuristics are applied to extract different protocol fields and delimiters. Although useful in practice, the precision of these systems is often limited. That is, it is not always possible to extract all required information about a protocol from the network traffic alone. To address the limited precision of techniques that operate directly on the network traces, a number of systems [15]–[18] were introduced that focus on the (server) application. More precisely, these systems operate by observing the execution of the application while it is processing input messages. This allows them to infer the structure of a message (i.e., its constituent fields) with higher precision, and it provides insight into field semantics that are not available to network-trace-based approaches. A common property of *all* previous systems (whether network- or behavior-based) is that they only extract the format of individual protocol messages.That is, these systems do not aim at reverse engineering the protocol state machine, and, therefore, cannot produce specifications for *stateful network protocols*.

In this paper, we introduce Prospex (Protocol Specification Extraction), a system that can automatically infer specifications for stateful network protocols, i.e. including state machine information. To the best of our knowledge, this is the first system with this capability. Our analysis builds upon a system introduced in previous work [17], which can extract the format specifications of individual messages by monitoring the application as it processes its inputs. For this paper, our system was extended in two main directions. First, we developed a mechanism to identify messages of the same type. This information is leveraged to combine similar messages into clusters. The second extension is related to the inference of a protocol state machine. The protocol state machine encodes all sequences of messages that are permitted by the protocol. Information about the state machine is required to be able to engage in a "meaningful" conversation with a communication partner, e.g., knowing when a certain message can be sent.

In summary, the contributions of this work are the following:

- We propose several features to determine when two messages in a network session are similar. These features take into account not only the format of messages, but also the effect that receiving each message has on server execution. This allows us to automatically identify and cluster messages of the same type. Automatically recognizing different message types allows us to use a set of messages of the same type to generate a corresponding message format specification.
- We present a technique to automatically infer the protocol state machine. This state machine specifies the order in which messages can be exchanged, given no prior knowledge about the protocol under analysis. We fur-

ther show that our technique consistently outperforms existing approaches for state machine inference.

- We applied our system to a number of real-world applications that implement complex, stateful protocols. The results demonstrate that our techniques are capable of extracting meaningful message formats and protocol state machines. This is true both for protocols used by benign applications (such as SMTP, Samba, or SIP) and protocols used by malicious software (such as the C&C protocol used by Agobot).
- We leverage the output of our system to automatically produce protocol specifications for the open-source Peach fuzzing platform [19]. To this end, we actively contributed to the development of Peach and extended its support for stateful protocols. Running Peach with our specifications allowed us to automatically find vulnerabilities in real-world applications.

2. System Description

The input to our system are a number of application sessions. An application session is a connection between hosts that allows the involved machines to exchange data. Each session typically consists of a sequence of *messages*. Each of these messages has a message *type*, which is defined by a message format specification. The message format specifies the structure of a message, typically as a number of *fields*. The structure of the whole application session is determined by the *protocol state machine*. The protocol state machine defines the order in which messages of different types can be sent.

The objective of our system is to automatically infer the specification of an unknown protocol that a client uses to communicate with a server. More precisely, given a sequence of messages that a client sends to a server, we are interested in the specifications of these messages, as well as the protocol state changes that these messages result in.

To this end, our system proceeds in several phases, as shown in Figure 1.

Dynamic taint analysis. In this phase, dynamic data tainting is used to observe the application as it processes incoming messages. The resulting execution traces show the operations performed on data that was read from the network.

Session analysis. Initially, we analyze these execution traces, splitting them into individual messages. Then, we perform *message format inference* on each message, resulting in detailed format specifications for single messages.

Message clustering. We extract a number of features for each message from the execution trace. These features take into account the previously inferred message formats as well as the effect of each message on the application's behavior. The similarity between messages is determined using these features. Then, we apply the partitioning around medoids
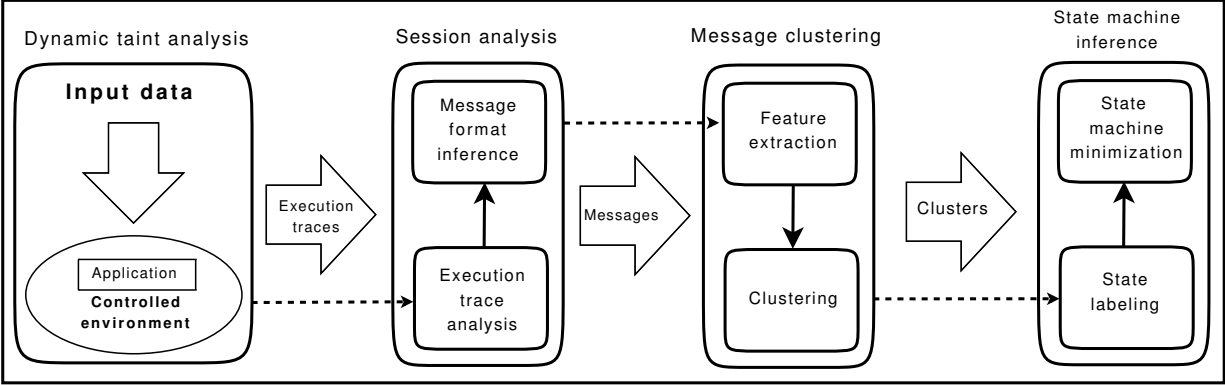
Figure 1. System overview.

(PAM) clustering algorithm [20] to group similar messages into types. Finally, we derive a generalized message format specification for each type.

State machine inference. In this phase, we infer a state machine that models the order in which messages of different types may be sent. Initially, we construct a state machine that accepts exactly the sequences of messages observed during training. Then, we use a novel algorithm, based on domain-specific heuristics, to label the states of this state machine. States that may be similar are assigned identical labels. Finally, we apply the Exbar [21] algorithm to produce a more general, minimal state machine by merging similar states. Together with the generalized message format specifications for the different types of messages, this minimal state machine represents the reverse-engineered network protocol.

Fuzzing. Optionally, our tool can translate the extracted protocol specifications into input for the Peach fuzzing platform [19]. As we will show in Section 3, this allows Peach to test code that is only accessible in later protocol stages, finding "deeper" security vulnerabilities in real-world applications.

2.1. Session Analysis

The purpose of the session analysis phase is to automatically retrieve the message format specifications for the messages that are passed between client and server within an application session. To this end, we leverage an observation that was exploited in previous work [15], [17] for the analysis of individual messages. This observation suggests that it is advantageous to monitor how a program *processes* its input messages instead of analyzing the traffic that is exchanged between hosts at the network-level, because this allows more precise inference of message formats. By using dynamic taint analysis [22]–[24], we can precisely track how the application, which "understands" the messages and implements the protocol state machine, handles input data. As a result of the session analysis phase, we obtain a sequence

of messages for an application session, each represented as a tree of high-level messages fields. Previous work on message format inference includes systems that analyze either single inputs [15], [16] or, more generally, multiple inputs [17], [18]. Our current implementation extends work from [17] and automatically splits sessions into sequences of messages. Thus, the need for human assistance during this phase is removed.

Like previous work, our system is not capable of reverse engineering encrypted traffic. While this is an intrinsic limitation of approaches that only analyze network traffic, it could be overcome by systems that observe server behavior. For this, one could manually or automatically [25] identify buffers that hold decrypted messages and then use these buffers as a starting point for the analysis.

The following paragraphs outline the steps performed by our system to retrieve the message formats.

Recording execution traces. Initially, we execute the application that implements the protocol that we are interested in (e.g., a server program). The program is run in a controlled environment that supports dynamic data tainting [22]–[24]. This allows us to record all operations that involve data read from protocol messages. Then, we engage the server in a series of application sessions, typically by connecting with a client program, performing some common tasks.

In this paper, we limit ourselves to the analysis of the communication in a single direction. That is, we infer the protocol state machine only for one of the communication partners. Also, we only determine the specifications of the messages that this communication partner receives. For ease of presentation, we will refer to this communication partner as "server," and to the other as "client." Note that it would be possible to use our techniques to simultaneously monitor both the client and the server, eventually combining the two different state machines and sets of message formats.

With dynamic data tainting, the system assigns a unique label to each input byte and tracks the propagation of these labels throughout the execution of the program. The output

of this step is, for each application session, an execution trace that contains all executed instructions as well as the taint labels of all instruction operands.

Splitting a session into messages. The execution traces that we record contain all instructions that are executed during an application session. As the next step, this trace needs to be split according to the individual messages. Since we assume no prior knowledge of message boundaries, we use a simple heuristic: The first message starts with the first input byte that the server receives. All subsequent input is considered to be part of this message. This continues until the server writes data to the socket from where it had received the input (that is, the server sends a reply). The next byte received from the client is considered to denote the start of the next message. This is repeated until all execution traces are split into segments, where each segment corresponds to one message. While this approach is not fully general, it is significantly more accurate than considering each network packet as a message by itself. The reason is that clients sometimes break a message into several packets (for example, interactive protocols). The server collects these packets until a complete message has arrived before a response is sent. Our heuristic correctly handles this case and combines multiple packets into a single message.

Inferring message formats. Once our system has identified an execution trace segment for each protocol message, we use the techniques presented in previous work [15], [17] to determine the format of each message. Using these techniques, we analyze an execution trace segment and split the corresponding message into fields. As a result, each message is represented as a tree of fields with associated semantics (i.e., delimited field, length field, pointer field, keyword, file name, etc.). The output of the session analysis step is a sequence of messages for each application session. Each message is represented as a tree of nested fields.

2.2. Message Clustering

After the session analysis phase, the system has extracted a format specification for every individual message. However, there is no information about similarities between messages or their types.

Previous systems that perform message format inference operate on messages of a *single*, known type. However, our goal is to infer a protocol specification assuming no prior knowledge about message types (in fact, we do not even know *a priori* how many message types there are for a certain protocol). Therefore, we require a step that can recognize the types of different messages.

Thus, the goal of the message clustering phase is to assign a type to each message. To this end, we define a metric of similarity between messages, and use it to cluster together similar messages. Our similarity metric is based on the assumption that messages of the same type share similar message formats *and* that the server "reacts" in a similar fashion upon receiving them. Thus, in addition to comparing message formats, we propose a number of similarity features that are based on the analysis of the application's actions as it processes different messages. Once all similar messages are clustered, we label each cluster (and all corresponding messages) with a type.

By assigning types to messages, we can operate on a more abstract representation of protocol sessions. Moreover, for each cluster, we generate a generalized message format specification that describes all messages in this cluster.

2.2.1. Feature Extraction and Similarity Computation. To be able to cluster related messages, we require a way to assess their similarity. For this, we introduce a number of features and corresponding distance functions that allow our system to calculate the similarity between two messages. These features can be divided into three groups that are discussed in the following paragraphs. For each feature, we compute a normalized similarity score between 0 (meaning completely different) and 1 (meaning identical).

Input similarity. Clearly, when comparing two messages, the structure and order of the fields that these messages are composed of play an important role. That is, we would assume that two messages of the same type also contain similar fields in a similar order. To capture this intuition, we use a sequence alignment algorithm (the Needleman-Wunsch algorithm [26], to be more precise). The goal of this algorithm is to take two sequences as input and find those parts that are similar, respecting the order of the elements. These similar parts are then aligned, exposing differences or missing elements in the sequences. In our case, we use the sequences of fields that each message is composed of. For more details on how the comparison between two messages is implemented, we refer the reader to [17].

Execution similarity. In addition to the format of the input messages, we also expect that messages of the same type are handled by similar code in the application. That is, when a message of a certain type is received and processed, we assume that the program uses the same code fragments, library calls, and system calls, at least to a certain degree. This intuition is captured by the following execution similarity features, which can be directly derived from the recorded execution traces.

- *System call feature*: This feature takes into account the types of system calls (as indicated by their system call number) that were invoked during the processing of a message. These system calls are stored as a set, that is, the order is not taken into account for this feature.
- *Process activity feature*: This feature is related to the system call feature, but focuses on system calls related to the generation or destruction of processes (such as **clone**

and **kill**). Process-activity-related system calls are typically very indicative for the behavior of an application. When considered together with all other system calls (as part of the previous feature), these calls would not have sufficient weight in the similarity calculation.

- *Invoked function feature*: For this feature, we store in a set the target addresses of `call` operations that are executed during the processing of a message (if these addresses are within the application's text segment).
- *Invoked library functions feature*: This feature is used to track (dynamically-linked) library calls that are made by an application. To capture this feature, we record the target addresses of `call` operations that are outside the program's text segment. In the case of statically-linked binaries, we would recognize a library call as a regular function call.
- *Executed addresses feature*: For this feature, we use the set of addresses of instructions that are executed by the application while it is processing a specific message (if these addresses are within the application's text segment).

For each of the execution features listed above, we record a set of numbers or addresses that are associated with a certain message. To compare two messages, we employ the Jaccard index [27] to determine the similarity between two features, defined as:

$$J(a, b) = \frac{|a \cap b|}{|a \cup b|}$$

In the equation above, a is the set of elements associated with a feature of the first message, while b is the set that represents the same feature of the second message. Clearly, $J(a, b)$ yields 0 when the sets are disjoint and 1 when they are identical. Note that we calculate five execution similarity scores, one for each of the five execution features.

Impact similarity. The last group of features captures the response of the server to a message that is received. Typically, a server application will execute a series of actions when receiving a (legitimate) request from a client. The goal of the following two impact similarity features is to represent some of these actions at a high level of abstraction.

- *Output feature.* This feature captures the output behavior of the server while processing a message. In particular, we are interested in all system calls that cause the server to write out data. More precisely, we consider the following four destinations for data write operations: the socket to which the client is connected, other network sockets, files, and the terminal. The socket to which the client is connected captures cases in which data is returned to the client (thus ending the message, as explained in Section 2.1), while other network sockets refer to cases in which a server sends data over a different connection. File and terminal destinations simply represent operations where the application writes data to one of these sinks.

For each write operation, we also analyze the taint status of the data that is written. This allows us to distinguish be-

tween operations that write tainted data (i.e., data previously received from the client) and those that write other data. We then label each byte of output with a tuple ⟨*sink, tainted*⟩ that specifies where the data was written to and whether it was tainted or not. The output feature is represented by a sequence of such tuples, with consecutive duplicates removed.

Finally, we use the Needleman-Wunsch sequence alignment algorithm to compare output sequences, as for the input similarity. The result is the output similarity score.

- *File system feature.* This feature captures the file system activity of the server when handling an input message. Therefore, we consider system calls that perform file system actions, such as opening or closing a file, or obtaining information about a file or directory. In a first step, we represent the file system activity as a set of ⟨*operation, path*⟩ tuples, where operation is one of {**open**, **close**, **read**, **write**, **rename**, **stat**, **mkdir**, **rmdir**}, and path is the path of the file that the system call operates on.

The name of the file or directory on which a system call operates may be specific to the individual execution trace, and, therefore, needs to be generalized. Specifically, we look for prefixes of the path that are either hardcoded in the binary (by scanning for strings in the program's file on disk) or are found in one of the program's configuration files (if provided by an analyst). We can also detect those parts of a path that are tainted, and as such, represent a parameter of a client request. To perform generalization of a path, we first attempt to look for the longest prefix that matches a string that is found in the binary. The remaining parts are then replaced with one of the special tokens *TAINT*, *CONFIG*, or *VARIABLE*. The VARIABLE token is used for all parts that are neither tainted nor appear in a configuration file.

As a result of the previous step, the file system feature is represented by a set of tuples, for example, ⟨open, *"/CONFIG/TAINT"*⟩ or ⟨write, *"/var/log/samba/VARIABLE"*⟩. To compute the similarity measure between the file system features of two messages, we use the Jaccard index, as for the execution similarities.

2.2.2. Clustering. Based on the features and similarity functions described in the previous section, we compute the distance between a pair of messages a and b as $d(a, b) = 1 - \sum_i w_i s_i(a, b)$, where s_i is the similarity measure for feature i, and $\sum_i w_i = 1$. The weights w_i are selected in such a way that each of the three groups of features described above has the same overall weight of $\frac{1}{3}$, and that features in the same group have equal weight. Once a distance matrix is computed, we can use off-the-shelf clustering techniques to classify our data. Specifically, we employ the partitioning around medoids (PAM) algorithm [20].

Like most partitioning-based clustering techniques, the PAM algorithm takes as input the number k of clusters to generate. To determine a suitable value for k, we employ a

generalization of the Dunn index. The Dunn index [28] is a standard intrinsic measure of clustering quality, defined as:

$$D(k) = \frac{min_{1 \leq i \leq k}\{\min_{1 \leq j \leq k}\{\delta(C_i, C_j)\}\}}{max_{1 \leq i \leq k}\{\Delta(C_i)\}} \qquad (1)$$

where $C_1, .., C_k$ are the clusters, $\Delta(C_i)$ is the diameter of cluster C_i, and $\delta(C_i, C_j)$ is the distance between the two clusters. Since the numerator of Equation 1 is a measure of cluster separation, while the denominator is a measure of cluster compactness, k should be chosen so that the Dunn index is maximized. To compute the distance between two clusters (δ in Equation 1), we use the single-linkage distance defined as $\delta(C_i, C_j) = min_{a \in C_i, b \in C_j}\{d(a, b)\}$. To compute the diameter of a cluster (Δ in Equation 1), we use one of the measures defined in [29], which is based on *Relative Neighborhood Graphs*. Once clustering has been performed, we derive a format specification for each message type by merging the formats of all messages in the corresponding cluster. This merging step leverages techniques from [17].

2.3. State Machine Inference

The previous clustering phase identifies similar messages in application sessions, assigning a different type to each cluster. As a result, each session s_i can be represented as a sequence $S_i = (\sigma_1, .., \sigma_h)$, where $\sigma_1, .., \sigma_h \in M$ and M is the set of message types. The goal of the state machine inference phase is to infer an acceptor machine that can recognize sequences of message types that represent valid sessions of the protocol under analysis. Unfortunately, this problem cannot be solved exactly, even if we assume that the language comprising all valid sequences is a regular language. The reason is that Gold [30] has proved that a regular language cannot be learned from positive examples only. Moreover, the problem is even more difficult for more powerful languages.

A commonly-used approach to infer a regular language from a labeled training set (a labeled training set is a set of example strings, labeled *accept* or *reject*), is to find the smallest automaton that is consistent with that training set [31]. Such an approach selects the simplest, most-generic hypothesis consistent with the observations. Unfortunately, this technique cannot be directly applied to our problem. The reason is that only positive examples are available (all sessions are labeled *accept*), therefore, the minimal automaton consistent with our training set accepts all possible sequences of message types. To avoid such an over-generalization, we need to restrict the hypothesis space using domain-specific knowledge.

2.3.1. Augmented Prefix Tree Acceptor (APTA). As mentioned previously, the input to the state machine inference phase is a set Λ of message sequences S_i, where each S_i represents one observed application session. In a first step,

we can represent the set Λ as an *augmented prefix tree acceptor* (APTA) T [31].

An APTA is an incompletely-specified deterministic finite state automaton (DFA), with a state transition graph that is a tree. The root of the tree is the initial state of the DFA, and each branch represents an application session. As an example, consider that we observe two application sessions of the Agobot malware. The sequences of the message types of these two sessions are: (**login**, **bot.dns**, **bot.status, mac.logout**) and (**login**, **mac.logout**, **login**, **bot.status**, **bot.dns**, **mac.logout**). The APTA for this example is shown in Figure 2. States of T may be labeled either *accept*, or *reject*. In our example, all states are labeled *accept* (marked with an "A") because any prefix of a valid protocol session is also a valid session (if this were not the case, only the two final states would be *accept* states, and other states would be unlabeled). T is an incompletely-specified acceptor DFA that accepts only the sequences in the training set (and their prefixes). For any other sequence, the result is unspecified.

The APTA T is used as a starting point to find the protocol state machine. This is done by finding the *minimal* DFA that is consistent with T. To find such a DFA, we can leverage existing algorithms (such as Exbar [21]) that start from T and successively merge pairs of states. Clearly, states with different labels can never be merged. However, in our training set, all states of T are labeled *accept*. Thus, the result of directly applying an existing algorithm would be an over-general DFA with only a single state. To address this problem, we introduce an algorithm that assigns different labels to the states of T (discussed in the next Section 2.3.2). This restricts the possible merges, since only states with the same label may be merged. Finally, we use Exbar to obtain a minimal DFA that is consistent with that labeling, as discussed in Section 2.3.3. This minimal DFA represents our state machine.

2.3.2. State Labeling Algorithm. The goal of the state labeling algorithm is to find states in the APTA that are different. By assigning different labels to these states, we can prevent them from being merged. To this end, we leverage the observation that a common pattern in application layer network protocols is that a message or a sequence of messages must be sent before the server can perform certain actions. As an example, in the Agobot command and control protocol, a login is required before other commands become available. In SMB/CIFS, a "TREE CONNECT" operation must be performed to connect to a share before file operations can be issued. In addition, certain commands may lead the server away from a state where it can perform these actions. For instance, a logout command in Agobot or a "TREE DISCONNECT" in SMB/CIFS make previously available commands impossible to execute.
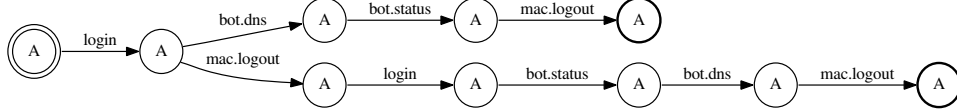
Figure 2. APTA for the Agobot example.

Our state labeling algorithm attempts to identify states that represent similar application conditions. That is, we attempt to identify cases in which an application can process similar commands, based on the sequence of messages that it previously received. To this end, we extract simple patterns from the observed application sessions. These patterns have the form of regular expressions on sequences of message types. More precisely, each pattern has the form:

$$. * r(a_1|..|a_j)*, \quad (r, a_1, .., a_j \in M) \quad (2)$$

where "$*$" means zero or more repetitions of the previous term and "$.$" matches any message type. We call such a pattern a prerequisite.

A prerequisite requires that, for the server to be in a state where it can meaningfully process a message of type m, it must first receive a message of type r, optionally followed only by messages in the set $A_r = a_1, .., a_j$.

The message of type r is a message that always occurs before m. That is, in all application sessions, a message of type r was found before m. This is to capture the case where a connect or login message must be sent before message m. Note that Equation 2 allows messages of any type to occur before r (including more occurrences of r).

The set of optional messages A_r is the set of all messages that, in at least one application session, have been seen between the last occurrence of r and a message of a type m. In the Agobot example, $login$ always occurs before messages in the set $M_{login} = \{bot.dns, bot.status, mac.logout\}$. Furthermore, only $bot.dns$ and $bot.status$ occur between the last $login$ and messages in the M_{login} set. Therefore, the prerequisite $.*login(bot.dns|bot.status)*$ will be added for all three message types in M_{login}. We provide a more precise description of our algorithm for inferring prerequisites in Appendix A.

Once all prerequisites have been computed, we label each state q of T with the set of message types that are allowed as input in that state. A message type m is allowed in a state q if the sequence of message types leading to q exactly matches all prerequisites for m (since T is a tree, there is only one path leading from the root to state q). The labeled state tree for the Agobot example is shown in Figure 3.

Hitting set heuristic. The technique described above fails to detect a prerequisite for a message m when there are multiple, alternative paths to a state where m is allowed. As an example, in an SMTP session, either $HELO$ or $EHLO$ may be the first message, but one of these two is required before a $RCPT\_TO$ message may be sent.

Furthermore, even if there is only one login message type according to the specification of a protocol, this message type may be split into several clusters by our tool (for instance, when the login message can have significantly different, optional parameters). To be able to handle such situations, we generalize Equation 2 and infer prerequisites in the form:

$$. * (r_1|..|r_k)(a_1|..|a_j)* \quad (3)$$

That is, we require only one of the messages $r_1, .., r_k$ to be present in a session before m can be received. To infer such prerequisites, we generalize the algorithm described above, as detailed in Appendix A.

End-state heuristic. In addition to the techniques described previously, we also use a simple heuristic to detect end-states in the protocol state machine. It is common for application layer protocols to have one (or more) message types that request the termination of the protocol session. To detect those message types, we simply look for messages that, throughout all observed application sessions, appear only last in a session. In T, we mark all states that follow such messages as end-states, setting their label to the empty set (since no messages of any type are allowed in those states).

2.3.3. Exbar. Based on a state tree (APTA) that is labeled by our heuristics, we can now infer a minimal DFA. The problem of deriving the smallest DFA consistent with a labeled training set is an important problem in grammar inference, and has been proven NP-complete by Gold [32]. Both approximate and exact algorithms have been proposed to solve it (see [31] for an up to date survey of existing techniques). Exbar [21] is the state-of-the-art, exact algorithm for minimal consistent DFA inference. Thus, we apply Exbar to the state tree T, once it has been labeled by the previously-discussed algorithm. The result is the generalized protocol state machine.

The state machine for the Agobot example (Figure 3) is shown in Figure 4. Here, we have once more replaced the state labels with *accept*. Once this generalization phase is complete, we assume that any sequence of message types that leads to an unspecified state transition is not a valid protocol session. Therefore, we add an additional *reject* state (not shown in Figure 4) to the state machine, and make it the endpoint for all unspecified transitions. In the state machines shown throughout this work, this *reject* state is also omitted for ease of presentation. In Section 3.5, we evaluate the performance of Exbar on our datasets.
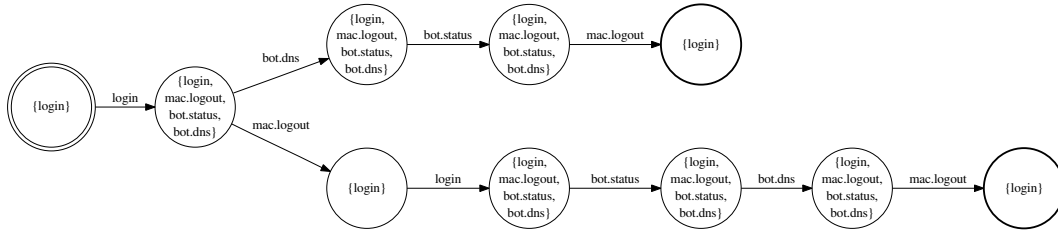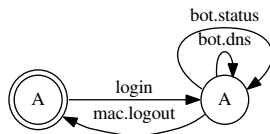
Figure 3. Labeled State Tree for Agobot example.

Figure 4. Inferred state machine for Agobot example.

2.4. Creating Fuzzing Specifications

As a final step, our tool is able to export the state machine and message format descriptions to the XML-based protocol specification format used by the Peach fuzzing platform [19].

Fuzzing is a black-box software testing technique that is based on the principle of feeding an application with random input, while observing crashes or other undesired behavior [33]. To achieve better code coverage of the tested application, advanced fuzzers (such as Peach [19]) generate test data based on the grammar of the file formats or network messages understood by the target application (we refer the reader to [34] for a recent overview of fuzzing techniques). Unfortunately, without any knowledge of the protocol state machine, a (stateful) network protocol cannot be effectively fuzzed. The reason is that a server will typically discard messages with types that are not acceptable in the current protocol state. Thus, stateful protocol fuzzers such as Snooze [35], additionally use a specification of a protocol state-machine to reach deep protocol states.

Prospex is able to automatically extract a grammar for protocol messages, as well as a protocol state machine; stateful, grammar-based fuzzing is, therefore, a natural application. We chose to leverage an existing tool for fuzz testing, and selected Peach [19], mainly because it is an open-source project under active development, and it provides most of the required features. The main limitation of Peach was the limited support for statefulness. To address this limitation, we contributed to the design and development of improved statefulness features for Peach, which have been integrated into release 2.2. To use Prospex specifications for fuzzing, we simply translate the message formats and state machine extracted by our tool to Peach XML. The Peach fuzzing framework then provides all the mechanisms necessary to perform stateful fuzz testing of real-world applications that implement the target protocol.

3. Evaluation

We have tested our tool on a number of applications that implement stateful, application-layer protocols. In particular, we chose a bot protocol, SMB, SMTP and SIP, as they are all stateful protocols implemented in complex server applications that are widely deployed. Because of a limitation of our current system (our taint tool only runs under Linux), we only analyzed server programs that are available to us as Linux binaries. However, this does not represent a general limitation of our approach.

The quality of the specifications produced by our tool is limited by the quality and variety of the data used to train it. As for all trace-based approaches, our system cannot learn behaviors that do not occur in the training data. For the purpose of this evaluation, we trained our system using small datasets that covered a meaningful subset of the functionality of each protocol, such as using SIP to perform phone calls or SMB to browse shared files and folders. The goal of this evaluation is to demonstrate that, provided suitable training data, we can produce accurate state machines and message formats for complex, stateful protocols. Furthermore, our tool can help a human malware analyst to understand a previously-unknown malware protocol. Finally, we show that we can automatically generate fuzzing specifications that are subsequently used to find security vulnerabilities in real-world server programs.

3.1. State Machine Inference

We applied our system to one malware protocol (Agobot C&C), two text-based protocols (SMTP, SIP), and one binary protocol (SMB). For each protocol, the system created state machines that ranged from four states (Agobot) to 13 states (SMB).

Agobot. We selected the well-known Agobot as the malware example. The reason is that Agobot implements a custom C&C protocol and is representative for a whole family of bots, for example, Phatbot and Forbot [36]. For C&C,

Agobot uses a text-based protocol that resembles the IRC protocol. However, the malware author has extended the protocol by incorporating additional command keywords. These commands typically trigger malicious bot behavior, for example, spreading via scanning and remote exploits, relaying traffic, or downloading binaries from the web. The automatic analysis of bots can provide valuable information about the malware's communication protocol and the available commands, which can help an analyst to better and faster understand the bot's internal functionality.

For our experiments, we set up an IRC server and configured an Agobot instance such that the bot connected to a local IRC channel, listening for commands. We then mimicked a bot herder, issuing several commands to the bot while monitoring it. We then ran our tool on the collected traces and obtained the state machine in Figure 5. Moreover, the system has correctly produced format specifications for the commands that we sent to the bot. Of course, for a more realistic scenario, it would be desirable to trace the bot while a real bot herder is issuing commands.
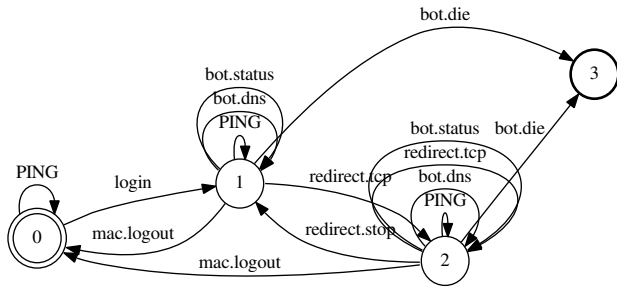

Figure 5. Inferred state machine for Agobot command and control protocol.

SMTP protocol. As an example of an application that implements a stateful, text-based protocol, we have chosen the widely-deployed mail transport agent `sendmail` (version 8.13.8). The application implements SMTP (Simple Mail Transfer Protocol). To infer the state machine, we first recorded 16 SMTP application sessions on our group's e-mail server. We then replayed this small training set to a sendmail server instance that we were tracing. Figure 6 shows the SMTP state machine that our system inferred. It can be seen that two different message types were created for each the MAIL FROM and RCPT TO commands. This is due to the fact that those mail clients that initially send an EHLO command are typically using extended options (additional flags and keywords) in subsequent SMTP commands (for example "ORCPT" in the RCPT TO command). Because of the resulting, different message formats, our system distinguishes between simple and extended versions of these SMTP commands.

Server Message Block (SMB) protocol. As an example of a complex, stateful, binary protocol, we have chosen

SMB/CIFS. In our experiments, we used version 3.0.26a of the `Samba` software suite, and traced the `smbd` daemon while using the `smbclient` utility to browse shared directories, performing common operations such as writing, reading, and deleting files and directories. Using this setup, we produced a training set of 31 recorded sessions. The state machine inferred from the SMB dataset can be seen in Figure 7. The login sequence leading to State 3 is clearly visible. After that, when the DFS (distributed file system) option is enabled, the client first connects to the IPC$ share to obtain a DFS referral for the requested share. Otherwise, the client directly connects to the requested share in State 6. In this state, most of the file system operations are available. Operations on a file are performed by opening the file, reading or writing, and finally closing it (States 8-10). According to this state machine, only one file may be opened at any given time. Of course, this is not a limitation of the SMB/CIFS protocol, but a peculiarity of how the `smbclient` tool employs it. In fact, `smbclient` always closes a file before operating on the next one. Finally, notice that States 11 and 12 are artifacts of our system, caused by the limited variety of the training set. The reason is that, in the training set, "DELETE" and "QUERY DISK" requests were always preceded by find requests. This highlights the dependence of our system on the quality and variety of data in the training set.

Session Initiation Protocol. The text-based Session Initiation Protocol (SIP) [37] is used for setting up and controlling communication connections. In our experiment, we traced the well-known, open-source telephony server `Asterisk` [38] (version 1.4.0), which is typically used as part of a Voice-Over-IP (VOIP) infrastructure. Our test environment consisted of three (virtual) machines, one of them running the Asterisk server, while the two other additional machines served as clients. In our test configuration, we created two SIP peers, each including a voice box. The client machines had installed either CounterPath Corporation's proprietary softphone `X-Lite` [39] (version 2.0) or the open-source softphone `Ekiga` [40] (version 2.11). To simulate different client behavior, the softphones were configured to either automatically answer incoming calls using a built-in auto-answer feature, automatically answer after a short delay (e.g., permit ringing) by using a GUI automation tool [41], or to not answer at all (triggering the voice box). Then, we initiated a number of phone calls to these peers by using a softphone, including simultaneous phone calls on multiple lines. These training calls were used by our tool to generate protocol specifications for the observed call initialization use-cases. Figure 8 shows the SIP state machine.

3.2. Quality of Protocol Specifications

To evaluate the quality of the protocol specifications inferred by Prospex, we need to assess their *soundness*
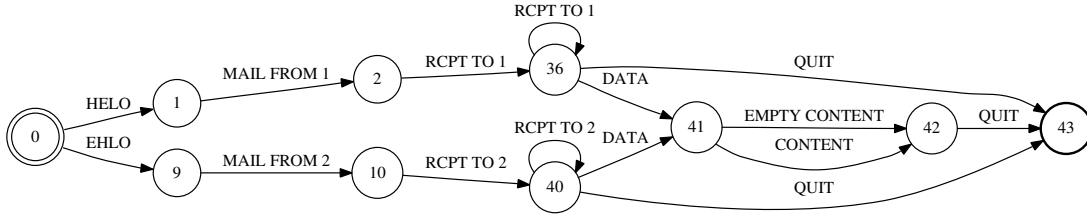
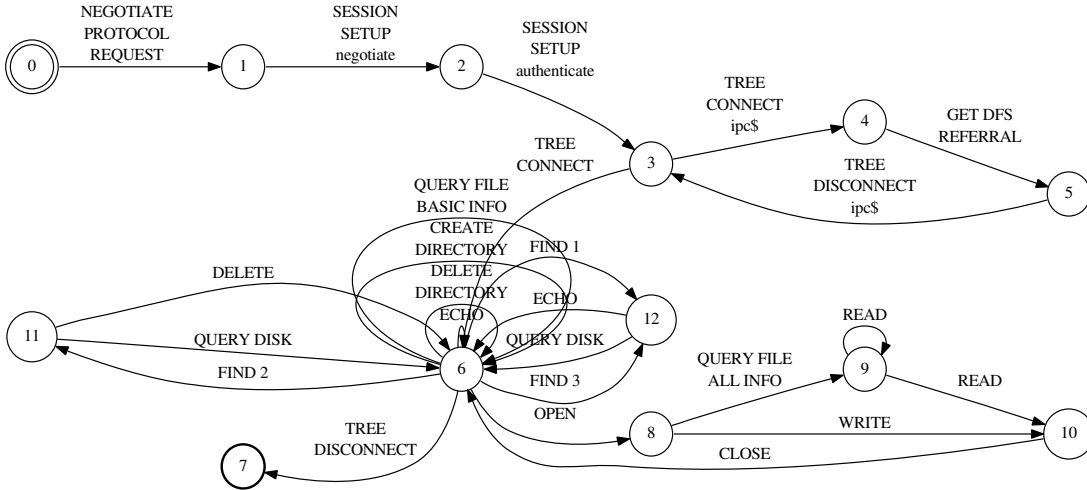Figure 6. Inferred state machine for the SMTP protocol.

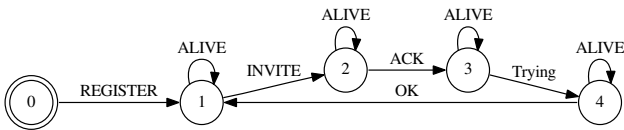Figure 7. Inferred state machine for the SMB/CIFS protocol.

Figure 8. Inferred state machine for the SIP protocol.

and *completeness*. For the purpose of this paper, we say that a protocol specification is *complete* if it is not overly restrictive. That is, the protocol specification accepts (parses) valid protocol sessions. Conversely, we say that a protocol specification is *sound* if it is not overly permissive. That is, it rejects invalid protocol sessions.

As for all trace-based approaches, the *completeness* of inferred specifications is limited by the variety of behaviors observed in the training data. Therefore, we evaluate *completeness* with respect to the subset of protocol functionality that was exercised during training. For instance, for the SMB protocol we take into account the browsing of shared files and directories, but not the use of printing services.

3.2.1. Protocol Completeness. In the first step, we demonstrate that our protocol specifications are complete. To this end, we used our protocol specifications to parse real-world

network traces (that were not part of the training data) of SMTP, SMB, and SIP traffic . Note that this shows the *completeness* of both the message formats and the state machines inferred by our tool. The reason is that, for successful parsing, Prospex has to correctly determine the format of each individual message and recognize their correct ordering.

For parsing, we used an enhanced version of the single message parser presented in [17], which includes support for multiple states. Each result was achieved by using the value of k where the generalized Dunn index reaches its maximum (as discussed in Section 2.2.2).

SMTP results. For SMTP, we recorded our group's Postfix [42] e-mail server traffic (incoming traffic on port 25) during a period of four weeks. Then, we split the dumps into TCP sessions and parsed them, using the automatically-generated SMTP protocol specification with $k = 10$.

Out of 31,903 total sessions, we were able to parse 29,832 sessions (93.5%) successfully. We found that the remaining 2,071 sessions (6,5%) were all using TLS encryption, which we cannot handle properly as one of the limitations of our system is its inability to handle encrypted traffic. This shows that our system can fully parse (unencrypted) real-world traffic, generated by a number of clients and sent to

a different mail server implementation than the one used to infer the protocol specifications.

SMB results. To test our SMB protocol specification, we used `smbclient` to browse shared directories on both Windows and Linux servers, and recorded 80 sessions. For $k = 23$, only 8 sessions fail to parse. We examined these sessions and determined that parsing fails because of (1) error conditions not present in the training set (such as attempting to read from a non-existing file), (2) writing of long files; a limitation of our training set was that only short files were written, small enough to be sent in a single write message, and (3) insufficient generalization of the state machine (as discussed in Section 2.3, states 11 and 12 in Figure 7 are artifacts of our system).

SIP results. For generating SIP traffic, we used `X-Lite` [39] to initiate phone calls to different SIP peers in a laboratory Voice-Over-IP environment. We recorded a set of 80 SIP sessions during these calls. Using the state machine for the indicated optimum of $k = 6$, we were able to parse all of the traffic successfully.

3.2.2. Protocol Soundness. In the next step, we evaluate the *soundness* of the inferred specifications.

Soundness of the Message Formats. To show that our protocol specifications are not overly permissive, we first need to demonstrate that the inferred message format for each cluster is not too general, as it should neither parse arbitrary messages nor messages that have a different type. To this end, we compute the *message format specificity*.

Initially, we manually label every message in the training set with its actual message type (such as "HELO" or "TREE CONNECT"). Then, we mark each cluster with the message label of the majority of its messages (while, ideally, all messages in a cluster would share the same label, this step is necessary if different message types are incorrectly clustered together). In the next step, we select a certain cluster. Then, we find all training set messages that are *not* labeled with the label of this cluster. These messages are then parsed with the cluster's message format. When the clustering phase was successful and the message formats are *sound* (not too general), we would expect most parsing attempts to fail. This step is then repeated for all clusters. Finally, we calculate the ratio r of successfully parsed messages to the number of total parsing attempts. The format specificity is then computed as $1 - r$.

For the value of k where the Dunn index reaches its maximum, our tool achieves a message format specificity of 1 for all four datasets (Agobot, SMTP, SMB and SIP). This means that (a) no cluster contains messages of multiple types, and (b) the message format for a cluster does *not* parse any messages of a different type.

Soundness of the State Machines. The next goal is to evaluate the *soundness* of the inferred state machines. To do

so, we require a reference state machine for each protocol. We created these reference state machines manually, using information from specification documents (if available), and integrating it with our own testing and reverse engineering efforts. Clearly, our tool cannot learn parts of the protocol that were not exercised in our training data, so we do not include them in the reference state machine. We then performed n (for $n = 50,000$) random walks over our inferred state machine, generating n sessions that our specification considers valid. These sessions were fed to the reference state machines. The idea is that an overly permissive state machine would generate sessions that are not recognized by the true protocol. We found that, for all four protocols, all sessions were accepted. Thus, our inferred state machines are sound.

3.3. Comparative Evaluation

The previous section has shown that our techniques allow us to infer accurate specifications. However, there exist alternative approaches to infer an automaton from positive examples only. One popular approach is based on the minimum message length (MML) principle [43]. According to this principle, a solution should minimize the length of the description of the state machine together with the dataset it tries to account for. Since minimizing this quantity is an NP-complete problem, several approximate algorithms have been proposed to attempt to solve it. The sk-strings algorithm [44] is one such algorithm, which has previously been applied to mine specifications [45] of a software component from program execution traces. The authors of [44] also introduced the beams algorithm [46], which outperforms sk-strings. We obtained the implementations of both algorithms from the authors [47].

To compare the performance of previous techniques with our system, we leverage the *precision* and *recall* metrics introduced in [48]. *Precision* is closely related to the *soundness* metric described in the previous section. It measures the ratio of sequences generated by a random walk over the inferred automaton that are accepted by the reference automaton. Conversely, *recall* measures the ratio of sequences generated by the reference automaton that are accepted by the inferred automaton. It is a measure of *completeness*. For details on how these metrics are computed, we refer the reader to [48].

Results are shown in Table 1. For sk-strings, we show results using the OR heuristic (which was the best performer in [44]) and the AND heuristic (which is evaluated in [48]). We run sk-strings with tail lengths of $1, 2$ and 3 and $s = 0.5, 0.75, 1$, and select the best solution based on MML. Similarly, we run the beams algorithm with beam widths of $1, 2, 4, 8, 16$ and 32.

Prospex clearly outperforms previous tools on all four datasets. The sk-strings algorithm using the OR heuristic does not produce *sound* results on most datasets ($P \simeq 0.12$).

Sk-strings with the AND heuristic and beams produce better results. However, only Prospex consistently provides a *sound* state machine ($P = 1$). Previous algorithms over-generalize on at least one dataset. Somewhat surprisingly, neither sk-strings nor beams succeed in learning a state machine for the rather simple Agobot dataset.

	Agobot		SMTP		SMB		SIP	
	P	R	P	R	P	R	P	R
Prospex	1	1	1	1	1	.58	1	1
beams	.56	1	.89	1	1	.50	1	1
skstrings(and)	.79	.20	1	.88	1	.30	1	.01
skstrings(or)	.11	.92	.11	1	.12	.62	1	1

Table 1. Precision (P) and Recall (R) of inferred automata with respect to reference automaton.
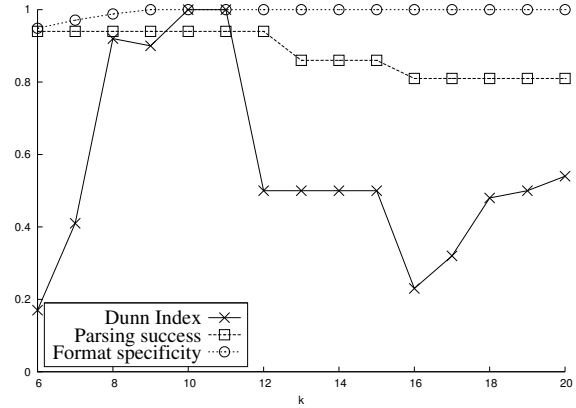
3.4. Robustness of k

In this section, we examine the robustness of our tool to the choice of the parameter k, the number of message types (clusters) that need to be provided as input to the clustering algorithm. The number of clusters is a trade-off between *soundness* and *completeness*. With too many clusters, we expect the inferred model to be *sound* but over-specific, and therefore *incomplete*. Conversely, with too few clusters, we expect a *complete* but over-permissive (*unsound*) model. We wish to demonstrate that, for a relatively large range of values for k, Prospex produces a *sound* and *complete* protocol specification. Therefore, we measure the following two properties over a range of values for k:

Parsing success rate. To compute this property, we use the generated protocol specification for each k to parse network traces of real-world traffic, and measure the ratio of successfully parsed sessions to the total number of sessions. This is a measure of *completeness*, so we expect it to to *decrease* as k increases.

Message format specificity. This is the property introduced in Section 3.2.2. It is a measure of *soundness*, so we expect this property to *increase* as k increases. The reason is that more clusters imply fewer messages per cluster, so the message formats for each cluster become more specific.

In Figure 9, we show both properties for the SMTP and SMB protocols over a range of values for k. In addition, the figures show the generalized Dunn index, which, as described in Section 2.2.2, is used to choose the value of k. The Dunn index is normalized to the $(0, 1)$ range. It can be seen that the maximum of the Dunn index correlates with the optimal choice of k with regard to parsing quality and message specificity. This confirms that the Dunn index is a good predictor to select k, resulting in protocol specifications that are specific *and* successful in parsing.

Specifically, in Figure 9(a), the Dunn index reaches its maximum at $k = 10$. This corresponds to the optimal parsing

(a) SMTP
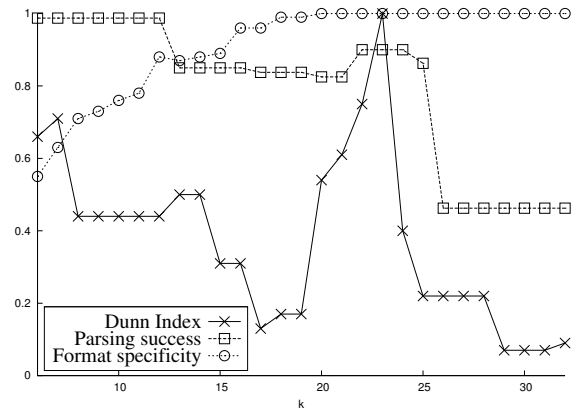
(b) SMB

Figure 9. Robustness of k

results and a message format specificity of 1, which demonstrates the suitability of our approach. Similarly, Figure 9(b) shows that values of k around the choice of 23 (between 22 and 25) produce high parsing results, while having a message format specificity of 1.

3.5. Exbar Performance

We allowed Exbar to run for up to 5 minutes for each value of k, and were able to infer state machines of up to 25 states (starting from state trees with over 200 states). For the optimal values of k, selected using the Dunn index, Exbar terminated in less than 0.03 seconds for all of our test cases. Nonetheless, for very large values of k, Exbar might not terminate in the allotted time. In such cases, approximate algorithms could be used instead [31], but a better solution is to increase the size of the training set, since DFA learning is harder when the training set is sparse (as was empirically demonstrated by the Abbadingo competition [49]).

3.6. Fuzzing Experiments

As discussed in Section 2.4, our system can be used to create input specifications for the Peach fuzzer. This allows the fuzzer to use the automatically inferred state machine while fuzzing the message's field values according to the inferred field types.

SMB fuzzing. We automatically converted the protocol specifications generated by Prospex for the SMB/CIFS protocol into more than 2,100 lines of Peach XML. This allowed us to use Peach to fuzz the latest versions of the Samba server and the Windows XP SMB/CIFS implementation. Unfortunately, we did not find any vulnerabilities in these programs. This may be due to the fact that both are mature, widely-deployed services that have been patched for many vulnerabilities related to input validation errors in the past. Therefore, we target an older version of Samba (version 3.0.2a, which is subject to an arbitrary file access vulnerability [50]) to validate our tool. By searching the network traces captured during the fuzzing run, we were able to verify that the fuzzer had been able to find the vulnerability. More specifically, the fuzzer downloaded the /etc/passwd file, which should not have been accessible through the SMB service. The /etc/passwd file is commonly used to test for file traversal vulnerabilities on UNIX variants. Notice that successful exploitation of this vulnerability requires the fuzzer to navigate deep into the protocol state machine (to State 10 in Figure 7). Furthermore, this attack is only possible because the message format inference has automatically identified a field in the "OPEN" message as a file name. Peach only makes directory traversal attempts on fields that are marked as file names.

SIP fuzzing. We generated fuzzing specifications for SIP and ran the Peach fuzzer on the Asterisk server. After checking the fuzzer logs, we noticed that sending the server an "OK" message with status code "0" triggered a segmentation fault that crashed Asterisk. This could be used to launch a denial-of-service (DOS) attack. For the server to successfully accept and parse the message that crashes it, the fuzzer has to navigate successfully to State 4 in the SIP state machine (shown in Figure 8). Finding vulnerabilities of this kind by using stateless fuzzing is practically infeasible. Even though we found this to be a known vulnerability [51] that has already been addressed in the newest versions of Asterisk, it shows that our system is capable of creating fuzzing specifications that can be used to automatically find vulnerabilities in real-world software.

4. Related Work

Protocol reverse engineering is not a new concept. Since proprietary, closed protocols started to emerge on the Internet (e.g., such as the OSCAR protocol, used by ICQ and AIM [52]), there has been interest to reverse engineer these protocols with the goal of providing free, open-source alternatives. For example, Samba [53] aims to offer a free implementation of the Microsoft SMB/CIFS file sharing protocol. Although popular, protocol reverse engineering is still a largely manual task. It is tedious and labor-intensive.

Session replay. The first automated protocol analysis approaches emerged within the context of honeypots. In order to capture malicious code that delivers its payload after a series of interactions over the network, researchers started working on systems that could replay application sessions automatically. To this end, systems such as RolePlayer [12] and ScriptGen [13], [14] analyze network traffic and attempt to generalize the traces so that correct replies can be generated to new requests. Although useful, the main focus of these systems is not to reverse engineer and understand the entire protocol that is analyzed, but to continue the interaction with a malicious program long enough so that its payload can be intercepted. Hence, these systems only focus on the protocol format to the extent necessary for replay, in particular, on the recognition of fields that contain cookie values or IP addresses. ScriptGen is the only previous work that attempts a kind of state machine inference. However, the proposed technique is limited because no generalization takes place. Thus, the resulting state machine is a tree, similar to the APTA in Section 2.3.1, which can only parse sessions identical to those previously observed.

Protocol analysis. Reacting to the emerging need for the automated analysis and reverse engineering of entire protocols, systems were proposed that attempt to discover the complete protocol format. In [54], the authors propose to apply bioinformatics techniques (such as sequencing algorithms) to network traffic. The goal of the system is to identify protocol structure and fields with different semantics. In [11], an improved technique was proposed that uses recursive and type-based clustering instead of byte-wise alignment. The advantage of such network trace-based approaches is that it is straightforward to gather large datasets for training. Their shortcoming is that network traces provide a limited amount of information and no information on field semantics, making classification of messages into types extremely challenging.

Recently, four approaches were presented that propose to extract protocol information by observing the execution of a program while it processes input messages [15]–[18]. However, these systems focus on reversing message formats, and leave state machine inference for future work.

Specification mining. Automatically extracting a protocol state machine from a set of observed protocol interactions is related to the problem of extracting temporal specifications for software components (such as API or method call sequences) from program traces [45], [55]–[58]. Here, we focus on how work in this field performs state machine

inference. In [55], each relevant event is directly mapped to a state in the automaton. This approach is not suitable for protocol inference, where typically there exist message types that are valid in many different states (such as the "ALIVE" message in Figure 8). In [45], the sk-strings algorithm is used to infer a state machine. As discussed in Section 3.3, this algorithm does not provide acceptable performance for most of our datasets. Other works [56], [57] only infer properties conforming to simple patterns, such as alternation between two events. Finally, [58] uses an active learning approach, and learns state machines using the $L*$ algorithm [59]. The $L*$ algorithm requires a teacher that can answer membership queries. In [58], the teacher is implemented using model checking techniques. This approach cannot be easily applied to network protocol inference.

Automated white-box testing. Performing fuzzing of an application based on an automatically reverse-engineered network protocol is related to concolic testing [60], white-box fuzzing [61], [62], and related approaches [63], [64]. These techniques leverage symbolic execution of a target application to generate test cases that provide better code coverage than black-box fuzzing approaches. They have been successfully applied to a wide variety of software such as Linux file system implementations [65], the entire GNU coreutils [64], and the JavaScript interpreter of Internet Explorer 7 [62]. To the best of our knowledge, none of these tools have yet been applied to real-world implementations of stateful network protocols. Also, we believe that these techniques are complementary to ours. That is, symbolic execution could be added to Prospex to overcome some of its limitations, such as its inability to express arbitrary relationships between protocol fields. Conversely, the protocol specifications generated by Prospex could be used to enhance white-box fuzzing of complex network applications by leveraging a grammar-based constraint solver [62].

5. Conclusions

In this paper, we presented Prospex, a system to automatically extract application layer protocol specifications. Our system monitors the execution of a (server) program that processes network input. Based on the recorded execution traces, the tool produces accurate message format specifications for different types of messages and a generalized protocol state machine.

Our technique proceeds in three main steps: First, we split application sessions into individual messages and extract their formats. The second step is responsible for clustering similar messages. The notion of similarity is established not only by comparing message formats, but also by analyzing the overall behavior of the server in reaction to each input. Based on the clusters, we can assign a type to each message, a process that required manual analysis in previous work.

Finally, the third step infers a generalized protocol state machine that reflects the sequences in which messages may be exchanged.

Our experiments demonstrate that the presented approach works well in practice. Our system can analyze real-world programs, producing specifications for complex protocols such as SMB/CIFS. Moreover, our system is able to help malware analysts by automatically reverse-engineering a non-standard protocol used by a malicious bot program. Additionally, our system can create detailed input specifications for a stateful fuzzer.

References

[1] P. Oehlert, "Violating Assumptions with Fuzzing," *IEEE Security and Privacy*, vol. 3, no. 2, 2005.

[2] R. Kaksonen, M. Laakso, and A. Takanen, "Software Security Assessment through Specification Mutations and Fault Injection," in *IFIP Joint Working Conference on Communications and Multimedia Security (CMS)*, 2001.

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

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

[5] N. Borisov, D. Brumley, H. J. Wang, J. Dunagan, P. Joshi, and C. Guo, "A Generic Application-Level Protocol Analyzer and its Language," in *14h Symposium on Network and Distributed System Security (NDSS)*, 2007.

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

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

[8] D. Dagon, G. Gu, C. Lee, and W. Lee, "A Taxonomy of Botnet Structures," in *Annual Computer Security Applications Conference (ACSAC)*, 2007.

[9] P. Porras, H. Saidi, and V. Yegneswaran, "A Multi-perspective Analysis of the Storm (Peacomm) Worm," Computer Science Laboratory, SRI International, Tech. Rep., 2007.

[10] "Danmec / Asprox SQL Injection Attack Tool Analysis," http://www.secureworks.com/research/threats/danmecasprox/?threat=danmecasprox, 2008.

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

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

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

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

[15] J. Caballero, H. Yin, Z. Liang, and D. Song, "Polyglot: Automatic Extraction of Protocol Format using Dynamic Binary Analysis," in *14th ACM Conference on Computer and Communications Security (CCS)*, 2007.

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

[17] G. Wondracek, P. Milani Comparetti, C. Kruegel, and E. Kirda, "Automatic Network Protocol Analysis," in *15th Symposium on Network and Distributed System Security (NDSS)*, 2008.

[18] W. Cui, M. Peinado, K. Chen, H. Wang, and L. Irun-Briz, "Tupni : Automatic Reverse Engineering of Input Formats," in *ACM Conference on Computer and Communications Security (CCS)*, 2008.

[19] "Peach Fuzzing Platform," http://peachfuzzer.com, 2008.

[20] L. Kaufman and P. Rousseeuw, *Finding Groups in Data: An Introduction to Cluster Analysis*. Wiley, 1990.

[21] K. J. Lang, "Faster Algorithms for Finding Minimal Consistent DFAs," NEC Research Institute, Tech. Rep., 1999.

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

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

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

[25] Z. Wang, X. Jiang, W. Cui, and X. Wang, "ReFormat: Automatic Reverse Engineering of Encrypted Messages," NC State University, Tech. Rep. 2008-26, 2008.

[26] S. Needleman and C. Wunsch, "A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins," *Journal of Molecular Biology*, vol. 48, no. 3, 1970.

[27] P. Jaccard, "The Distribution of Flora in the Alpine Zone," *The New Phytologist*, vol. 11, no. 2, pp. 37–50, 1912.

[28] J. Dunn, "Well Separated Clusters and Optimal Fuzzy Partitions," *Journal of Cybernetics*, vol. 4, 1974.

[29] N. R. Pal and J. Biswas, "Cluster Validation Using Graph Theoretic Concepts," *Pattern Recognition*, vol. 30, no. 6, 1997.

[30] E. M. Gold, "Language Identification in the Limit," *Information and Control*, vol. 10, no. 5, 1967.

[31] M. Bugalho and A. L. Oliveira, "Inference of Regular Languages Using State Merging Algorithms with Search," *Pattern Recognition*, vol. 38, no. 9, 2005.

[32] E. M. Gold, "Complexity of Automaton Identification from Given Data," *Information and Control*, vol. 37, no. 3, 1978.

[33] B. Miller, L. Fredriksen, and B. So, "An Empirical Study of the Reliability of UNIX Utilities," *Communications of the ACM*, vol. 33, no. 12, 1990.

[34] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*, 1st ed. Addison-Wesley, 2007.

[35] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna, "Snooze: Toward a Stateful Network Protocol Fuzzer," in *Proceedings of the 9th Information Security Conference (ISC)*, 2006.

[36] T. Holz, "A Short Visit to the Bot Zoo [Malicious Bots Software]," *Security & Privacy, IEEE*, vol. 3, no. 3, 2005.

[37] "RFC 3261 - SIP: Session Initiation Protocol," http://www.ietf.org/rfc/rfc3261.txt, 2008.

[38] "Asterisk: The Open Source PBX and Telephony Platform," http://www.asterisk.org, 2008.

[39] "X-Lite softphone," http://www.counterpath.com, 2008.

[40] "Ekiga - Free your speech," http://www.ekiga.org, 2008.

[41] "The Linux Desktop Testing Project," http://ldtp.freedesktop.org, 2008.

[42] Wietse Venema, "Postfix," http://www.postfix.org, 2008.

[43] C. Wallace and M. Georgeff, "A General Objective for Inductive Inference," Department of Computer Science, Monash University, Tech. Rep., 1983.

[44] J. Patrick and P. North, "The sk-strings Method for Inferring PFSA," in *Workshop on Automata Induction, Grammatical Inference and Language Acquisition at the 14th International Conference on Machine Learning (ICML97)*, 1997.

[45] G. Ammons, R. Bodík, and J. R. Larus, "Mining Specifications," *SIGPLAN Not.*, vol. 37, no. 1, 2002.

[46] A. Raman, P. Andreae, and J. Patrick, "A Beam Search Algorithm for PFSA Inference," *Pattern Analysis and Applications*, vol. 1, 1998.

[47] "PFSA Toolkit," http://www.cs.usyd.edu.au/~rcdmnl/PFSA.

[48] D. Lo and S. Khoo, "QUARK: Empirical Assessment of Automaton-based Specification Miners," in *13th Working Conference on Reverse Engineering (WCRE)*. IEEE Computer Society, 2006.

[49] K. J. Lang, B. A. Pearlmutter, and R. A. Price, "Results of the Abbadingo One DFA Learning Competition and a New Evidence-Driven State Merging Algorithm," in *ICGI 98: Proceedings of the 4th International Colloquium on Grammatical Inference*. Springer-Verlag, 1998.

[50] "Potential Arbitrary File Access," http://www.securityfocus.com/archive/1/377618, 2004.

[51] "Asterisk DOS Vulnerability," http://secunia.com/advisories/24579, 2007.

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

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

[54] M. Beddoe, "The Protocol Informatics Project," in *Toorcon*, 2004.

[55] J. Whaley, M. C. Martin, and M. S. Lam, "Automatic Extraction of Object-Oriented Component Interfaces," *SIGSOFT Softw. Eng. Notes*, vol. 27, no. 4, 2002.

[56] D. Engler, D. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code," in *ACM Symposium on Operating Systems Principles*, 2001.

[57] J. Yang and D. Evans, "Perracotta: Mining Temporal API Rules from Imperfect Traces," in *28th Internl. Conf. on Software Engineering (ICSE 2006*.

[58] R. Alur, P. Černý, P. Madhusudan, and W. Nam, "Synthesis of Interface Specifications for Java Classes," *SIGPLAN Not.*, vol. 40, no. 1, 2005.

[59] D. Angluin, "Learning Regular Sets from Queries and Counterexamples," *Inf. Comput.*, vol. 75, no. 2, 1987.

[60] K. Sen, D. Marinov, and G. Agha, "CUTE: A Concolic Unit Testing Engine for C," in *ESEC/FSE-13: Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering.* ACM, 2005.

[61] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated Whitebox Fuzz Testing," in *Network and Distributed Security Symposium (NDSS).* Internet Society, 2008.

[62] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based Whitebox Fuzzing," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008.

[63] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: Automatically Generating Inputs of Death," in *13th ACM Conference on Computer and Communications Security (CCS)*, 2006.

[64] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[65] J. Yang, C. Sar, P. Twohey, C. Cadar, and D. Engler, "Automatically Generating Malicious Disks Using Symbolic Execution," in *IEEE Security and Privacy*, 2006.

[66] R. M. Karp, ""Reducibility Among Combinatorial Problems"," in *Complexity of Computer Computations.* Plenum Press, 1972.

Appendix A: Prerequisites Inference Algorithm

In this section, we detail the algorithms that we use to infer the prerequisites described in Section 2.3.2. Algorithm 1, together with Algorithms 2 and 3, computes prerequisites in the form of Equation 2. A prerequisite for message type m is specified by the tuple $\langle m, r, A_r \rangle$. To implement the hitting set heuristic and to infer prerequisites in the form of Equation 3, we replace the call to the `get_required` function in Algorithm 1 with a call to `get_required_sets` (Algorithm 4).

The `hitting_set` function in Algorithm 4 finds a solution to the minimum hitting set problem for sets in Y. That is, it finds the smallest set ρ of message labels such that $\rho \cap y \neq \emptyset, \quad \forall y \in Y$. That is, we want to find the minimum number of message types such that at least one type is present on any path from the start state to a state where m is received. The minimum hitting set problem is NP-complete [66], but we impose the restriction $|\rho| \leq K$ and solve it by exhaustive search. The constant K was set to 5 in our experiments; we do not expect a protocol specification to have more than 5 message types leading to the same state transition, or our clustering algorithm to be so inaccurate as to split messages of a single message type into more than 5 clusters. Since `get_required_sets` returns a set of sets, Algorithms 1 and 3 must also be modified accordingly.

Algorithm 1 `infer_prerequisites`

Input: The set of message types M. The training set composed of n application sessions $S_1, .., S_n$, where $S_i = \sigma_{i,1}, ..\sigma_{i,|S_i|}$ and $\sigma_{i,j} \in M$.

Result: the set of prerequisites P.

  for each $m \in M$ **do**
    $R_m = get\_required(m)$
  $R = \bigcap_{m \in M} R_m$
  $P = \emptyset$
  for each $r \in R$ **do**
    $M_r = m \in M | r \in R_m$ // the set of messages which share requirement r
    $A_r = get\_allowed(r, M_r)$
    for each $m \in M_r$ **do**
      add $\langle m, r, A_r \rangle$ to P

Algorithm 2 `get_required`

Input: A message type $m \in M$. The training set $S_1, .., S_n$.

Result: $R_m \subset M$, the set of msg. types required before m

  $Y = \emptyset$
  for each instance $\sigma_{i,j}$ of m in the training set **do**
    $y = \{\sigma_{i,1}, .., \sigma_{i,j-1}\}$
    // the set of message types found before $\sigma_{i,j}$ in S_i
    add y to Y
  return $\bigcap_{y \in Y}$

Algorithm 3 `get_allowed`

Input: A message type $r \in M$. A set of message types $M_r \subset M$. The training set $S_1, .., S_n$.

Result: $A \subset M$, the set of message types allowed after r

  $A = \emptyset$
  for each instance $\sigma_{i,j}$ of m in the training set **do**
    consider the application session $S_i = \sigma_{i,1}, .., r, \sigma_{i,k+1}, .., \sigma_{i,j}, ..$ // $\sigma_{i,k} = r$ is the last occurrence of r in S_i before $\sigma_{i,j}$
    $a = \{\sigma_{i,k+1}, .., \sigma_{i,j-1}\}$
    add a to A
  return A

Algorithm 4 `get_required_sets`

Input: A message type $m \in M$. The training set $S_1, .., S_n$.

Result: $R_m \subset 2^M$, the set of requirements for m

  $R_m = \emptyset$
  compute Y as in get_required
  while $(y \neq \emptyset \quad \forall y \in Y)$ **do**
    $\rho = hitting\_set(Y)$
    add ρ to R_m
    set y to $y - \rho$ for each $y \in Y$
  return R_m