Inference and Analysis of Formal Models of Botnet Command and Control Protocols

Chia Yuan Cho Domagoj Babić Eui Chul Richard Shin Dawn Song

University of California, Berkeley
{chiayuan@cs,babic@cs,ricshin,dawnsong@cs}.berkeley.edu

ABSTRACT

We propose a novel approach to infer protocol state machines in the realistic high-latency network setting, and apply it to the analysis of botnet Command and Control (C&C) protocols. Our proposed techniques enable an order of magnitude reduction in the number of queries and time needed to learn a botnet C&C protocol compared to classic algorithms (from days to hours for inferring the MegaD C&C protocol). We also show that the computed protocol state machines enable formal analysis for botnet defense, including finding the weakest links in a protocol, uncovering protocol design flaws, inferring the existence of unobservable communication back-channels among botnet servers, and finding deviations of protocol implementations which can be used for fingerprinting. We validate our technique by inferring the protocol state-machine from Postfix's SMTP implementation and comparing the inferred state-machine to the SMTP standard. Further, our experimental results offer new insights into MegaD's C&C, showing our technique can be used as a powerful tool for defense against botnets.

Categories and Subject Descriptors

F.1.1 [**Computation by Abstract Devices**]: Models of Computation—*automata*; I.2.6 [**Artificial Intelligence**]: Learning—*concept learning*; C.2.2 [**Computer-Communication Networks**]: Network Protocols—*applications, protocol verification*; D.4.6 [**Operating Systems**]: Security and Protection—*invasive software*

General Terms

Security, Algorithms, Experimentation, Performance

Keywords

Protocol Model Inference and Analysis, Response Prediction

1. INTRODUCTION

Protocol inference is a process of learning the inner workings of a protocol through passive observation of the exchanged messages or through active probing of the agents involved in the message exchange. Even for small simple protocols, manual protocol inference (a.k.a. reverse-engineering) is tedious, error-prone, and time-consuming. Automatic protocol inference sounds attractive, but poses a number of technical challenges, for some of which we propose novel solutions in this paper.

The applications of protocol inference are numerous. The main application we are interested in is the inference of botnet protocols. Botnets are the primary means through which denial of service attacks, theft of personal data, and spamming are committed, causing billions of dollars of damage annually [1]. Defeating such botnets requires the understanding of their inner workings, i.e., their communication protocols. The second application we are interested in is inferring models of implementations of frequently used protocols. While public standards are often available for such protocols, implementations rarely strictly follow the standard, either due to bugs in the implementation, pitfalls in understanding the standard, or ambiguities in the standard itself. In this setting, automatic protocol inference technology can be used for fingerprinting and checking adherence to the standard. Other possible applications of protocol inference include: automatic abstraction of agents participating in message exchange for assume-guarantee-style verification of protocols, fuzz testing of protocol implementations (e.g., [10]), and reverse-engineering of proprietary and classified protocols.

Our main motivation is to provide the security community with new techniques and tools to fight botnets. The technology we have developed is a powerful weapon against botnets, enabling automatic inference of protocol models that can be used by both human analysts and automatic tools. After presenting our main technical contribution, we propose a number of analyses of inferred models: identification of protocol components most susceptible to disruptive attacks (e.g., for the purpose of finding the most efficient way of bringing down a botnet), identification of protocol design flaws, detection of back-channel communication to which the analyst has no direct access, and detection of differences among protocol implementations.

1.1 A Brief Overview of Protocol Inference

The inner workings of a protocol can be modelled in a number of ways, but state-machine models are by far the most standard. Thus, the problem of protocol inference is isomorphic to the problem of learning a state-machine describing the protocol. Modelling protocols with finite and infinite state-machines is a complex topic,

This material is based upon work partially supported by the National Science Foundation under Grants No. 0311808, No. 0448452, No. 0627511, and CCF-0424422, by the Air Force Office of Scientific Research under Grant No. 22178970-4170, by the Army Research Office under grant DAAD19-02-1-0389, and by the Office of Naval Research under MURI Grant No. N000140911081. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation, the Air Force Office of Scientific Research, the Army Research Office, or the Office of Naval Research.

CCS'10, October 4–8, 2010, Chicago, Illinois, USA.
Copyright 2010 ACM 978-1-4503-0244-9/10/10 ...$10.00.

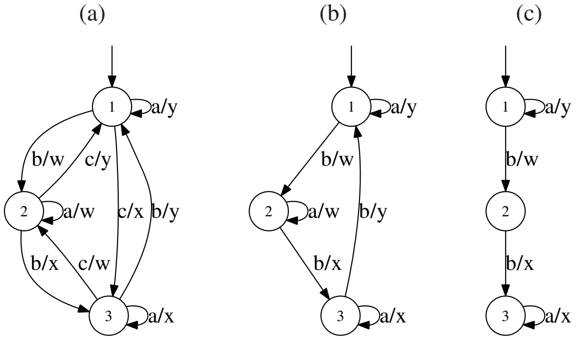


Figure 1: Illustration of the difference between a complete model with respect to the entire protocol alphabet (on the left), complete state-machine with respect to a given subset of messages (in the middle), and an incomplete state-machine (on the right).

but only two dimensions of the problem are relevant to this paper. The first dimension is the *finiteness* of the state-machine. In this paper, we focus on inferring finite state-machine models, both in terms of the number of states and the size of the input (resp. output) alphabet. The second dimension is the *completeness of the state-machine*, a concept originating in the theory of automata. A complete state-machine has transitions (resp. outputs) defined for every input alphabet symbol and from every state. An incomplete state-machine might be missing some transitions, i.e., for some state-input pairs, the next state (resp. output) is undefined.

It is important to distinguish the automata-theoretic concept of state-machine completeness from the model completeness. Consider a simple protocol Π with input messages $\{a,b,c\}$ and output messages $\{w,x,y\}$, specified by the state-machine in Figure 1a. Suppose we have to reverse-engineer the protocol from an implementation of Π while treating it as a black-box. The first step in the inference is to discover the valid input messages. To our knowledge, there exists no automatic approach guaranteeing to discover all the input messages. Suppose a subset M of input messages is discovered during the inference, e.g., $M = \{a,b\}$. An automatic protocol inference approach learns a complete state-machine (Figure 1b) with respect to M if from every state of the learned state-machine and for every message from M, the next state and output are known. Otherwise, the inferred state-machine is incomplete (Figure 1c). The completeness of state-machine models is critical for aforementioned applications, because the more transitions are missing from the model, the higher the uncertainty of the analysis. In this paper, we focus on inferring complete state-machines through interactive *on-line* inference, in contrast to the previous work [10, 21], which focused on passive *off-line* inference of incomplete models. Given a complete alphabet (i.e., set of input messages), our approach infers complete finite-state protocol models with desired accuracy and confidence (the higher the accuracy and confidence, the higher the computational cost).

Passive off-line inference techniques can only learn from a sequence of observed messages. Off-line inference of a minimal (or within a polynomial size of minimal) state-machine from observed network communication is a computationally difficult problem (NP-complete [16], [34, p. 98–99]). Heuristic best-effort inference algorithms are polynomial in the number and size of observed message sequences, as in work of Hsu et al. [21], but there are no guarantees that the inferred model will be minimal, or even at most polynomially larger than the minimal. While off-line techniques can infer models from a relatively small number of observed message sequences, such models are inherently incomplete, rendering any further analysis imprecise.

In contrast, on-line inference techniques are allowed to query the agents involved in the message exchange proactively. On-line techniques, like Angluin's $L*$ algorithm [3], have a polynomial worst-case complexity and produce complete models. Despite polynomial complexity, a number of challenges, especially in the context of botnet protocol inference, remain: (1) Automatic inference of complete protocol state-machines in the real-world network setting requires solving a number of subtle technical challenges, ranging from theoretical ones, like choosing the right formal model, to technical ones, like reverse-engineering message formats. (2) Even a simple protocol with twenty states might require tens of thousands of message sequences to be generated. (3) In order to avoid synchronized attacks by a large botnet on the university infrastructure, the experiments had to be anonymized by tunneling all the traffic through Tor [15]. (4) Exacerbated by the usage of Tor and likely overloading of botnet servers, the network delay averaged 6.8 seconds per message in our experiments, dramatically reducing the number of probing sequences feasible in reasonable amount of time.

1.2 Main Contributions

On the inference side, the most important contribution of this paper is a novel approach for complete state-machine inference in the realistic high-latency network setting. Three innovations rendered such inference possible: First, our formulation of the protocol inference problem as a Mealy machine inference problem results in more compact models and provides a simple, broadly understood formal underpinnings for our work. Second, we propose a highly effective prediction technique for minimizing the number of queries generated during the inference process. Third, we propose two optimizations to the basic $L*$: parallelization and caching.

On the analysis side, we show how the computed complete models can be used as a formal basis to study and defeat botnets: First, we show how to identify the weakest links in a protocol, especially in the context where multiple pools of bots partially share the same resources. Such weakest links are critical for normal functioning of one or more agents participating in the protocol. Second, we show that the inferred model can be used to uncover protocol design flaws. Third, we demonstrate how inferred models can be used to prove the existence of unobservable communication back-channels among botnet servers, although we have no access to those communication channels. Besides proving the existence of such channels, the analysis we propose can actually construct a state-machine representing the model of the back-channel communication. Fourth, we demonstrate how complete models can be used for detecting differences between distinctive implementations of the same protocol.

On the experimental side, we show how to design an effective protocol inference system and provide empirical evidence that our optimizations — query response prediction, parallelization, and caching — speed up the inference process by over an order of magnitude compared to the basic $L*$ algorithm. For instance, our prediction technique alone reduces the time required for the MegaD model inference from an estimated 4.46 days to 12 hours. Applying formal analysis to inferred complete models, we uncover previously unknown facts about the MegaD botnet. Analyzing critical links, we identify the critical components of the botnet shared among multiple pools of bots. Analyzing the properties of the inferred model, we discover a design flaw in MegaD. More precisely, we find a way to bypass MegaD's master server authorization and gain unlimited access to fresh spam templates, which can be used to train spam filters even before a significant percentage of bots starts sending spam based on those templates. Analyzing the back-channels, we prove that MegaD's servers communicate with each other, and we even construct a formal model of such communication. Analyzing differences among Postfix and MegaD's SMTP implementations, we discover a number of interesting differences, useful for detection and fingerprinting. We validate our technique by inferring the complete protocol state-machine from Postfix's SMTP implementation, checking its equivalence against the standard.

2. PROBLEM DEFINITION

A communication protocol is a set of rules for exchanging information over some medium (e.g., the Internet). These rules regulate data representation (i.e., the message format), encryption, and the state-machine of the communication. Any automatic technique for reverse-engineering of real-world protocols has to deduce message formats, handle encryption, and infer state-machines. The first two components of the problem have received significant attention of the research community [5, 7, 12, 13, 36]. The third component — protocol state-machine inference — has received far less attention, and is the focus of this paper.

In the first part of this section, we define the model inference problem informally, and leave a more formal treatment for the subsequent sections. In the second part of this section, we go further to propose several automatic analyses of the model. We define several related problems, which can be solved precisely only if a complete state-machine of the protocol is known. In the third part of this section, we outline our assumptions.

2.1 Model Inference

The goal of protocol model inference is to learn a state-machine describing the protocol composed of a finite set of states and a transition relation over a finite alphabet. In general, it is not possible to learn a completely accurate model, without having access to a source of counterexamples that show when the learned model differs from the actual system [3]. Thus, every protocol inference approach is necessarily an ε-approximation, i.e., the inference cost is proportional to the desired accuracy ε and confidence γ.[1] When counterexamples are available, $L*$ can make at most a polynomial number of queries, but in the approximation setting, the number of queries depends on the desired accuracy and confidence.

There are two basic types of finite state-machines: the Moore machine [28] and the Mealy machine [26]. Informally, the former distinguishes states according to whether they are accepting or not, while the latter has no accepting states and distinguishes states according to the sequence of outputs produced from a sequence of transitions. Since protocols are reactive systems[2], the Mealy machine is a more appropriate model. The problem this paper addresses is how to learn the Mealy machine describing the studied protocol in the realistic network setting, treating the protocol implementation as a black-box and learning the state-machine from active probing. We actually set the bar higher: The problem we want to solve is to learn the minimal (the fewest states) complete (transitions defined for all inputs and states) Mealy machine describing the protocol.

2.2 Model Analysis

Once the protocol model is constructed, we wish to perform four types of analysis: identification of the critical links in the protocol, identification of protocol design flaws, proving the existence of the background communication over unobservable channels, and (dis)proving equivalence of different implementations of the same protocol (a.k.a. equivalence checking [24]).

Identification of the critical links in a protocol is important for optimizing the attacks on the botnet. We define the problem as follows: Given an initial state of the protocol and some set of bad actions (e.g., spamming), represented with output responses,[3] that we want to prevent from happening, what is the minimal set of transition edges that need to be disrupted in order to prevent the bots from executing those actions? The bad actions can be disrupted either by making it impossible for the bots to reach the state from which the bad action is executed, or by disrupting the bad actions themselves.

Identification of protocol design flaws can be done through manual inspection of the model or automatic model checking (e.g., [9]). In either case, the analyst needs to come up with a set of properties and then check whether the model satisfies them. For instance, one of the properties we checked was: "Bot cannot obtain spam templates before (1) being authenticated by the master server and (2) getting a command to download spam templates."

Proving the existence of background communication among servers whose communication we cannot eavesdrop is important for gaining knowledge about the communication over channels we have no access to. The knowledge of existence of such channels can help security researchers in detecting infiltration traps. We define the problem as follows: Given a client (a bot) communicating with a certain number of servers (three in the MegaD case: the master, SMTP [23], and template server), can we prove existence and build a model of inter-server communication, only from the observed communication between the client and servers?

Equivalence checking (e.g., [24]) is an automatic analysis that takes two formal models and either proves that models are equivalent, or finds counterexamples showing the differences. In our setting, we were especially interested in finding differences between MegaD's custom SMTP implementation and the standard SMTP to detect the features that could be used for fingerprinting. In a broader setting, equivalence checking can be used for detecting deviations from the standard, differences among different implementations of the same protocol, and uncovering implementation flaws.

2.3 Assumptions

Determinism. We assume that the protocol to be learned is deterministic, i.e., that the same sequence of inputs from the initial state always produces the same sequence of outputs and ends in the same state. Among all the protocols we studied, we found only one minor easy-to-handle source of non-determinism in the MegaD botnet protocol. We explain later how we handle that specific source of non-determinism. Some limited amount of non-determinism can be handled by extending the alphabet. For example, if a protocol implementation in state s responds to message m either immediately with r or waits for ten seconds and responds with t, one can split m into two messages, m_0 and m_{10}, such that the response to m_0 (resp. m_{10}) is r (resp. t). Each of the two messages can transition from s into different states.

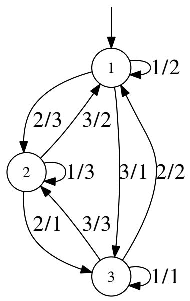
Resettability. State-machine inference algorithms require the means of resetting the machine to a fixed start state. The sequence of inputs that reset the state-machine into its start state is known as a *homing sequence* and every finite state-machine has such a sequence [32]. Network protocol state-machines are often easily reset by initiating a new connection or session. Thanks to the prior work on the message format reverse-engineering [6, 12, 13], we know the messages that reset the state-machines of the protocols we study.

[1] The ε accuracy should not be confused with the empty string, also denoted ε.

[2] Reactive systems maintain an ongoing interaction with their environment rather than produce some final value upon termination.

[3] Bad output responses in a Mealy machine correspond to bad states of an equivalent Moore machine.

(a)



(b)

		E		
		1	2	3
S	ε	2	3	1
	2	3	1	2
	3	1	2	3
$S\cdot\Sigma_I$	1	2	3	1
	$2\cdot 1$	3	1	2
	$2\cdot 2$	1	2	3
	$2\cdot 3$	2	3	1
	$3\cdot 1$	1	2	3
	$3\cdot 2$	2	3	1
	$3\cdot 3$	3	1	2

Figure 2: A Mealy Machine and the Corresponding Observation Table. The initial state is denoted by an incoming edge with no source. Edges are labeled with input/output symbols. In this paper, all three set of labels — state, input, and output labels — are unrelated sets of integers.

Finiteness. We are interested in inferring finite-state protocols (or finite abstractions of infinite-state ones) with finite input and output alphabets. We leave the inference of more expressive models for the future work.

Known Message Format Semantics. To abstract input and output messages with a finite alphabet, we rely upon the previous work on automatic message format reverse-engineering [7]. Another interesting option is the automatic message clustering and abstraction proposed by Comparetti et al. [10].

Known Encryption. Many botnet protocols encrypt messages, complicating protocol inference. If protocol messages are encrypted, our approach needs to know the protocol encryption and decryption functions. We leverage the previous work of Caballero et al. [5] to automatically extract and circumvent encryption functions.

3. BACKGROUND MATERIAL

This section briefly surveys prior work required for understanding this paper. In the first half, we provide a brief overview of the main definitions and results in the automata theory and inference. In the second half, we briefly describe botnets, which are our main targets. Even though the targets that we study are highly specific, the results of our research are applicable to a broad range of protocols in the realistic network setting.

3.1 Inference of Mealy Machines

Protocol inference is a special case of the grammatical inference problem [14], because the problem of the finite state-machine inference is isomorphic to the problem of learning a regular language. In this paper, we solely use Mealy machines, as they are a more appropriate model for reactive systems, which have neither accepting nor rejecting states, and are more compact than Moore machines [28] by a linear factor equal to the size of the output alphabet.

DEFINITION 1 (MEALY MACHINE [26]). *A Mealy machine is a six-tuple $(Q,\Sigma_I,\Sigma_O,\delta,\lambda,q_0)$, where Q is a finite non-empty set of states, $q_0\in Q$ is the initial state, Σ_I is a finite set of input symbols, Σ_O is a finite set of output symbols, $\delta : Q\times\Sigma_I\longrightarrow Q$ is the transition relation, and $\lambda : Q\times\Sigma_I\longrightarrow\Sigma_O$ is the output relation.*

In this paper, we focus on inferring *complete* protocol state machines. A complete Mealy Machine is defined as follows:

DEFINITION 2 (COMPLETE MEALY MACHINE). *A Mealy machine $(Q,\Sigma_I,\Sigma_O,\delta,\lambda,q_0)$ is complete if and only if δ and λ are defined $\forall m\in\Sigma_I$ and $\forall q\in Q$.*

We rely upon prior work [6] for reverse-engineering of the alphabet and claim no contribution on that front. We are not aware of any automatic technique capable of reverse-engineering the complete alphabet, so we necessarily have to work with a subset. Given a subset of the alphabet, our technique infers a complete state-machine (Definition 2), in contrast to prior work on protocol inference that learns incomplete state-machines [10, 21]. To the best of our knowledge, our work is the first to demonstrate a technique for complete protocol state-machine inference in the realistic network setting.

In this paper, input symbols will be denoted with letters from the beginning of the alphabet a,b,c, output symbols with letters from the end of the alphabet x,y, and strings of input symbols with letters r,s,t,u. The set of all symbols in a given string is denoted as $[\,]$, e.g., $[a\cdot b\cdot b]=\{a,b\}$, where '$\cdot$' is the concatenation operator. We extend the δ and λ relations to strings of characters from Σ_I, for example, $\delta^*(q,a\cdot b\cdot c)=\delta(\delta(\delta(q,a),b),c)$ and $\lambda^*(q,a\cdot b\cdot c)=\lambda(q,a)\cdot\lambda(\delta^*(q,a),b)\cdot\lambda(\delta^*(q,a\cdot b),c)$. We shall frequently use a more intuitive term *response* for a string of output symbols returned by λ^*.

Angluin [3] proposed the first polynomial algorithm, called $L*$, for learning Moore machines. Shahbaz and Groz [33] adapted the algorithm for Mealy machines and proposed several optimizations. In this section, we briefly describe Shahbaz and Groz's algorithm, while in the following section, we describe our improvements and optimizations of their algorithm. The $L*$ algorithm maintains an observation table:

DEFINITION 3 (OBSERVATION TABLE [3]). *An observation table is a triple (S,E,T), where S is a prefix-closed*[4] *set of strings of input symbols from Σ_I^* and set E is a suffix-closed set of strings from Σ_I^+. Let set $S\cdot\Sigma_I$ be defined as a concatenation of all strings in S with every alphabet symbol in Σ_I, i.e.: $S\cdot\Sigma_I=\{s\cdot a\mid s\in S,a\in\Sigma_I\}$. Then map $T:(S\cup S\cdot\Sigma_I)\times E\longrightarrow\Sigma_O^+$ is a map from $S\cup S\cdot\Sigma_I$ and E to a non-empty sequence of output symbols. Map T has an additional property: $\forall s\in S\cup S\cdot\Sigma_I, r\in E, x\in\Sigma_O^+$. $(T(s,e)=x)\Rightarrow|r|=|x|$.*

Intuitively, S and $S\cdot\Sigma_I$ can be seen as two sets of rows, E as a set of columns, and T as a map that maps input strings obtained by concatenation of rows ($S\cup S\cdot\Sigma_I$) and columns (E) to output strings of the same length as the column part of the input string. For example, from Table 2b, it follows that $T(2\cdot 1,3)=2$, where $2\cdot 1\in S\cdot\Sigma_I$ is a row and $3\in E$ is a column.

Angluin introduced two observation table properties: closure and consistency. Only the first property is important for understanding this paper, while the second is always satisfied if T is computed in a specific way [33].

DEFINITION 4 (CLOSED OBSERVATION TABLE [3]). *We say that an observation table is* closed *if and only if $\forall s\in S\cdot\Sigma_I$. $\exists t\in S$. $\forall r\in E$. $T(s,r)=T(t,r)$.*

Intuitively, a table is closed if for every row s in the $S\cdot\Sigma_I$ set of rows, there is a row t in S having exactly the same responses, i.e., $\forall r\in E$. $T(s,r)=T(t,r)$. For example, Table 2b is closed. Table T is usually computed in such a way that no two rows in S are equivalent, i.e., there is always only one representative in S of each class of equivalent rows. For example, rows ε and $2\cdot 3$ in Table 2b are equivalent, so only one representative (ε) is in S. Further, the representative put in S is minimal, according to the following ordering.

[4] Let *Pref* be a function that returns a set of all prefixes of a string. Set S is prefix-closed if and only if $\forall s\in S$. $Pref(s)\subseteq S$. Note that an empty string ε is a prefix of every string. The definition of a suffix-closed set is similar.

DEFINITION 5 (LEXICOGRAPHIC ORDERING). *Let lexicographic ordering relation $<: \Sigma_I^* \times \Sigma_I^* \longrightarrow Bool$ be a total order relation over two strings of input symbols, say $s = a_0 \cdots a_n$ and $t = b_0 \cdots b_m$, defined as follows.*

$$s < t = \begin{cases} |s| < |t| & \text{if} \quad |s| \neq |t| \\ a_j <_a b_j & \text{if} \quad s \neq \varepsilon \wedge a_j \neq b_j \wedge \forall 0 \leq i < j < |s|.a_i = b_i \\ \mathit{false} & \text{otherwise} \end{cases}$$

Define $s \leq t$ as $s < t \vee s = t$. The alphabet symbols are ordered according to some arbitrary total ordering $<_a$.

Intuitively, the definition imposes a total ordering on strings, given a total order of the alphabet symbols, e.g., if $a <_a b <_a c$, then $b \cdot a < b \cdot c$. We use this ordering in our implementation of the $L*$ algorithm (Algorithm 1) to choose the minimal representative of a class of equivalent rows.

Algorithm 1 The $L*$ Algorithm: Closing the Observation Table.

```
$S = \emptyset, S \cdot \Sigma_I = \{\varepsilon\}, E = \Sigma_I$
while $\exists s \in S \cdot \Sigma_I$ . $(\forall t \in S \cdot \Sigma_I$ . $s \leq t) \wedge$
      $(\forall t \in S$ . $\exists r \in E$ . $T(s,r) \neq T(t,r))$ do
    Move $s$ to $S$
    for all $a \in \Sigma_I$ do
        $S \cdot \Sigma_I = S \cdot \Sigma_I \cup s \cdot a$
        for all $u \in E$ do
            Compute response to $s \cdot a \cdot u$
            Take suffix $x$ of the response (last $|u|$ symbols)
            Update table $T(s \cdot a, u) = x$
Run sampling and Algorithm 2 if there is a counterexample
```

Next, we give a high-level description of the $L*$ algorithm. $L*$ has two steps. In the first step, $L*$ initializes S to a set containing an empty string, $S \cdot \Sigma_I$ to an empty string ε, the columns E to the input alphabet[5], and T to the responses of the system under study to inputs constructed from concatenating the rows (i.e., an empty string at this point) with columns (individual alphabet symbols at this point). The algorithm then iteratively keeps closing the table and re-computing the responses to all $S \cdot \Sigma_I \cdot E$ sequences of messages. Each sequence has to be generated, transmitted on the network, response recorded, and stored into table T. The $S \cdot \Sigma_I \cdot E$ queries generated in the first step of $L*$ are called membership queries.

In the second step, $L*$ constructs a complete minimal conjecture automaton from a closed table. This construction is performed as follows: The rows in S represent the minimal set of states, rows in $S \cdot \Sigma_I$ represent transitions for every symbol in Σ_I, and the range of T (i.e., the elements of the table) represent the output relation. The conjectured automaton is then, in the original algorithm [3], passed to a *teacher*, which either confirms that the machine has been correctly learned, or returns a counterexample. The counterexample is a sequence of inputs and corresponding outputs that does not match the conjectured machine. Of course, we cannot ask the bot master to tell us whether our conjectured state-machine is correct or not, so we use a sampling-based approach [3], which randomly generates uniformly distributed sequences of messages used to sample the protocol's state-machine and discover mismatches. If the response of the agents involved in the message exchange does not match those predicted by the conjectured state-machine, we have a counterexample. Based on the number of sampling sequences and the number of conjectures, one can compute the accuracy and confidence that the conjectured machine is correct, as proposed by Angluin [3]. The sampling queries generated in the second step of $L*$ are called equivalence queries, for they check the equivalence between the conjectured and to-be-learned machine.

Algorithm 2 The $L*$ Algorithm: Handling Counterexamples.

```
Let $r, x$ be a counterexample input ($r$) and its response ($x$)
Let $r = p \cdot s$, s.t. $p \in S \cup S \cdot \Sigma_I$ and $p$ is the longest such prefix
for all $u$ suffixes of $s$ do
    $E = E \cup u$
    for all $t \in S \cup S \cdot \Sigma_I$ do
        Compute response to $t \cdot u$        ▷ If $t \cdot u = r$, response is $x$
        Take suffix $y$ of the response (last $|u|$ symbols)
        $T(t, u) = y$
Run the inner loop of Algorithm 1 if $T$ changed
```

Algorithm 2 illustrates Shahbaz and Groz's algorithm for handling counterexamples. When a counterexample, an input (r) and the corresponding response (x), is found, the algorithm finds the longest prefix of r existing in $S \cup S \cdot \Sigma_I$, and trims that prefix. The remaining suffix and all its suffixes are added to the columns (set E) of the observation table, and the responses for all the rows extended with newly added suffixes are computed.

The number of required membership queries depends on the size of the state-machine to be inferred. Even a twenty-state machine could require tens of thousands of queries. To reduce the number of membership queries, we developed a novel response prediction heuristic, which exploits redundancy in inferred models. Any mispredictions are guaranteed to be detected by the sampling queries with the desired accuracy and confidence. Since the number of sampling queries is computed solely from the accuracy, confidence, and the number of conjectures [3], detecting mispredictions does not require any additional sampling queries.

Our response prediction heuristic exploits the abundance of self-loop transitions, defined as follows.

DEFINITION 6 (SELF-LOOP TRANSITIONS). *Let q be some state and a some input symbol. Transitions for which $\delta(q,a) = q$ are called* self-loop transitions*.*

3.2 Botnets

A botnet is a network of compromised hosts controlled remotely by botmasters to carry out nefarious activities such as denial of service, theft of personal information, and spamming. Botmasters control their botnets through a system of Command and Control (C&C) servers. MegaD is a mass spamming botnet first observed 2007, and was responsible for one-third of the world's spam at its peak[6]. The main MegaD C&C server used by each pool of bots is the Master server, which points bots to a set of auxiliary (Template, SMTP, and Drop) servers. A spamming MegaD bot contacts only the Template and SMTP servers [8].

MegaD has been the target of multiple takedown attempts. The most recent attempt occurred in Nov. 2009, but MegaD bounced back after the takedown. The common practice to botnet takedowns is to identify as many C&C servers as possible, and attempt to cripple the entire botnet by sending abuse notifications to all ISPs involved simultaneously.[7] This is an expensive exercise requiring careful co-ordination among all parties involved. In this paper, we show through protocol inference that MegaD's SMTP servers are actually the critical link in the C&C, and taking just SMTP servers down would be a cheaper and simpler option.

[5] The algorithm can also trivially handle monotonically increasing alphabets, useful when new messages are discovered during the inference, but we abstracted that away for clarity.

[6] http://www.m86security.com/trace/i/Mega-D,spambot.896∼.asp

[7] http://blog.fireeye.com/research/2009/11/smashing-the-ozdok.html

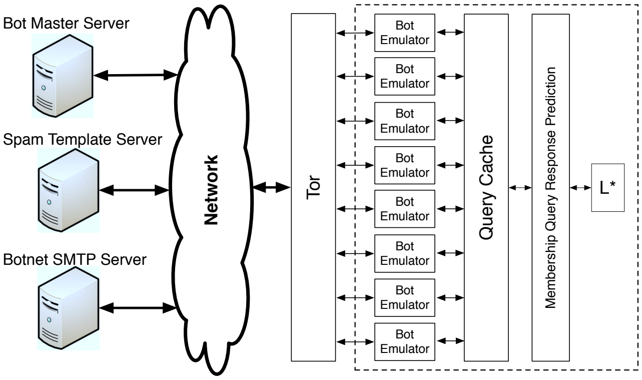


Figure 3: Architecture of Our Protocol Inference Engine. Our inference system is encircled with a dashed line.

4. INFERENCE OF PROTOCOL MODELS

This section presents our technique for inference of complete protocol state-machines in the realistic network setting. The high-level architecture of our implementation is shown in Figure 3. Our implementation is composed of several components: a bot emulator, a query cache, a membership query predictor, and $L*$.

The bot emulator is a script we wrote that receives queries (strings of symbols from the input alphabet) from $L*$ and concretizes them into valid protocol messages sent to botnet servers. Once the bot emulator receives a response, it abstracts the response into strings of symbols from the output alphabet and sends such abstracted strings to $L*$. We describe abstraction and concretization in Section 4.1.

We built our bot emulator from scratch, to assure it cannot perform any of the malicious activities (spamming and infection) of the real bot. In addition, we carefully crafted our experiments not to cause any harm to any party involved (infected users, ISPs, C&C servers, and botmasters). For example, our bot emulator is careful not to construct corrupted messages intentionally, minimizing the impact even on the C&C servers.

The query cache acts as a concentrator of parallel query responses and caches the results, so that each sequence of messages has to be sent over the network only once. Through parallelization, we achieved 4.85X reduction in the time required for the inference using eight machines, each running one bot emulator. We describe both parallelization and caching in Section 4.2.

The membership query predictor attempts to predict what is the most likely response to membership queries. Learning even a state-machine of a medium size can require a significant number of queries (c.f. Section 3.1). As queries can be long strings of input messages, and getting a response to each message can take a long time due to network delay (6.8 sec on average in our experiments), accurate prediction of responses is important to infer protocol state-machines in reasonable amount of time. Erroneous predictions are guaranteed to be detected (with desired accuracy and confidence) using sampling queries and fixed by backtracking to the first mistake made by the predictor. We present our prediction heuristic in Section 4.3.

In Section 4.4, we explain how we handled the only discovered source of non-determinism in the MegaD protocol, state-machine resetting, and generation of sampling queries.

4.1 Message Abstraction and Concretization

$L*$ constructs queries over the abstract input alphabet and passes them to our bot emulator, which concretizes the alphabet symbols into valid network messages and sends them to botnet servers. When responses are received, our emulator does the opposite — it abstracts the response messages into the output alphabet and passes them to $L*$. Construction of the input and output alphabets is a partially automatic and partially manual process. We reverse-engineer the message formats and their semantic content using automatic protocol reverse engineering [6] and encryption/decryption modules extracted from the bot binary [5]. Once we learn the message formats, we perform abstraction manually. The manual abstraction is a straight-forward process, but requires intelligence in deciding which message fields are important. In particular, the most important field, not surprisingly, turns out to be the message type field.

Besides abstraction, another important aspect of automatic protocol inference is concretization of messages, i.e., generation of valid network messages. If messages are invalid or have invalid session tokens, the server will simply reject them. To generate valid messages, the bot emulator uses the automatically reverse-engineered message format grammar, rewrites the message fields as needed, and encrypts the messages before transmission. The bot emulator rewrites the tokens of server-bound messages using tokens issued by the server in the same session. If the token has not been issued, the emulator rewrites tokens with a random value, to learn how servers handle invalid tokens.

To assure our experiments are reproducible, we include MegaD's C&C message format tree in Figure 4 and a list of all abstracted messages used in this paper in Table 1. MegaD uses a proprietary C&C protocol for communication with its master and template servers, and a non-standard SMTP protocol for communication with its SMTP server. To model the C&C protocol messages, we use three fields: the *MsgType* field found in all messages, the *SubType* field in the INIT and GETCMD messages, and the *Config* element found only in the spam template messages. We define a unique symbol for each unique {*MsgType*, *SubType*, *Config*} combination as described in Table 1. To model the bot's communication with the SMTP server, we abstract MegaD's SMTP dialogs at two different levels of abstraction. When we model the role of the SMTP server in the overall C&C protocol, we abstract MegaD's spam capability test dialog and pre-spam notification dialog each with two symbols, one in either direction (IDs 14 and 15 in upper half and IDs 12 and 13 in lower half of Table 1 respectively). When we analyze the details of MegaD's SMTP protocol implementation, we abstract each individual SMTP message within a TCP connection into individual symbols.

4.2 Query Cache

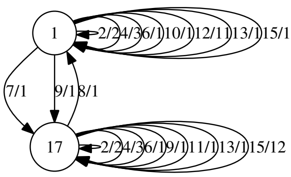
In our implementation, the query cache is just a file that stores pairs of input message sequences and the corresponding responses (i.e., sequences of output messages). The cache acts primarily as a concentrator of parallel query responses. This architecture (Figure 3) simplifies the implementation of $L*$, for only the queries have to be issued in parallel, but responses can be processed sequentially. More precisely, the membership query loop starting on Line 4 of Algorithm 1 and the column extension loop starting on Line 3 of Algorithm 2 both execute a number of independent membership queries, which can all be either predicted or run on the network in parallel. Our implementation of $L*$ emits all these queries in parallel. The queries are partitioned among a number of machines (eight, in our case) and independently run on the network. In addition, the cache has a more traditional role of caching the responses that have been tested on the network — such responses can be reused at any point, because of the determinism assumption (Section 2.3).

4.3 Response Prediction

Studying the membership queries that $L*$ makes while learning protocol models, we came to realize that there is a significant

ID	MsgTy	SubTy	Template	Semantic Label	Direct.
-	0x00	0x00	-	INIT	→ MS
1	0x00	0x01	-	GETCMD	→ MS
2	0x03	-	-	DLSUCCESS	→ MS
3	0x05	-	-	DLERROR	→ MS
4	0x06	-	-	DL1	→ MS
5	0x09	-	-	DL2	→ MS
6	0x16	-	-	HOSTINFO	→ MS
7	0x22	-	-	CAP_TESTPASS	→ MS
8	0x23	-	-	CAP_TESTFAIL	→ MS
9	0x25	-	-	TS_RECVED	→ MS
10	0x09	-	-	PONG	→ MS
11	0x1c	-	-	TEMPLATE_ACK	→ TS
12	0x1d	-	-	GET_TEMPLATE	→ TS
13	0x1e	-	-	SPAM_STATUS	→ TS
14	-	-	-	TEST	→ SS
15	-	-	-	NOTIFY	→ SS
-	0x01	-	-	INFO	← MS
1	-	-	-	TCP_CLOSE(-)	← MS ← TS
2	0x04	-	-	ACK_DLRESULT	← MS
3	0x07	-	-	ACK_DL1	← MS
4	0x0a	-	-	ACK_DL2	← MS
5	0x0e	-	-	WAIT2	← MS
6	0x15	-	-	GETHOSTINFO	← MS
7	0x18	-	-	WAIT1	← MS
8	0x1c	-	None	TEMPLATE	← TS
9	0x21	-	-	TESTSPAMCAP	← MS
10	0x24	-	-	DLTEMPLATE	← MS
11	0x1c	-	RENEW	RENEW	← TS
12	-	-	-	TESTPASS	← SS
13	-	-	-	NOTIFY_RECVED	← SS

Table 1: Abstraction of MegaD's Communication with its Master and Template Servers. The upper half of the table shows the input alphabet (sent by our bot emulator to various servers), and the lower half the output alphabet (responses sent by servers to our bot). We use the MsgType, SubType and Template config fields for abstraction. Messages sent to and received from C&C servers are abstracted as input and output alphabet symbols (the ID column) respectively, where MS is the Master Server, TS is the Template Server and SS is the SMTP Server. The no-response message is denoted as TCP_CLOSE(-). The INIT message is sent to reset each session. The INFO message is always followed by another message, and that second message is used for abstraction. The SMTP messages are abstracted according to the standard [23].



amount of redundancy in the inference process. Namely, many states have many self-loops (Definition 6). The figure on the left illustrates this phenomenon on a small piece of the MegaD state-machine. Such self-loops increase the number of membership queries, without helping $L*$ to distinguish states. We believe that there are two major factors contributing to the abundance of self-loops: (1) Our goal is to learn complete models, which means we need to determine the response to *all* input alphabet symbols (i.e., messages) from every state. Most protocols use each message for a single, well-defined purpose, which means that most often sending an unexpected message from some state causes that message to be ignored, either at that state or in an error state. (2) We need to abstract conservatively the messages into the input and output alphabets *before* the protocol state-machine is known. The conservative overestimation frequently increases the redundancy in the model, e.g., increasing the number of self-loops, which cannot be eliminated before the state-machine is known — a chicken and egg problem. On the other hand, reducing the size of the alphabet oversimplifies the inferred state-machine. Thus, it is important to be conservative in choosing the alphabet and develop effective techniques for improving the performance of the inference process.

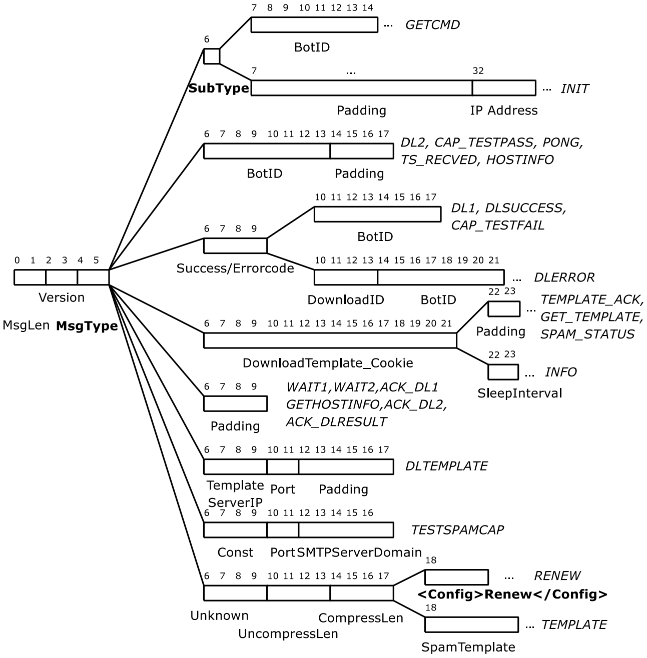


Figure 4: MegaD's C&C Message Format Tree. Message type fields used for abstraction are shown in bold. We elide the tails of messages not relevant to our abstraction process. Abstracted messages are shown on the right in italics.

To understand the intuition behind response prediction, consider the running example shown in Figure 2a. There are three self-loops, and we shall focus on the self-loop incident to state 2. Since self-loops return back to the same state, the response to $2 \cdot 1 \cdot u$ (with self-loop) and $2 \cdot u$ (without self-loop) will be the same for every input string u. Accordingly, the table entries for $2 \cdot 1 \cdot u$ (row 5) and $2 \cdot u$ (row 2) in Table 2b are the same. Recall from Algorithm 1 that the length of membership queries increases with each outer-for-loop iteration. Thus, the $2 \cdot u$ query must preceede the $2 \cdot 1 \cdot u$ query, and the response to the former can be used to predict the response to the latter at the end of each outer-for-loop iteration.

We exploit the redundancy with a two-level heuristic. The first level, called *restriction-based* prediction, exploits the abundance of self-loops, for guessing responses to membership queries with high accuracy. The second level, called *probability-based* prediction, exploits the fact that the same input messages often leads to the same output message. Any possible prediction errors are detected using random sampling (Section 3.1). If an error was made predicting the responses to membership queries, our algorithm backtracks to the first erroneously predicted membership query, fixes the error (according to responses from sampling queries), and continues running $L*$. Importantly, the same sampling queries can be used both to detect missing states (as in the classical Angluin probabilistic sampling [3]) and mispredictions.

We begin the formal presentation of our prediction techniques by showing that entries in the S set have no self-loops, and therefore can be used for prediction of responses that have self-loops.

THEOREM 1. *Let $s \in S, s = c_0 \cdot c_1 \cdots c_n$ be a string of n characters from Σ_I. Let $q_0, q_1, \cdots, q_n, q_{n+1}$ be a sequence of states visited by $\delta^*(q_0, s)$. For $0 \leq i < n+1$, every two adjacent states are different, i.e., $q_i \neq q_{i+1}$.*

PROOF. Proof is by induction on the string length. The ε string trivially satisfies the condition, as there is only one state in the sequence of visited states. Let $s = c_0 \cdots c_k$ and $s \in S$. Algorithm 1 on Line 5 appends $a \in \Sigma_I$ to s, and then in the loop that follows computes responses to membership queries $s \cdot a \cdot u$, where $u \in E$ are the columns of T. Without loss of generality, let $s \cdot a$ be the smallest string, according to our lexicographic ordering (Definition 5). String $s \cdot a$ is moved to S (Line 3) only if $\forall t \in S \,.\, \exists u \in E \,.\, T(t,u) \neq T(s \cdot a, u)$. Every $s \cdot a$ string is obtained by extending strings from S, so we know that $s \in S$. Thus, we conclude: $\exists u \in E \,.\, T(s,u) \neq T(s \cdot a, u)$, which implies that $\delta^*(q_0, s \cdot u) \neq \delta^*(q_0, s \cdot a \cdot u)$, because s and $s \cdot a$ are (eventually) both in S and they denote different states. Since $s \cdot a \cdot u$ and $s \cdot u$ have the same prefix and suffix, symbol a must explain the difference in the response of the state-machine. By induction hypothesis $\forall 0 \leq i < k \,.\, c_i \neq c_{i+1}$. So we have: $\delta^*(q_0, s \cdot a) = c_0 \cdots c_k \cdot c_{k+1}$. Assume $c_k = c_{k+1}$. Thus, $\delta^*(q_0, s \cdot a) = \delta^*(q_0, s)$, a contradiction. So, we have proved that in $\delta^*(q_0, s \cdot a)$, the last two visited states are different, which proves the theorem. □

COROLLARY 1. *From Definition 6, it follows that $\delta^*(q_0, s)$ for $s \in S$ has no self-loop transitions.*

Intuitively, Theorem 1 suggests that strings from S, which are loop-free, could be used to predict responses to input strings that are similar to strings from S, but have additional symbols producing self-loop transitions sprinkled around. We formalize that intuition using the following two definitions.

DEFINITION 7 (DIFFERENTIATING SET). *Let $D \subseteq \Sigma_I$ be a set of all symbols in S, more precisely, $D = \{c \mid c \in [s], s \in S\}$. As S grows during the execution of Algorithm 1, D is going to be a monotonically increasing set.*

DEFINITION 8 (RESTRICTION FUNCTION). *Let $a \in \Sigma_I$, $s \in \Sigma_I^*$, and $D \subseteq \Sigma_I$. The restriction function $\rho : \Sigma_I^* \times D \longrightarrow \Sigma_I^*$ is then defined recursively as follows:*

$$\rho(s, D) = \begin{cases} \varepsilon \ \ if \ \ s = \varepsilon \\ \rho(r, D) \ \ if \ \ s = a \cdot r \ \wedge \ r \in \Sigma_I^* \ \wedge \ a \notin D \\ a \cdot \rho(r, D) \ \ if \ \ s = a \cdot r \ \wedge \ r \in \Sigma_I^* \ \wedge \ a \in D \end{cases}$$

Intuitively, the restriction function just deletes the symbols that are not in a given set from a string. For example, $\rho(a \cdot b \cdot b \cdot a \cdot c \cdot d, \{b, d\}) = b \cdot b \cdot d$.

Now, our restriction-based prediction rule can be simply stated as: Given any membership query $s \cdot a \cdot u$, compute $s' = \rho(s \cdot a, D)$, and if s' already exists in $S \cup S \cdot \Sigma_I$, use the values in the s' row of table T to predict the values for the $s \cdot a \cdot u$ row. More formally: $T(s \cdot a, u) = T(\rho(s \cdot a, D), u)$, if the $\rho(s \cdot a, D)$ entry exists in T. With this simple rule, we get highly accurate prediction, with few errors.

The restriction-based prediction saves around 73% of membership queries in our experiments with MegaD. Analyzing the results, we identified one missed prediction opportunity. If the restricted input string does not exist in the table, the previously presented technique is helpless. To improve the performance of the restriction-based prediction even further, we track the set of observed response messages for every input symbol. When the restriction-based approach fails, we apply a simple probability-based prediction: If a particular input message produces the same output message for *all* previous queries, we predict that response will be the same. If multiple different responses were observed for the same input, we do not predict it. It would certainly be possible to lower the prediction threshold — say, by picking the response that happens in at least 90% of cases — at the cost of increasing the number of erroneous predictions and the cost of backtracking. We leave this fine-tuning for future work. Even with the simple probability-based prediction currently implemented, we gain an additional 13% reduction in the number of membership queries on MegaD, in addition to savings achieved by the restriction-based prediction.

Mispredictions can produce erroneous state-machine conjectures. However, we exploit the same random sampling equivalence checking mechanism in $L*$ (Section 3.1) to detect mispredictions. Thus, mispredictions are guaranteed to be found with desired accuracy ε and confidence γ. Once an error is detected, $L*$ backtracks to the first erroneously predicted query, fixes it using the sampling query response, removes all subsequently predicted entries from the observation table, and continues the inference process. All the prediction savings, both in the prior discussion and in the experimental evaluation, take the cost of backtracking into account. So, it is obvious that our prediction is very effective (86% total reduction in the number of queries) and accurate (inaccurate prediction would require more frequent backtracking, reducing the savings).

4.4 Determinism, Resettability, and Sampling

This section describes the non-standard and non-obvious aspects of using $L*$ in our setting. More precisely, we discuss the impact of the determinism and resettability assumptions (Section 2.3) and the role of the sampling process in achieving the desired accuracy of the model.

Both the determinism and resettability assumptions were relatively easy to satisfy in our setting. In our experiments, the exchange of messages was deterministic, except for one corner case: Sometimes master servers respond with an arbitrarily long sequence of INFO messages, which are always terminated with a non-INFO message. Our inference infrastructure discards all the INFO messages, and treats the first non-INFO message as the response. This was the only source of non-determinism we encountered. To reset the state-machine, we begin both membership and sampling queries with an INIT message (c.f. Table 1), which initiates a new session. Once the session is started, every input message produces a response — an output message.

As discussed in Section 3.1, we use a sampling-based approach for equivalence queries. We generate uniformly distributed random sequences of input messages, the number of which is determined by the desired model accuracy and confidence [3]. Once our implementation of $L*$ closes the table, it conjectures a state-machine, which is then tested through sampling. The responses to sampling queries are never predicted, for the purpose of sampling queries is to discover new states that do not exist in the currently conjectured model and to discover prediction errors.

5. ANALYSIS OF INFERRED MODELS

In this section, we analyze the complete protocol models obtained from our inference technique with the goal of gaining deeper understanding of MegaD. We present techniques to analyze the protocol models to identify the critical links in botnet C&Cs, design flaws, the existence of background communication channels between C&C servers, and to identify implementation differences for fingerprinting and flaw detection.

5.1 Identifying the Critical Links

Transitions in our Mealy machine models represent actions. Certain actions might be considered as *bad*, in the sense that they represent a malicious or undesirable activity. When Mealy machines are used, such activities are represented with transitions (more precisely, output responses).[8] Once the bad transitions are identified, we wish to find a way to prevent such transitions from ever being executed.

More formally, given a protocol state-machine $M = (Q, \Sigma_I, \Sigma_O, \delta, \lambda, q_0)$ and a set of bad output symbols $B \subseteq \Sigma_O$ representing bad actions, we wish to identify the minimal number of transitions we have to disrupt to prevent the bots from executing transitions that would produce output symbols from B. There are two ways (not mutually exclusive) of achieving this. The first option is to make it impossible for bots to reach the states from which bad actions could be performed by cutting a set of transitions in the state-machine, i.e., we wish to assure that $\forall s \in \Sigma_I^*, a \in \Sigma_I \,.\, \lambda\left(\delta^*(q_0, s), a\right) \notin B$. The second option is to disrupt the bad transitions themselves, i.e., to remove the transitions so as to assure that the following property holds: $\forall s \in \Sigma_I^* \,.\, [\lambda^*(q_0, s)] \cap B = \emptyset$.

Such an analysis can be done using max-flow min-cut algorithms [11], such that the initial state is a source, and the state at which the bad transition originates is a sink. We performed this analysis on the MegaD state-machine and arrived at a trivial conclusion for a single pool of bots: since MegaD is a spamming botnet and a single spamming edge is the only bad edge, taking down any one of the botnet servers and the corresponding transitions in the state-machine would prevent a pool of bots from spamming. However, since different pools of bots talk to different sets of servers, it does not stop other pools of MegaD bots from spamming. Unsatisfied with this outcome, we attempt to develop an approach that works across multiple pools of bots.

We extended the encoding of messages into alphabet symbols shown in Table 1 by partitioning the set of messages into disjoint sets, one set per server, so as to include the IP addresses of the servers as an additional field. We shall refer to this extended alphabet as IP-extended. We ran our inference technique independently on both master servers we have access to (each pool of bots talks to a different master server), and computed a projection (defined below) of one state-machine onto the IP-extended alphabet of the other.

DEFINITION 9 (STATE-MACHINE PROJECTION). *The projection of a finite state-machine $M = (Q, \Sigma_I, \Sigma_O, \delta, \lambda, q_0)$ onto alphabet Σ_A is defined as a non-deterministic finite state-machine $M' = (Q, \Sigma_I \cap \Sigma_A, \Sigma_O, \delta', \lambda', q_0)$, such that the following holds for $\forall a \in \Sigma_I, x \in \Sigma_O, q_i, q_j \in Q$:*

$$\begin{aligned}
(q_i, \varepsilon, q_j) \in \delta' &\quad \textit{iff} \quad (q_i, a, q_j) \in \delta \wedge a \notin \Sigma_A \\
(q_i, a, q_j) \in \delta' &\quad \textit{iff} \quad (q_i, a, q_j) \in \delta \wedge a \in \Sigma_A \\
(q_i, \varepsilon, \varepsilon) \in \lambda' &\quad \textit{iff} \quad (q_i, a, x) \in \lambda \wedge a \notin \Sigma_A \\
(q_i, a, x) \in \lambda' &\quad \textit{iff} \quad (q_i, a, x) \in \lambda \wedge a \in \Sigma_A
\end{aligned}$$

Intuitively, all transitions of M on alphabet symbols not in Σ_A are replaced with non-deterministic transitions (ε), and the corresponding outputs are replaced with empty outputs (ε). The resulting state-machine may be non-deterministic.

Computing a projection of a state-machine inferred from communication of our bot emulator with one master server onto the alphabet of the machine inferred from communication with another master server, we identified the key components shared among multiple pools of bots. The results are presented in Section 6.2.

5.2 Identifying Design Flaws

We identified a design flaw in the MegaD protocol, thanks to the fact that our inference approach infers complete state-machines. Given a complete state-machine and a specification (i.e., a set of properties expressed in a suitable formal logic), it is possible to automatically determine whether the properties hold using automatic model checkers (e.g., [9]). In our case, the state-machines were simple enough that we could manually check a number of interesting properties. We explain the flaw we found later in Section 6.3.

5.3 Identifying Background-Channels

In situations when a client (a bot, in our case) talks to multiple servers, it might be interesting to prove whether there exists any background communication between the servers. Such background communication channels can indicate infiltration traps, which security researchers need to be aware of before attempting to bring a botnet down, or simply reveal interesting information about the protocol.

To detect the background-channels, we devised the following analysis: We restrict our bot emulator to communicate only with a single server at the time, and infer the protocol model M_T (for the template server), M_S (for the SMTP server), and M_M (for the master server). Then, we allow our bot emulator to communicate with all the servers and compute the model M. We compute the projection (Definition 9) of M onto input alphabets used for building individual server communication models (M_T, M_S, and M_M), and compare the obtained projection with the model of communication with the individual servers. Any differences imply that there exist background communication channels. We prove existence of communication between MegaD's servers in Section 6.4.

5.4 Identifying Implementation Differences

Once the complete models of two different implementations of the same protocol are computed, comparison of the models can reveal interesting deviations useful for fingerprinting and flaw detection. While it is possible to perform automatic equivalence checking of large finite-state models (e.g., [24]), our models were simple enough that we can do such an analysis manually. We discuss the differences between Postfix SMTP 2.5.5 and MegaD's implementation in Section 6.5.

6. EXPERIMENTAL EVALUATION

We implemented our version of $L*$ in $\sim$ 1.7 KLOC of C++ and the bot emulator and experimental infrastructure in $\sim$ 2.3 KLOC of Python and Bash scripts. Our prototype performs up to eight parallel queries (as shown in Figure 3) concurrently tunneled through Tor [15]. We conducted the experiments over a period of three weeks starting March 27^{th}, 2010. Figure 5 illustrates the inferred MegaD protocol state-machine.

In the rest of this section, we evaluate our protocol inference approach on the MegaD botnet C&C distributed system, MegaD's non-standard implementation of SMTP, and the standard SMTP as implemented in Postfix 2.5.5. We present the results of our analysis of inferred complete models, and validate our inference approach by comparing the inferred SMTP models against the SMTP standard.

6.1 Performance and Accuracy

In this section, we present the experimental evidence of the effectiveness of our response prediction technique and discuss the model accuracy.

[8] The conversion of Mealy to Moore machines introduces additional states, usually one extra state per transition. Hence the two concepts, bad transitions in Mealy machines and bad states in Moore machines are, as a matter of fact, equivalent concepts. Thus, talking about *bad transitions*, as opposed to *bad states*, makes more sense in our setting.

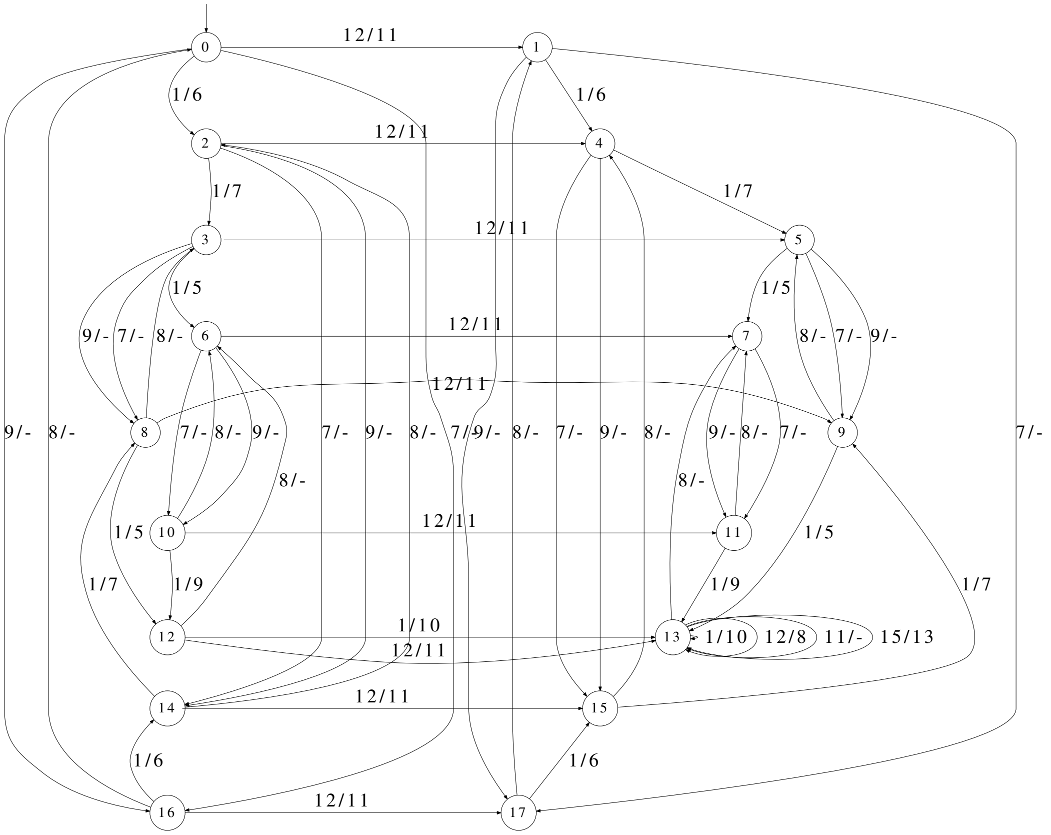


Figure 5: Protocol State-Machine of MegaD's distributed Command and Control system. Self-transition edges are removed for clarity. The state-machine transitions are labeled according to the alphabet in Table 1. For example, 15/13 denotes NOTIFY / NOTIFY_RECVED. The process of spamming is triggered in state 13, through self-edges 1/10, 12/8, 11/-, 15/13.

The prediction results are shown in Table 2. The overall reduction in the number of queries that have to be sent over the network is between 24.5% (for MegaD's SMTP) and 86.1% (for MegaD's C&C). We believe there are two main reasons our prediction is much more effective on the MegaD C&C than on SMTP: First, C&C is a more complex protocol that involves three different types of servers, two of which use proprietary protocols. Second, our understanding of the two protocols when we were designing message abstractions was very different — we knew nothing about the C&C state-machine, while the SMTP state-machine is well known [23]. We believe this inherent lack of knowledge about an unknown protocol model, yet to be inferred, results in some amount of redundancy. However, it is important to be conservative when abstracting messages, as otherwise it is easy to miss important states and transitions. This inherent tradeoff between accuracy and redundancy makes our prediction technique even more valuable, as we can infer larger protocols, without sacrificing accuracy. As a matter of fact, since sending and receiving a single message through Tor took around 6.8 seconds on average, 86.1% prediction accuracy means that our response predictor saved $\frac{(56{,}716-6{,}406)\times 6.8}{3600\times 24} = 3.95$ days of computation, reducing the total amount of time required to infer the MegaD C&C to around 12 hours.

Parallelization of the experiment improved the performance even further. While a single bot emulator would return a response message every 6.8 seconds, on average, eight parallel bots would return a message every 1.4 seconds, a 4.85X improvement on average in addition to improvements obtained by response prediction. The experiment does not parallelize perfectly, because of the increased load on the same Tor servers. We envision that a more powerful, perhaps even highly-distributed, protocol inference infrastructure would parallelize even better.

To check the accuracy of our models, we used $\varepsilon = 10^{-2}$ error probability and $\gamma = 10^{-6}$ confidence factors [3]. Achieving that accuracy required us to generate at least 1,798 uniformly random sampling queries for the MegaD C&C and 1,451 for MegaD and Postfix's SMTP upon $L*$ termination. Equivalence queries were ran in parallel and cached, but not predicted, as predicting those queries would defeat the purpose of such queries (detecting missed states and prediction errors).

6.2 Analysis of Critical Links

Attempts to bring down large botnets are frequent, but are costly and ineffective. The common practice is to run as many pools of bots as possible to obtain a wide coverage of C&C servers, and then attempt to cripple the entire botnet by sending abuse notifications to all ISPs involved simultaneously. This is an expensive exercise requiring careful co-ordination among all parties involved. Based on our analysis of critical links in the MegaD protocol, we discovered a less expensive alternative.

Using the technique explained in Section 5.1, we inferred complete models of communication with MegaD's two different master servers, and computed a projection of one model onto the alphabet of the other model as shown in Figure 6. The figure shows the states and links shared by two different pools of bots talking to *different* master servers and the servers that the master server points

	MegaD C&C		MegaD SMTP		STD SMTP	
	Q.	Msgs	Q.	Msgs	Q.	Msgs
Basic $L*$	10,978	56,716	1,190	4,522	1,386	5,894
RESTR	-8,024	-42,872	-294	-980	-476	-1764
STAT	-1,456	-7,514	-22	-88	-0	-0
BCKT	+24	+76	+24	+90	+56	+252
Total	1,522	6,406	898	3,544	966	4,382
Reduct.	-86.1%	-88.7%	-24.5%	-21.6%	-30.3%	-25.7%
Accur.	99.7%	99.9%	92.4%	97.8%	88.2%	96.8%

Table 2: Results of Membership Queries Prediction. The Queries (Q.) column shows the number of queries and the Msgs column shows the number of messages. The first row represents the results obtained in the standard $L*$ algorithm [33] without any response prediction. The RESTR row shows the reduction in the number of queries and messages by using restriction-based prediction. The following row (STAT) shows additional reductions obtained by using the statistics-based approach a top of RESTR. The BCKT row shows the increase in the number of queries and messages sent due to backtracking, caused by prediction errors. The Total row shows the total numbers of queries and messages after prediction reductions and increases due to backtracking. The total reduction in the number of queries and messages and the accuracy of the prediction are shown in the last two rows.

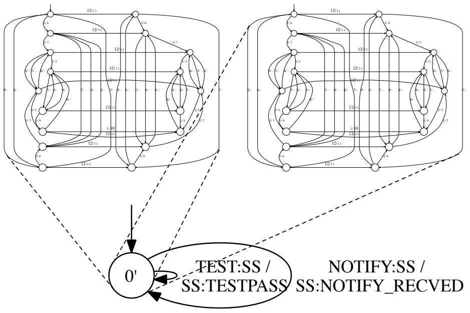


Figure 6: Intersection of Models of Communication with Two Different Master Servers. The resulting state-machine shows that the SMTP server is shared among two pools of bots communicating with two different master servers.

to. The projection shows that the SMTP server is shared across two pools of bots belonging to different master servers. Furthermore, the existence of the SMTP server is critical to MegaD's ability to spam. In particular, a successful pre-spam notification dialog with the shared SMTP server (the NOTIFY:SS/SS:NOTIFY_RECVED edge) *always* signals to the bot to start spamming. We experimentally confirmed that MegaD bots do not start spamming without this notification. Thus, the analysis clearly shows that taking down the SMTP server would disable spamming in all the pools of bots sharing that SMTP server. However, without actually attempting to execute an attack on the SMTP part of MegaD's infrastructure, it is difficult to evaluate how useful our insight is in practice. For example, the botmasters could replace the SMTP servers and use master servers to update bots on the new SMTP server locations.

6.3 Analysis of Design Flaws

Having a complete model of MegaD's C&C enabled us to check a number of properties on the inferred state-machine. In particular, we found that there is an unexpectedly short path through the state-machine to getting the templates from the template sever.

During a normal spam cycle, a MegaD bot would take the following path $0 \rightarrow 16 \rightarrow 14 \rightarrow 8 \rightarrow 12 \rightarrow 13$ to the spamming state 13. Upon reaching state 13, the bot sends the GET_COMMAND message, to which the master server responds with the location of a chosen template server. The bot then sends the GET_TEMPLATE request to the template server together with a 16-byte bot identifier issued by the master server, and receives templates as a response.

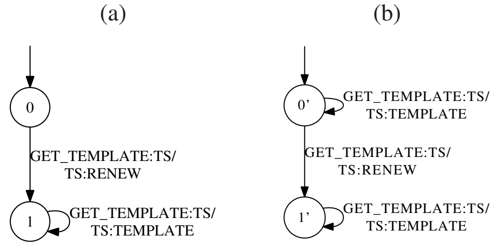


Figure 7: A Proof of the Existence of Background-Channel Communication. (a) A Model Obtained Only from the Communication with the Template Server. (b) The Projection of the C&C State-Machine onto the Template Server Alphabet. States {0,2,3,6,7,8,10,11,12,13,14,16} and {1,4,5,9,15,17} of the C&C in Figure 5 are projected onto states 0' and 1' respectively. Notice the state-machine is non-deterministic.

Our inferred C&C protocol model reveals the existence of multiple shorter paths to obtaining spam templates. In particular, the shortest path is $0 \rightarrow 1$, along which the bot emulator bypasses the master server, directly contacts the template server with a random 16-byte identifier, and obtains the templates while bypassing the master server. We successfully exploited this protocol design flaw, regularly obtaining fresh spam templates.

Without knowing the botnet developer's intentions, one may argue that what we discover may be a "feature" instead of a "flaw". For instance, the "feature" might facilitate reconfiguration of the template server location. However, we rule out such a possibility because the protocol uses an encryption-protected 16-byte identifier issued by the master server in the GET_TEMPLATE request. Since this identifier can be spoofed, it is clearly a flaw.

6.4 Analysis of Background-Channels

Our analysis discovered that the template server behaves differently depending on whether the bot communicated with the master server or not. If the bot talks only to the template server (Figure 7a) by sending a GET_TEMPLATE request, the template server responds with RENEW, and sends the templates only after the second request. On the other hand, if the bot talks to the template server after the regular message exchange with the master server, the template responds to the first GET_TEMPLATE request immediately with spam templates. This difference proves that there exists communication between the master and template servers. The model of this communication is shown in Figure 7b.

6.5 Analysis of Implementation Differences

We cross-checked the results of our inference with the SMTP standard [23], and compared the inferred model with the MegaD's non-standard SMTP implementation. To test the standard SMTP inference, we set up a Postfix SMTP 2.5.5 server on Mac OS X 10.6.3 and ran our inference technique against the server, using the standard set of 14 SMTP commands and input types from the SMTP specification [23], shown in Figure 8c. Figure 8a shows the inferred protocol model. For clarity, we removed all self-loop transition edges in the figure. As discussed in Section 6.1, the discrepancy is at most $\varepsilon = 10^{-2}$ with confidence $\gamma = 10^{-6}$.

Our evaluation on standard SMTP shows that the protocol model obtained from our inference technique is equivalent to the SMTP standard, with one implementation-specific deviation. We found and confirmed that the Postfix SMTP 2.5.5 implementation deviates from the SMTP standard by terminating a connection with response code '221' when it receives email contents sent without a prior DATA command; the standard specifies keeping the connection open while returning '500' to indicate an unrecognized command. We further evaluated our model by comparing it against the SMTP model inferred by Prospex [10]. We manually translated our inferred Mealy machine into Prospex's representation (Moore machine). The comparison reveals that a number of edges are missing from the SMTP model that Prospex inferred. On inter-state transitions alone, our complete SMTP model specifies the behavior of ten edges that Prospex missed. For example, the behavior of sending email contents without prior HELO or EHLO is described in our model, but not in [10]. We bold these missing edges in Figure 8a. Clearly, our technology is able to infer complete models, unlike the prior work.

Our model completely captures the protocol and is smaller (5 states) than the incomplete Prospex model (10 states). There are two reasons for this. First, we use Mealy machines, which are more compact. Second, since we infer complete minimal machines, minimization of the state-machine can be done completely and thoroughly. For example, sending either HELO or EHLO from the initial state transits to a single state in our model instead of two different states.

We also compared MegaD's non-standard SMTP using the same standard set of 14 SMTP commands and input types that we used to infer the standard SMTP protocol model. The inferred MegaD non-standard SMTP protocol model is shown in Figure 8b. A comparison of Figures 8a and 8b yields a high degree of resemblance, with two major differences: (1) States 0 and 1 in Figure 8a are merged into a single state in Figure 8b, which indicates that the usual SMTP dialog with MegaD's SMTP server may take place without a prior HELO or EHLO. (2) The server abruptly closes the connection once it receives a data terminator '.' without a prior DATA command. This result suggests that our protocol model inference technique may be applied to fingerprint and identify MegaD's SMTP masquerading servers.

7. LIMITATIONS

In this section, we highlight the limitations of our protocol model inference and analysis techniques, and discuss possible directions for future work.

Currently, we make no effort to obfuscate and hide our probing traffic from the botmaster's possible detection. Since our protocol inference approach generates a large number of probing queries to the botnet C&C servers, the botmaster could potentially detect our queries. The botmaster may react by changing the locations or protocol configurations of the C&C servers, thwarting our inference effort. Hiding our traffic in the background noise would be an interesting area for future research.

We also note that our protocol state-machine inference approach would not work if any assumptions laid out in Section 2.3 are violated. The determinism and finiteness assumptions are the most limiting.

A protocol that behaves non-deterministically (e.g., a date/time triggered behavior) is more challenging to infer. As discussed in Section 2.3, one option is to discretize time and encode it directly into the alphabet. Despite obvious limitations, our conjecture is that such an approach could be sufficient for inferring a majority of existing protocols. Only future research can (dis)prove our conjecture.

While infinite-state protocols can be abstracted with finite-state machines (c.f. Mohri-Nederhof algorithm for abstracting context-free with regular ones [27]), such abstractions might not be precise enough for all potential applications. To make things worse, many protocols have mildly-context-sensitive features, like buffer lengths. The grammatical inference techniques [14] for such more expressive languages are still in their infancy. Inference of more expressive models is a promising research direction, not only in the context of protocol inference.

The focus of our work is the model inference technique itself, and we relied upon prior work [6] and manual abstraction to come up with the alphabet, which might be incomplete, i.e., it might not contain all messages that can cause a state-change in the protocol. The clustering of messages in a single direction for abstraction can be automatic [10], but cannot guarantee the completeness of the alphabet either. To our knowledge, automatic reverse-engineering of the complete alphabet is an open problem.

8. RELATED WORK

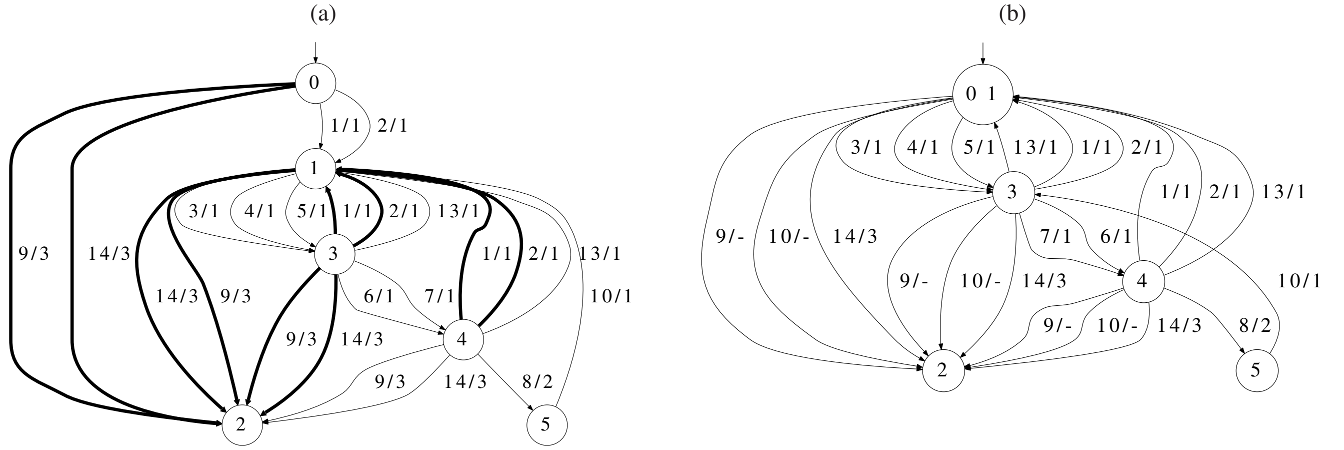
The work presented in this paper is in the intersection of protocol model inference and grammatical inference, and also contributes to the previous work in the area of analysis of botnets (e.g., [2, 17, 18, 19, 22]).

The most closely related work, to our knowledge, is the work of Comparetti et al. [10] on the Prospex system. The automatic clustering and abstraction feature in Prospex is more advanced than the manual abstraction that we did, so our future work is likely going to focus on improving that aspect of our system. Unlike our approach, Prospex adopts passive off-line inference and models protocols with Moore machines. These choices have a number of consequences: (1) Inferred models are incomplete state-machines, which means that any subsequent analysis is bound to be imprecise. In contrast, our approach infers complete state machines. (2) Since protocols are reactive systems, there are no accepting or rejecting states. To work around that problem, Prospex differentiates states using regular expressions describing messages received before each state is reached. The regular expression labels prevent the incomplete state-machine minimization algorithms, like Exbar [25], to merge all the states into a single state. We avoid this problem altogether by using a model more appropriate for reactive systems — Mealy machines and $L*$ algorithm guarantee that the inferred machine is minimal. (3) The minimization of incomplete state-machines is a known NP-complete problem [30], so it is questionable whether Prospex would scale to large complex protocols. Our approach dodges the NP-completeness using proactive inference, at the cost of a small probability of error ($\varepsilon = 10^{-2}$) with high confidence ($\gamma = 10^{-6}$) [3].

Earlier work on protocol inference by Hsu et al. [21] does use Mealy machines, but also adopts off-line inference, which means that the inferred models will be incomplete and minimization is NP-complete. Their solution is an approximate algorithm. Unfortunately, even computing a model that is within a polynomial size of the minimal one is NP-complete [34, p. 98–99], meaning that their polynomial-time approximation algorithm will compute very large models in some cases. In contrast, we developed a version of $L*$ optimized for on-line protocol inference, with known advantages over the off-line approaches (polynomial computational complexity, completeness).

The results presented in this paper would not have been possible without the prior research on automatic message format reverse-engineering by Cui et al. [12, 13] and the work of Caballero et al. [7], which we used in this paper. All these techniques are crucial for both manual and automatic message abstraction into finite alphabets, so we are looking forward to the further progress in research on automatic message format reverse-engineering.

Another aspect important in general protocol model inference



(c)

ID	Semantics	Direction	ID	Semantics	Direction	ID	Semantics	Direction
1	HELO 1	→ SS	7	RCPT TO:<@a,@b:addr>	→ SS	13	RSET	→ SS
2	EHLO 1	→ SS	8	DATA	→ SS	14	QUIT	→ SS
3	MAIL FROM:<addr>	→ SS	9	content	→ SS	1	250	← SS
4	MAIL FROM:<>	→ SS	10	\r\n.\r\n	→ SS	2	354	← SS
5	MAIL FROM:<@a,@b:addr>	→ SS	11	EXPN all	→ SS	3	221	← SS
6	RCPT TO:<addr>	→ SS	12	VRFY usr	→ SS			

Figure 8: Inferred SMTP State Machines: (a) Postfix SMTP 2.5.5, (b) MegaD non-Standard SMTP and (c) Abstraction Table.

is dealing with encryption. Caballero et al. [6] recently proposed an automatic technique for extracting encryption routines from binary. Similarly, Wang et al. [36] deals with reverse-engineering of encrypted messages.

Once the protocol model is known, it can be used to incorporate into stateful protocol analyzers, like Bro [29] and GAPA [4], and firewalls, like Shield [35]. All these systems require protocol specifications to analyze traffic, detect intrusions, and improve security. The technology we developed can provide such specifications.

Our contributions to the field of grammatical inference (more precisely, regular language inference) extend the previous work of Shahbaz and Groz [33], by specializing their approach to protocol inference, and by a number of optimizations for decreasing the number of membership queries, which are expensive in the real network setting. While parallelization of $L*$ and introduction of a cache for concentrating results of parallel probes came as natural optimizations suitable for our setting, the output symbol prediction required more intellectual effort. Our inspiration came from the recent work of Gupta and McMillan [20]. They applied decision-trees [31], a standard machine-learning technique, to complete incomplete state-machines learned in the hardware verification setting for the purpose of abstracting hardware modules and achieving compositional verification.

9. CONCLUSIONS

We have proposed, to the best of our knowledge, the first technique to infer complete protocol state machines in the realistic high-latency network setting, and applied it to the analysis of botnet C&C protocols. While the classic $L*$ algorithm would have taken 4.46 days to infer the protocol model of the MegaD C&C distributed system, we introduced a novel and effective prediction technique to minimize the number of queries generated during the inference process, reducing the amount of time required to just 12 hours. This is further optimized through parallelization.

By analyzing the complete protocol models inferred by our technique, we offer novel insights to existing problems on botnet C&Cs. We hope that our new insights, gained through our protocol inference technique and novel analyses, will render future attacks on MegaD and other botnets cheaper and more effective. With the new technology to fight botnets we developed, we hope to see a decrease in the amount of spam and denial of service attacks, and an increase of everyone's productivity and security.

10. ACKNOWLEDGMENTS

We would like to thank Greg Bronevetsky, Vern Paxson, Nishant Sinha, and the anonymous reviewers for insightful comments to improve this manuscript.

11. REFERENCES

[1] 2007 malware report: The economic impact of viruses, spyware, adware, botnets, and other malicious code. Technical report, Computer Economics Inc., 2007.

[2] Moheeb Abu Rajab, Jay Zarfoss, Fabian Monrose, and Andreas Terzis. A multifaceted approach to understanding the botnet phenomenon. In *IMC '06: Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 41–52, New York, NY, USA, 2006. ACM.

[3] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[4] Nikita Borisov, David Brumley, Helen J. Wang, John Dunagan, Pallavi Joshi, and Chuanxiong Guo. Generic application-level protocol analyzer and its language. In *NDSS'07: Proceedings of the 2007 Network and Distributed System Security Symposium*. The Internet Society, Feb 2007.

[5] Juan Caballero, Noah M. Johnson, Stephen McCamant, and Dawn Song. Binary code extraction and interface identification for security applications. In *NDSS'10: Proceedings of the 17th Annual Network and Distributed System Security Symposium*, Feb 2010.

[6] Juan Caballero, Pongsin Poosankam, Christian Kreibich, and Dawn Song. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *CCS'09:*

Proceedings of the 16th ACM conference on Computer and communications security, pages 621–634, New York, NY, USA, 2009. ACM.

[7] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *CCS'07: Proceedings of the 14th ACM Conference on Computer and Communications Security*, pages 317–329, New York, NY, USA, 2007. ACM.

[8] Chia Yuan Cho, Juan Caballero, Chris Grier, Vern Paxson, and Dawn Song. Insights from the inside: A view of botnet management from infiltration. In *LEET'10: Proceedings of the 3rd USENIX Workshop on Large-Scale Exploits and Emergent Threats*, pages 1–1, Berkeley, CA, USA, 2010. USENIX Association.

[9] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.

[10] Paolo Milani Comparetti, Gilbert Wondracek, Christopher Kruegel, and Engin Kirda. Prospex: Protocol specification extraction. In *SP'09: Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, pages 110–125, Washington, DC, USA, 2009. IEEE Computer Society.

[11] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

[12] Weidong Cui, Jayanthkumar Kannan, and Helen J. Wang. Discoverer: Automatic protocol reverse engineering from network traces. In *SS'07: Proceedings of 16th USENIX Security Symposium*, pages 1–14, Berkeley, CA, USA, 2007. USENIX Association.

[13] Weidong Cui, Marcus Peinado, Karl Chen, Helen J. Wang, and Luis Irún-Briz. Tupni: Automatic reverse engineering of input formats. In *CCS'08: Proceedings of the 15th ACM Conference on Computer and Communications Security*, pages 391–402. ACM, Oct 2008.

[14] Colin de la Higuera. *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, 2010.

[15] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: the second-generation onion router. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium*, pages 21–21, Berkeley, CA, USA, 2004. USENIX Association.

[16] E. Mark Gold. Complexity of automaton identification from given data. *Information and Control*, 37(3):302–320, 1978.

[17] Julian B. Grizzard, Vikram Sharma, Chris Nunnery, Brent ByungHoon Kang, and David Dagon. Peer-to-peer botnets: overview and case study. In *HotBots'07: Proceedings of the 1st Workshop on Hot Topics in Understanding Botnets*, pages 1–1, Berkeley, CA, USA, 2007. USENIX Association.

[18] Guofei Gu, Roberto Perdisci, Junjie Zhang, and Wenke Lee. Botminer: clustering analysis of network traffic for protocol- and structure-independent botnet detection. In *SS'08: Proceedings of the 17th conference on Security symposium*, pages 139–154, Berkeley, CA, USA, 2008. USENIX Association.

[19] Guofei Gu, Phillip Porras, Vinod Yegneswaran, Martin Fong, and Wenke Lee. Bothunter: detecting malware infection through IDS-driven dialog correlation. In *SS'07: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, pages 1–16, Berkeley, CA, USA, 2007. USENIX Association.

[20] Anubhav Gupta, K. L. McMillan, and Zhaohui Fu. Automated assumption generation for compositional verification. *Form. Methods Syst. Des.*, 32(3):285–301, 2008.

[21] Tating Hsu, Guoqiang Shu, and David Lee. A model-based approach to security flaw detection of network protocol implementation. In *ICNP'08: Proceedings of the 15th IEEE International Conference on Network Protocols*, pages 114–123, Oct 2008.

[22] Anestis Karasaridis, Brian Rexroad, and David Hoeflin. Wide-scale botnet detection and characterization. In *HotBots'07: Proceedings of the 1st Workshop on Hot Topics in Understanding Botnets*, pages 7–7, Berkeley, CA, USA, 2007. USENIX Association.

[23] J. Klensin. RFC 5321: Simple Mail Transfer Protocol, Oct 2008.

[24] Andreas Kuehlmann and Florian Krohm. Equivalence checking using cuts and heaps. In *DAC'97: Proceedings of the 34th annual Design Automation Conference*, pages 263–268, New York, NY, USA, 1997. ACM.

[25] Kevin J. Lang. Faster algorithms for finding minimal consistent DFAs. Technical report, NEC, 1999.

[26] George H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955.

[27] Mehryar Mohri and Mark-Jan Nederhof. Regular approximation of context-free grammars through transformation. In *Robustness in Language and Speech Technology*, pages 153–163. Kluwer Academic Publishers, Dordrecht, 2001.

[28] E. F. Moore. Gedanken Experiments On Sequential Machines. In *Automata Studies, Annals of Mathematical Studies*, volume 34, pages 129–153, Princeton, NJ, USA, 1956. Princeton University Press.

[29] Vern Paxson. Bro: a system for detecting network intruders in real-time. In *SSYM'98: Proceedings of the 7th conference on USENIX Security Symposium*, pages 3–3, Berkeley, CA, USA, 1998. USENIX Association.

[30] C. P. Pfleeger. State reduction in incompletely specified finite-state machines. *IEEE Transactions on Computers*, 22(12):1099–1102, 1973.

[31] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

[32] R. L. Rivest and R. E. Schapire. Inference of finite automata using homing sequences. In *STOC'89: Proceedings of the 21st annual ACM symposium on Theory of computing*, pages 411–420, New York, NY, USA, 1989. ACM.

[33] Muzammil Shahbaz and Roland Groz. Inferring Mealy machines. In *FM'09: Proceedings of the 2nd World Congress on Formal Methods*, pages 207–222, Berlin, Heidelberg, 2009. Springer.

[34] B. A. Trakhtenbrot and Ya. M. Barzdin. *Finite Automata, Behavior and Synthesis*. North Holland, Amsterdam, 1973.

[35] Helen J. Wang, Chuanxiong Guo, Daniel R. Simon, and Alf Zugenmaier. Shield: vulnerability-driven network filters for preventing known vulnerability exploits. *SIGCOMM Computer Communication Review*, 34(4):193–204, 2004.

[36] Zhi Wang, Xuxian Jiang, Weidong Cui, Xinyuan Wang, and Mike Grace. ReFormat: Automatic Reverse Engineering of Encrypted Messages. In *ESORICS'09: Proceedings of the 14th European Symposium on Research in Computer Security*, volume 5789 of *Lecture Notes in Computer Science*, pages 200–215. Springer, Sep 2009.