PISE: Protocol Inference Using Symbolic Execution and Automata Learning

Ron Marcovich Orna Grumberg Gabi Nakibly
Technion - Israel Institute of Technology
{ron.mar,orna,gnakibly}@cs.technion.ac.il

***Abstract*—Protocol Inference is the process of gaining information about a protocol from a binary code that implements it. This process is useful in cases such as extraction of the command and control protocol of a malware, uncovering security vulnerabilities in a network protocol implementation or verifying conformance to the protocol's standard. Protocol inference usually involves time-consuming work to manually reverse engineer the binary code.**

We present a novel method to automatically infer state machine of a network protocol and its message formats directly from the binary code. To the best of our knowledge, this is the first method to achieve this solely based on a binary code of a single peer. We do not assume any of the following: access to a remote peer, access to captures of the protocol's traffic, and prior knowledge of message formats. The method leverages extensions to symbolic execution and novel modifications to automata learning. We validate the proposed method by inferring real-world protocols including the C&C protocol of Gh0st RAT, a well-known malware.

I. Introduction

The process of gaining information about a communication protocol from the binary code that implements it is called *Protocol Inference*. Such a process is employed in several practical scenarios. Many malwares set up a command and control (C&C) channel with the attacker's server. Over that channel they receive commands from the server and send information gathered from the victim machine. Oftentimes, knowing the commands expected to be received by the malware is helpful to analyze the goals and logic of the malware. Nonetheless, these commands may not be easily obtained if no prior traffic of the C&C channel has been captured or the attacker server is no longer operational. In such cases one needs to infer the C&C protocol from the malware's binary only.

Additionally, security vulnerabilities of a network protocol software are often caused by deviations from the intended protocol's logic. For example, a vulnerability in an implementation of a server program may accept data from the client before the user was authenticated or after a message that closes the connection is received. Such deviations may be even intentional and pose an undesirable back door to the program. Moreover, improper implementation of a protocol logic may lead to non-conformance to other implementations of the same protocol. To reveal such improper implementations of a protocol, one needs to infer the protocol implemented by the software and compare it to the protocol's desired logic.

In this work we propose PISE: a method to automatically infer the protocol directly from a given binary code. The method extracts the protocol state machine and the formats of each of the protocol's message types. The method leverages an extended L* algorithm [2] to learn the protocol's state machine.

L* is based on a Teacher and a Learner, whose goal is to reveal the state machine of an unknown language (in our case, the sequences of the protocol's message exchanges). The Learner presents membership queries and equivalence queries that the Teacher answer.

To answer the algorithm's membership queries we use modified symbolic execution of the binary program. The modifications track the program's network activity and guide the symbolic execution through a messages exchange according to the Learner's query. The symbolic execution is also leveraged to uncover messages that may follow a valid message exchange. Predicates representing the protocol's message types are derived based on these example messages. As new message types are revealed they are fed back to the L* algorithm to extend the protocol's state machine. Equivalence queries of the L* algorithm are answered using a standard sampling approximation using membership queries.

We emphasize that the only input to the proposed method is the binary code of a single peer of the protocol. In particular, we do not assume: (1) Access to the binary code of the remote peer. (2) Access to network traffic recordings that contain valid protocol sessions. (3) Access to an online instance of the remote peer. Namely, we cannot recreate valid session traffic. (4) Prior knowledge of messages' formats and the partition to message types.

To the best of our knowledge, PISE is the first protocol inference method that has none of the above assumptions. We believe that the lack of such assumptions makes our method widely suitable for many real-world use cases of protocol inference. For example, often when the C&C protocol of a malware is analyzed, the protocol peer, i.e. the C&C server, is unobtainable nor it is online, furthermore past malware traffic has not been recorded. Hence, the binary code of the malware is the only source of information about the protocol.

Workshop on Binary Analysis Research (BAR) 2023
3 March 2023, San Diego, CA, USA
ISBN 1-891562-84-3
https://dx.doi.org/10.14722/bar.2023.23002
www.ndss-symposium.org

II. Previous Works

There are several published works that deal with the problem of learning information about a network protocol. The approach presented in [8] uses recorded network traffic as input. It analyzes the traffic and uses heuristics to extract different protocol fields.

To infer information about the protocol that may not be captured in recorded traffic only, several methods [4] [3] [17] [9] were introduced that combine the recorded traffic with execution traces of the server, allowing them to learn more about the messages' formats, and even gain some insights about the semantics of messages and message fields.

All of these works do not deal with the problem of learning the protocol state machine. This was the motivation for the work [7] which introduced Prospex. This work extended previous works in two directions: First, they developed a mechanism to identify messages of the same type. They use this information to partition messages with similar role in the protocol into clusters. The second extension is a method to infer the state machine of the protocol. Prospex method, like all the mentioned methods, requires captures of protocol's traffic.

Another important contribution is the work [6] that introduced a method for on-line inference of botnet C&C protocol, using active instances of it. They chose to represent a protocol as a Mealy machine and used L* extension by [14] for learning mealy machines. They actively query the control server responses for a sequences of messages. They also introduced caching and prediction optimizations to L* in order to reduce the amount of membership queries sent to the server. Their work, as an on-line method, assumes the server is available and answers appropriately. They also assume known message formats by using previous work [4].

Another related work is the work [1] that introduces a method to infer a Java class specification (order of method calls) using model checking and L*, which is similar to what we apply in our work. In their case, however, the alphabet (the methods) is known in advance. The work [10] utilizes L* for the purpose of model checking and suggests learning-based algorithms to automate assumption generation in assume-guarantee verification. The work [12] even extended this to include alphabet refinement, a technique to infer interface alphabets.

Another work called MACE [5] presents a method to learn a state machine of a server using L* and symbolic execution. They use L* extension for inferring Mealy state machines [14] and like our method, use symbolic execution to uncover messages that the client may send. There are, however, two main differences to our method: First, MACE assumes a known abstraction function is available that can extract the message type out of the server's response. Second, MACE assumes a running server is available, that can answer client requests. We do not have these assumptions.

III. Preliminaries

A. Deterministic Finite Automaton (DFA) Learning

A deterministic finite automaton (DFA) M is a five-tuple, $(Q, \Sigma, \delta, q_0, F)$, all of them nonempty, whereas: Q is a finite set of states, Σ is a finite set of input symbols (alphabet), $\delta : Q \times \Sigma \to Q$ is a transition function, $q_0 \in Q$ is an initial state and $F \subseteq Q$ is a set of accept states. Let $w = \sigma_1 ... \sigma_n$ be a string over the alphabet Σ. We say that M accepts w if there exist $r_0, ..., r_n \in Q$ such that $r_0 = q_0$, $r_n \in F$ and $\forall 0 \leq i \leq n-1, \quad r_{i+1} = \delta(r_i, \sigma_{i+1})$.

Let $\Sigma^* = \{\sigma_1 ... \sigma_n \quad | \ \sigma_i \in \Sigma, n \geq 0\}$ be the set of all finite strings over the alphabet Σ. We define $L(M) = \{w \in \Sigma^* \mid M \text{ accepts } w\}$ as the language of M. A set of words $L \subseteq \Sigma^*$ is a *regular language* iff there exists DFA M such that $L = L(M)$.

Automata learning identifies an unknown regular language L by learning a DFA M such that $L(M) = L$. The L* algorithm [2] solves this problem. It assumes a *minimally adequate Teacher*, which is an oracle that can answer two types of queries: *Membership query* and *Equivalence query*. In a Membership query, the Teacher should indicate whether a given word w is in L or not. In an Equivalence query, the Teacher should indicate, given a conjectured DFA M', whether $L(M') = L$, and provide a counterexample otherwise (a word in the symmetric difference of L and $L(M')$).

In its internal data, the L* algorithm saves a description of the currently learned automaton in a structure called observation table. The observation table is updated during the learning process according to the answers of the Teacher.

The basic L* algorithm assumes Σ (alphabet) is known. Several works [12] [13] propose extended algorithms to deal with an alphabet that is revealed (or grows) during the execution of the algorithm.

B. Symbolic Execution

Symbolic execution is a static method of analyzing a program. During the analysis it determines what constraints the program's input must satisfy in order to execute each execution path in the program. This is done by following the program's code assuming symbolic values for inputs. A symbolic state is defined to contain the current symbolic values for each variable in the program, as well as constraints that should hold in order to reach that state.

Symbolic execution begins with a single initial state located at the entry point of the program. The execution happens by stepping the set of states and generating new descendant states. Stepping a single state may result with multiple new descendant states, if, for example, the parent state corresponds to a conditional branch. Before stepping a state, the symbolic execution may verify that the current constraints of the state are satisfiable. This verification is done in order to discard states that represent infeasible paths, representing impossible executions. A symbolic state can represent an execution during which the program failed and terminated, namely exits with a non-zero return value. We call such a symbolic state an abort state.

IV. Problem Definition

Our goal, given a binary code of a program, is finding a DFA that accepts a language L, where each word in L matches a sequence of message types received and sent by the program. We say that this DFA is the state machine of the protocol implemented by the program. We say that a concrete run of the program is valid if and only if it does not finish in an abort state.

We denote by S and R the finite set of messages that may be sent and received by the program, respectively. S and R are disjoint. S and R are finite because messages are limited in length (See Assumption 1 below).

A session is a sequence of messages $m_1 \dots m_k$ such that $\forall 1 \leq i \leq k,\ \ m_i \in S \cup R$. We say that a session $m_1 \dots m_k$ is valid for the program if and only if there exists a valid run of the program along which the same sequence of messages are sent or received in exactly the same order as in $m_1 \dots m_k$. An empty session is considered valid. Valid sessions are prefix-closed, meaning that if $m_1 \dots m_k$ is a valid session then $\forall 1 \leq l \leq k-1,\ \ m_1 \dots m_l$ is also a valid session.

Messages are partitioned into subsets of message types according to their semantics in the protocol and their effect on the protocol state machine. Let T_D be a partition of message types of D where $D \in \{R, S\}$. Given a message $m \in D$, we denote by $type(m)$ the pair $\langle D, t\rangle$ such that $t \in T_D$ and $m \in t$. We call such a pair $\langle D, t\rangle$ a **Message Type**. t is finite because both S and R are finite.

We define the alphabet of the protocol as a finite set of pairs:

$$\Sigma_L = \{\langle D, t\rangle \mid t \in T_D, D \in \{R, S\}\}$$

Given a session $m_1 \dots m_k$ such that $\forall 1 \leq i \leq k, m_i \in S \cup R$, we define $\Theta : (S \cup R)^* \to \Sigma_L^*$:

$$\Theta(m_1 \dots m_k) = type(m_1) \dots type(m_k)$$

We abstract all valid sessions to a regular language L over the alphabet Σ_L:

$$L = \{\Theta(m_1 \dots m_k) \mid m_1 \dots m_k \text{ is a valid session}\}$$

Note that L is prefix closed because valid sessions are prefix closed. Note that R and S, as well as their partitions T_S and T_R, are unknown in advance, hence the alphabet Σ of the DFA in unknown in advance. It is the task of our method to uncover Σ_L as it determines the DFA.

1) Assumptions: We assume the following about the input to our method:

1) **Message Length is limited:** there exists N such that no message in the protocol to be inferred is longer than N bytes. This is a reasonable assumption since concrete messages must be finite. In practice, our method allow a message to be longer than N bytes, as long as the first N bytes may allow to infer its message type. This assumption is required since symbolic lengths are computationally difficult to infer.
2) **Protocol Regularity:** the protocol can be modeled as a DFA (See Section III-A) over Σ_L in terms of message types allowed from each state. Formally, L is regular language. If L is not regular, our algorithm will fail or will never complete the inference.

V. Learning a DFA of a Protocol (Exact Version)

The learning of the protocol's DFA is based on a modified version of the L* algorithm [2]. We modify the algorithm's queries in order to uncover the alphabet Σ, that is, the message types of the protocol. Initially, Σ is $\emptyset$.

For ease of exposition, we assume here a Teacher that is capable of answering the queries we present. However, this assumption is unrealistic when both the state machine and the alphabet Σ_L are unknown. In Section VI we propose suitable approximations for the Learner and the Teacher.

1) Modified Membership Queries: The classical membership query returns $True$ or $False$, indicating if a given w is in L or not. We modify it as follows: If $w \in L$, $True$ is returned together with a set $Cont(w)$ of message types $\langle D, t\rangle$ such that $w \cdot \langle D, t\rangle \in L$. If $w \notin L$, $False$ is returned. The set $Cont(w)$ may reveal new alphabet symbols.

2) Modified Equivalence Queries: In a classical equivalence query the Learner provides a conjectured DFA M over alphabet $\Sigma_M = \Sigma$. $True$ is returned if $L = L(M)$. Otherwise, $False$ is returned and a counterexample w in the symmetric difference of L and $L(M)$ is returned as well. We modify it as follows: $False$ is returned if there exist $w \in \Sigma_M^*$ for which one of the following hold: (1) w is in the symmetric difference of L and $L(M)$; (2) A set $Miss(M) \neq \emptyset$ exists such that for all $\sigma \in Miss(M)$, $\sigma \notin \Sigma_M$ but $w \cdot \sigma \in L$. In the former case, w is returned as a counterexample. In the latter case, every $w \cdot \sigma$ is considered a counterexample. $True$ is returned if for all $w \in \Sigma_M^*$, neither (1) nor (2) hold.

3) Handling a growing alphabet: Given a set of message types $C = Cont(w)$ or $C = Miss(M)$ output by a query, Σ is set to $\Sigma \cup C$. If Σ changes during this assignment, then we say that the alphabet grows. To handle a growing alphabet we use a modified L* algorithm presented in [13]. In a nutshell, the modified algorithm updates the observation table to handle the new alphabet symbols while the general learning cycle is kept similar to the classical L* algorithm.

4) Initialization and Output: The Learner starts with $\Sigma = \emptyset$. The first query of the Learner is $w = \varepsilon$. Note that the answer to this query is $True$ since an empty session is valid. $Cont(\varepsilon)$ is then added to Σ. The Learner continues to utilize queries according to the L* algorithm and extends Σ and the learnt DFA according to the queries' answers. The algorithm terminates when an equivalence query returns $True$. The algorithm outputs the learnt DFA that represents the protocol's state machine and Σ that represents the protocol's message types.

5) Correctness: The correctness of the algorithm is based on the modified definition of equivalence queries and correctness of the classical L* algorithm.

Theorem 1: The modified Learner terminates with $L(M) = L$ and $\Sigma_M = \Sigma_L$.

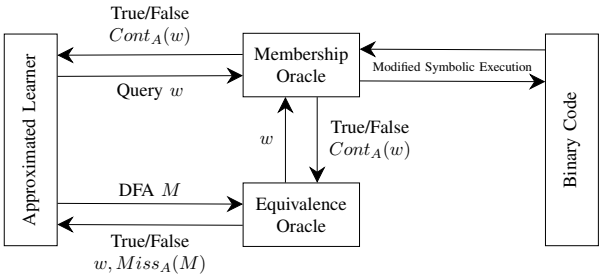

Fig. 1. Illustration of our algorithm

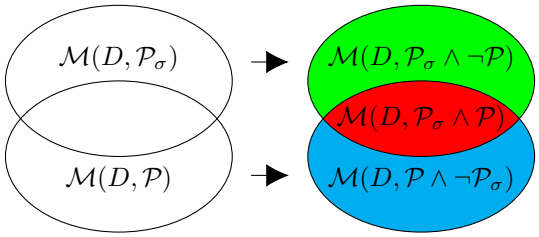

Fig. 2. Colliding predicates

Proof 1: The Learner terminates when it gets $True$ as an answer from the Teacher on an equivalence query. In this case, there is no $w \in \Sigma_M^*$, which is in the symmetric difference of L and $L(M)$. Thus, $L = L(M)$. Also, there is no $w \in \Sigma_{M^*}$, such that $\sigma \notin \Sigma_M$ but $w \cdot \sigma \in L$. This means that there is no (reachable) message type $\sigma \in \Sigma_L$ that has not been revealed already by our modified Learner. Consequently, $\Sigma = \Sigma_L$, as required. ■

VI. Learning a DFA of a Protocol (Approximation)

This section details how the answers to the queries of the modified L* algorithm presented in Section V are approximated. The components of the learning and their interactions are presented in Figure 1.

A. Characterizing Message Types

Recall that a message type is a pair $\langle D, t\rangle$ where t is a set of finitely many messages. We represent a set t using a predicate $\mathcal{P}$ describing the format of that message type. Hence, we represent a message type by a pair $\langle D, \mathcal{P}\rangle$. We use predicates over variables $\{B_0, \ldots, B_{N-1}\}$, representing the message bytes. Recall that N is a maximal length of a message (See Section IV-1). $m[i]$ denotes the value of the i-th byte of m, such that $0 \leq m[i] < 256$. We define $\mathcal{M}(D, \mathcal{P}_x)$ to be the set of messages from D that is matched by the predicate $\mathcal{P}_x$:

$$\mathcal{M}(D, \mathcal{P}_x) = \{m \in D \mid \mathcal{P}_x(m) = True\}$$

Given a set of messages $x \subseteq D$, $\mathcal{P}_x$ is extracted using the following simple definition: we hold constraints on message bytes that have the same value for all the messages in x. Formally, let $m \in x$, we define $\forall 0 \leq i \leq N-1$:

$$\varphi_i = \left\{ \begin{array}{ll} B_i = m[i], & \text{if } \forall m' \in x, m'[i] = m[i] \\ True, & \text{Otherwise} \end{array} \right\}$$

And $\mathcal{P}_x$ is defined as:

$$\mathcal{P}_x = \bigwedge_{i=0}^{N-1} \varphi_i$$

Note that the above definition may be replaced with a more elaborate one, if needed. We choose this definition because it is simple and is sufficiently useful for many real world protocols. In Section VI-E2c we explain how to generate predicates that are sufficiently general to describe message types even though we are given only a small subset of examples for that message type.

B. Handling Alphabet Changes

Recall that the exact version of our method is infeasible in our setting. To remedy this, we replace the sets of message types, $Cont(w)$ and $Miss(M)$, with their approximations, denoted $Cont_A(w)$ and $Miss_A(M)$, of *message type candidates*.

As we use an approximation to generate the new message type candidates, they may intersect with previously found message types currently in Σ. This breaks the assumption that sets of message types are pairwise disjoint. Therefore, we present here an algorithm that, given $C = Cont_A(w)$ or $C = Miss_A(M)$, incorporates C in Σ while making sure the elements of Σ remain pairwise disjoint. We denote by Σ the current alphabet and by Σ' the updated alphabet after the changes. The algorithm initializes $\Sigma' = \Sigma$.

Let $c = \langle D, \mathcal{P}\rangle \in C$ be a message type candidate. We say that $\langle D, \mathcal{P}\rangle$ collides with $\langle D_\sigma, \mathcal{P}_\sigma\rangle \in \Sigma$ if $D_\sigma = D$ and $\mathcal{M}(D, \mathcal{P}) \cap \mathcal{M}(D, \mathcal{P}_\sigma) \neq \emptyset$. In order to detect if c collides with σ, we check the satisfiability of $\mathcal{P} \wedge \mathcal{P}_\sigma$. If a collision is detected, then $\langle D_\sigma, \mathcal{P}_\sigma\rangle$ is removed from Σ and three message types are added to Σ': One that is based on a predicate of the intersection of the colliding message types and two message types that are based on the a symmetric differences of the colliding message types (see Figure 2). The procedure to handle collisions of a message type candidate c is presented in Algorithm 1.

Note that, if Σ originally includes only pairwise disjoint message types, then it is guaranteed that message types $\langle D, \mathcal{P}_\sigma \wedge \neg\mathcal{P}\rangle$ and $\langle D, \mathcal{P}_\sigma \wedge \mathcal{P}\rangle$ do not collide with any other message types in Σ. Therefore, it is only left to check for collision with $\langle D, \neg\mathcal{P}_\sigma \wedge \mathcal{P}\rangle$ in Σ'.

During the procedure we must discard unsatisfiable predicates. A predicate may become unsatisfiable in two special cases of collision: If $\mathcal{M}(D, \mathcal{P}_\sigma) \subset \mathcal{M}(D, \mathcal{P}')$ then $\mathcal{P}_\sigma \wedge \neg\mathcal{P}'$ is not satisfiable and should be discarded (line 6). If $\mathcal{M}(D, \mathcal{P}_\sigma) \supset \mathcal{M}(D, \mathcal{P}')$ then $\mathcal{P}' \wedge \neg\mathcal{P}_\sigma$ is not satisfiable and should not be inserted to Σ (line 13).

We run Algorithm 1 for every $c \in C$. After running this procedure for all message type candidates, the elements of the resulting Σ' are pairwise disjoint and are set as the new alphabet.

Algorithm 1 The procedure to handle message type candidate

function HANDLE_CANDIDATE($\langle D, \mathcal{P}\rangle \in C$, Σ)
  $\mathcal{P}' \leftarrow \mathcal{P}$, $\Sigma' \leftarrow \Sigma$
  for all $\langle D_\sigma, \mathcal{P}_\sigma\rangle \in \Sigma'$ **such that** $D = D_\sigma$ **do**
    if $\mathcal{P}_\sigma \wedge \mathcal{P}'$ is satisfiable **then**
      $\Sigma' \leftarrow \Sigma' \setminus \{\langle D, \mathcal{P}_\sigma\rangle\}$
      if $\mathcal{P}_\sigma \wedge \neg\mathcal{P}'$ is satisfiable **then**
        $\Sigma' \leftarrow \Sigma' \cup \{\langle D, \mathcal{P}_\sigma \wedge \neg\mathcal{P}'\rangle\}$
      end if
      $\Sigma' \leftarrow \Sigma' \cup \{\langle D, \mathcal{P}_\sigma \wedge \mathcal{P}'\rangle\}$
      $\mathcal{P}' \leftarrow \mathcal{P}' \wedge \neg\mathcal{P}_\sigma$
    end if
  end for
  if $\mathcal{P}'$ is satisfiable **then**
    $\Sigma' \leftarrow \Sigma' \cup \{\langle D, \mathcal{P}'\rangle\}$
  end if
  $\Sigma \leftarrow \Sigma'$
end function

Algorithm 2 Approximation of equivalence queries

function RUN_TEST_SUITE(M, $T \subset \Sigma_M^*$)
  for all $w \in T$ **do**
    $\langle w \in L, Cont_A(w)\rangle \leftarrow membership(w)$
    if $w \in L$ **then**
      if $Cont_A(w) \setminus \Sigma_M \neq \emptyset$ **then**
        return $\langle False, w, Cont_A(w) \setminus \Sigma_M\rangle$
      end if
      if $w \notin L(M)$ **then**
        return $\langle False, w\rangle$
      end if
    else if $w \in L(M)$ **then**
      return $\langle False, w\rangle$
    end if
  end for
  return $True$
end function

If during the above procedure message types are removed from Σ then the L* algorithm must be restarted with the updated Σ since the learning was done with inaccurate alphabet. If message types are only added to Σ (and not removed) then we say that Σ grows. In the latter case the method from the exact algorithm (Section V-3) is used without having to restart L*.

C. Equivalence Oracle

Answering equivalence queries in real world for black box systems is generally infeasible [2]. Therefore, we define here an oracle to approximate equivalence queries. We take advantage of a commonly used approach in which an equivalence query is approximated using a sampling oracle. We use the Wp-Method [11] to generate a test suite $T \subset \Sigma_M^*$ of queries w. In this method, T is generated using M and the alphabet Σ_M.

The procedure to run a test suite T against a conjectured DFA M is shown in Algorithm 2. Each $w \in T$ is tested using a membership query. If $Cont_A(w)$ contains symbols that are not in Σ_M, then $False$ is returned with w and $Miss_A(M) = Cont_A(w)$. If w is in the symmetric difference of L and $L(M)$, then $False$ is returned with w as a counterexample. If missing message types are not found and a counterexample w is not found in the entire test suite T, $True$ is returned and the learning terminates. Recall that, if $Miss_A(M)$ is returned, then every $w \cdot \sigma$, such that $\sigma \in Miss_A(M)$, is handled as a counterexample.

D. Modified Symbolic Execution

Our algorithm answers membership queries by symbolic execution of the binary code. During the symbolic execution we need to find feasible executions in which the binary code follows a given sequence of alphabet symbols. We call this process *monitoring* the sequence. To monitor a sequence, we introduce assertions and assumptions into symbolic execution. In order to link between a sequence of alphabet symbols and the binary code, we propose modifications, which we apply on top of classical symbolic execution. Some other modifications are required in order to find $Cont_A(w)$ during a phase of the symbolic execution called *probing*.

1) Assumptions and assertions in symbolic execution: We simulate assumptions and assertions using symbolic execution constraints and using the fact that symbolic execution checks the satisfiability of states during the execution and discards unsatisfiable executions. We insert assumptions to the constraints of a symbolic state when we want to restrict the considered executions to executions that satisfy the assumptions. Conflicting constraints or concrete values will cause the execution to become unsatisfiable and discarded. Similarly, we insert assertions to the constraints of a symbolic state when we want to restrict the considered executions to executions where both the constraints developed by the symbolic execution and all of the assertions are satisfiable.

2) Hooking calls to send and receive functions: Hooking calls in symbolic execution allows to replace the behaviour of a binary code in certain functions with customized procedures. When hooks are applied and a state s represents a call to a hooked function, the customized procedure is executed on s instead of the original function. The customized procedure should return a state that represents the program after the call to the function.

When we apply symbolic execution on the binary, we hook functions that send or receive messages in order to monitor a sequence and to discover message type candidates. These functions are replaced with customized procedures which we describe in VI-E. For simplicity of exposition, we assume there is a single send function and a single receive function. However, the presented method can be equally implemented in case there are several such functions.

We ask the user of our method to manually identify functions that send and receive messages. The user should also specify for each function whether it is a send function

or a receive function. Identifying these functions is most of the times simple and require a small manual effort by the user of our method. Send and receive functions share a common property that they call the send or receive system calls. Therefore, these functions are easy to find with classical reverse engineering and disassemble tools. In case the user cannot find these functions, he can choose to provide the send and receive functions of the operating system.

When a send or a receive function is called within a state s and our customized procedures are executed, it is necessary to extract the message from s. Therefore, in addition to identifying the functions to hook, the user should provide a code snippet that extracts the message from the symbolic state s that represents the call to these function. The code snippet that the user provides will most likely get the message from the parameters of the call. Note that inferring this code snippet manually is considered trivial.

3) Extending symbolic states to track query state: Let $w = \sigma_1 \dots \sigma_n$ be a sequence of alphabet symbols queried in a membership query. Let s be a symbolic state. Recall the definition of symbolic state from Section III-B. We extend the definition of s to contain additional properties. We use these properties to track which alphabet symbols from the sequence have been sent or received in the execution that s represents, from the initial state and up to s. In addition, we add to s properties that will be used during the probing phase of the query. The properties we add are propagated from every state to its descendant states. Properties are set and updated explicitly in certain steps of our algorithm as we will describe in Section VI-E.

a) Properties for monitoring: We extend s to store in addition to the original properties:

- A sequence $w_s = \sigma_1 \dots \sigma_n$. The sequence, and especially its predicates, are used to insert assumptions and assertions during the symbolic execution, to limit the considered executions to executions that match the query.
- An index i_s such that $0 \leq i_s \leq n$. This index marks the index of the last symbol monitored in the execution when reaching s. That is, the sequence $\sigma_1 \dots \sigma_{i_s}$ was sent or received in the execution up to s. i_s is incremented by one every time a send or receive that match σ_{i_s+1} occurs in the execution. If $i_s = n$, the entire sequence w_s was monitored in the execution represented by s.

When the symbolic execution is initialized to answer a membership query w, the initial state s is defined with $w_s = w$ and $i_s = 0$.

b) Properties for probing: When a state s is executed during the probing phase, we use two additional properties that we add to s:

- A symbolic value msg_s. msg_s is initialized during the first send or receive call that occurs in the probing phase. It is initialized with the symbolic value sent or received. msg_s will be used to generate a message type candidate for s as we explain in Section VI-E2c.
- A flag $D_s \in \{R, S, \emptyset\}$, which specifies whether the first communication in the probing phase was send or receive. It is initialized with $D_s = \emptyset$ and set only during the first send or receive in the probing phase.

E. Membership Oracle

Let $w \in \Sigma^*$ be a sequence of message types sent as a membership query. The oracle should answer whether $w \in L$ and if $w \in L$ it should also provide $Cont_A(w)$ – a set of message type candidates that may follow w. By definition, w is a sequence of message types. Recall that such a sequence corresponds to sessions of the protocol. We answer membership queries using symbolic execution of the binary code.

A symbolic execution begins with a single active initial state, located at the binary's entry point. By stepping forward active states iteratively, a set of new active states is generated, representing multiple different execution paths of the binary. Recall that we divide the symbolic execution into two phases: monitoring phase, which answers whether w is a valid session of the protocol, and the probing phase, which results in possible continuations of w. The latter phase is executed only if w is a valid session. During the monitoring phase we guide the symbolic execution to consider only execution paths that follow the given sequence w. During the probing phase, however, we take into account all feasible executions that are developed as continuations to the executions that we found during the monitoring phase.

1) Monitoring phase: Let $w = \langle D_{\sigma_1}, \mathcal{P}_{\sigma_1} \rangle \dots \langle D_{\sigma_n}, \mathcal{P}_{\sigma_n} \rangle$ be the queried sequence. We hook the functions in the binary that send and receive messages as we described in Section VI-D. The procedures inserted in the hooks are presented in the following subsections. We perform the monitoring phase in n stages: We start with a single initial state s_e located at the binary code's entry point with $i_{s_e} = 0$ and $w_{s_e} = w$. For each stage $1 \leq i \leq n$ we add constraints of the predicate of the message type $\langle D_{\sigma_i}, \mathcal{P}_{\sigma_i} \rangle$. These constraints restrict the symbolic execution to execution paths that send or receive a message of type $\langle D_{\sigma_i}, \mathcal{P}_{\sigma_i} \rangle$ at the current stage. We then resume the execution of all active states, until all active states are restricted to the i-th message type. In other words, we begin a stage i with states that have $i_s = i - 1$ and finish it with a set of new states with $i_s = i$. In the next subsections we explain in detail how we eliminate execution paths that do not match w. Recall that states with unsatisfiable constraints as well as abort states, are discarded automatically.

If we successfully finish the execution of the last stage (for $i = n$) with at least one active state, then there is at least one valid session of the binary code that matches the sequence of message types of w. In such a case the answer to the query is $True$. If, however, during one of the stages there are no active states left, then w represents invalid session for the binary code, and therefore the oracle returns $False$ as the query's result.

In Figure 3, the monitoring phase is illustrated in the left part of the figure. States that represent infeasible executions (infeasible constraints) are discarded (gray). States that represent feasible executions that do not match the query w are discarded as well (red). The stages of the monitoring are

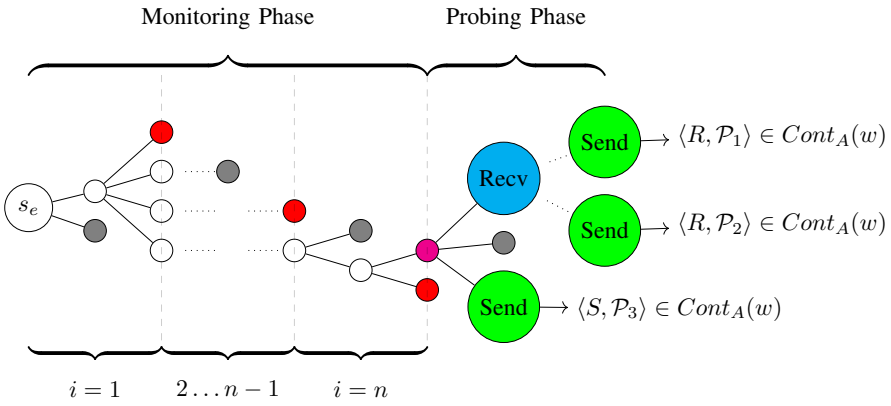

Fig. 3. Illustration of symbolic execution during membership query

illustrated as well. The figure represents a membership query that is answered with $True$, as a single active state (magenta) is found at the end of the monitoring phase.

a) Monitoring incoming messages: The receive function is hooked during the monitoring phase with the following procedure. The purpose of this procedure is to advance the monitoring of the queried sequence when the binary code receives a message. Let s be a symbolic state during the i-th stage in which the binary code calls the receive function, and let $\sigma_i = \langle D_{\sigma_i}, \mathcal{P}_{\sigma_i} \rangle$ where $i = i_s + 1$ be the next expected alphabet symbol in the query.

If σ_i represents outgoing messages (i.e $D_{\sigma_i} = S$), then s is not an execution path that can match the queried sequence w: the execution path represented by s receives a message in the i-th stage whereas the query w represents sessions that send a message from $\mathcal{M}(S, \mathcal{P}_{\sigma_i})$ in the i-th stage. Therefore we discard s.

On the other hand, if σ_i represents incoming message type (i.e $D_{\sigma_i} = R$), we move s to the $i+1$-th stage with s' as its successor and set $i_{s'} = i = i_s + 1$. We attach to s' an assumption that a message from $\mathcal{M}(R, \mathcal{P}_{\sigma_i})$ is read from the network. We implement this assumption by inserting the predicate $\mathcal{P}_{\sigma_i}(msg)$ to the constraints of s', where msg is the received message buffer. We acquire msg using the code snippet the user provides in order to extract it from s (See VI-D2).

Once s' resumes execution, it will continue as if the received message satisfies $\mathcal{P}_{\sigma_i}$ and thus "forcing" descendant states to follow only execution paths that represent the reception of messages from $\mathcal{M}(R, \mathcal{P}_{\sigma_i})$ during the i-th stage.

The pseudo-code in Algorithm 3 for $D = R$ describes the monitoring process that occurs when the receive function is called from a state s in order to receive a message msg.

b) Monitoring outgoing messages: The procedure to hook a send function is similar to the one used above for incoming messages. The purpose of the procedure is to perform the monitoring of the queried sequence when the binary code sends a message. Let s be a symbolic state in which the binary code calls the send function, and let $\sigma_i = \langle D_{\sigma_i}, \mathcal{P}_{\sigma_i} \rangle$ where $i = i_s + 1$ be the next expected alphabet in the query.

Algorithm 3 The monitor procedure

```
1:  function MONITOR(s, D ∈ {R, S})
2:      ⟨D_σi, P_σi⟩ = w_s[i_s + 1]
3:      msg ← user_code_snippet(s)
4:      if D_σi = D then
5:          s' ← s.copy()
6:          if D = R then
7:              s'.add_assumption(P_σi(msg))
8:          else if D = S then
9:              s'.add_assertion(P_σi(msg))
10:         end if
11:         i_s' ← i_s + 1
12:         return s'
13:     else
14:         return null
15:     end if
16: end function
```

In case σ_i represents incoming messages (i.e $D_{\sigma_i} = R$), it means that the execution path of s does not match the sequence query w and we discard s. On the other hand, if σ_i represents an outgoing message type (i.e $D_{\sigma_i} = S$), we move s to the $i+1$-th stage with s' as its successor and set $i_{s'} = i = i_s + 1$. We attach to s' an assertion that a message from $\mathcal{M}(S, \mathcal{P}_{\sigma_i})$ is sent to the network. We implement this assertion by inserting the predicate $\mathcal{P}_{\sigma_i}(msg)$ to the constraints of s', where msg is the sent message buffer. The procedure is presented in Algorithm 3 for $D = S$.

2) Probing phase: The purpose of the probing phase is to generate $Cont_A(w)$ for w for which the monitoring phase returned $True$. As above, we hook the send and receive functions of the binary code but insert different procedures. We describe them in the upcoming subsections. The aim of the probing procedure is to uncover all symbolic states that represent execution paths in which a message is sent or received following w. For each such state s the constraints on the message buffer received or sent, denoted as msg_s, are collected.

Recall the definition of msg_s and D_s from Section VI-D3b. Given a state s in the probing phase, the purpose of these hooks is to store msg_s and D_s and trigger the generation of message type candidates. We assume that all concrete values for msg_s in the context of state s belong to the same message type and generate a message type candidate to represent it.

In Figure 3, the probing phase is shown in the right part of the figure. Since the purpose of the probing phase is to discover all message type candidates that can follow the sequence w, we continue the execution from active states matching the query w at the end of the monitoring phase (magenta). That is, states with $i_s = n$. A message type candidate c is generated from every state s (green), and added to $Cont_A(w)$. We demonstrate the probing of message type candidates in Appendix A.

a) Probing outgoing messages: The hooking procedure used in the send function is straightforward. Here msg_s is the sent message's symbolic buffer. We assume that the symbolic buffer has enough constraints under the current state s that sufficiently represent the sent message type. Therefore, no further symbolic execution is needed. We set $D_s = S$ and $msg_s = msg$, where msg is the sent message. The state s is then passed to the procedure to generate a message type candidate.

b) Probing incoming messages: Let s be a state in which the binary code calls a receive function during the probing phase. Let msg be the symbolic received message. Upon calling receive, the content of msg is an unconstrained symbolic value as it is received as an input to the binary code. Hence, one cannot extract information on the format of the message type that is expected to be received in state s. To solve this, we present the following novel approach to uncover information regarding the expected received message type: we clone s to s' with $msg_{s'} = msg$ and $D_{s'} = R$. Then we resume symbolic execution of s'. During this execution we assume that the binary code will parse the received message, hence constraints will be developed on $msg_{s'}$ that will reveal the format expected by the binary code. We choose to resume the execution until the binary code sends or receives another message, or until the code terminates. We assume that until that point the code completes parsing the received message and acts upon its content, hence sufficient constraints are accumulated on the message buffer to identify the expected message type to be received. During these instructions s' is developed into possibly multiple descendant states.

Let $\widetilde{s}$ be a descendant state of s' in which either send or receive function are called for the first time after s'. Alternatively, $\widetilde{s}$ is a descendant state of s' in which the binary code terminates. $\widetilde{s}$ represents a point of the binary code in which the incoming message is already analyzed and its type is determined. Therefore it has passed through conditional branches that append constraints on $msg_{s'}$. These constraints will give us information on the structure of the message. $\widetilde{s}$ is then passed to the procedure to generate message type candidate. When $\widetilde{s}$ has successfully generated a message type candidate, it is removed from the set of active states to discontinue the execution path represented by $\widetilde{s}$. This execution path has already produced a message type that follows the queried sequence and there is no need to resume its execution.

c) Generating message type candidates: Let s be a state that successfully probed either sent or received message – msg_s. The purpose of the procedure described here is to generate a message type candidate from s. Note that concrete values satisfying the constraints of s on msg_s represent valid messages in the protocol. We assume that these concrete messages form a message type t. We ask the symbolic execution engine to solve msg_s and generate NUM_SOL[1] possible concrete values for msg_s.

Let x be the set of generated concrete messages. We extract $\mathcal{P}_x$ as described in Section VI-A. Then, we iteratively refine $\mathcal{P}_x$ by trying to find concrete values m' of msg_s, that contradict $\mathcal{P}_x$ in a sense that $\neg\mathcal{P}_x(m')$ is $True$. Such m' are concrete values that can appear in msg_s in a real execution (since they were solution to the msg_s). Nevertheless, they are not represented by $\mathcal{P}_x$. In case we find such m', we add them to x and regenerate $\mathcal{P}_x$. We repeat this process until the solver is unable to find additional m' that contradict $\mathcal{P}_x$. This procedure allows us to find a set $x \subset t$ that represents the variety of t.

Recall our statement from Section VI-A regarding the diversity of the set x when generating $\mathcal{P}_x$. The procedure described here uses a smaller, diverse enough set x to describe a larger set $\mathcal{M}(D_s, \mathcal{P}_x)$. $\mathcal{M}(D_s, \mathcal{P}_x)$ approximates the set of all concrete values of msg_s.

Let $concretize(s, buffer, n, C)$ be a function of the symbolic execution engine that returns up to n concrete values for $buffer$ that corresponds to the constraints of s and to additional constraints C. In case there are no possible values that follow the constraints, the function returns $\emptyset$. The pseudo-code of the message generation process is shown in Algorithm 4.

Algorithm 4 Generation of message type candidate

```
1: function GENERATE_ALPHABET_CANDIDATE(s)
2:     x ← ∅
3:     contrads ← ∅
4:     do
5:         P_x ← extract_predicate(x)
6:         contrads ← concretize(s, msg_s, NUM_SOL, ¬P_x)
7:         x ← x ∪ contrads
8:     while contrads ≠ ∅
9:     Cont_A(w) ← Cont_A(w) ∪ {⟨D_s, P_x⟩}
10: end function
```

VII. Optimizations

We develop several optimizations to reduce the running time of the method and allow it to scale to real-world protocol implementations. These optimizations take advantage of the characteristics of network protocols and the algorithm itself. Since symbolic execution is the most time consuming part of

[1]In our implementation $NUM\_SOL = 10$

the algorithm, the developed optimizations focus on reducing the number of needed symbolic executions, as well as reducing the running time of symbolic executions.

1) Prefix Closed Property: This optimization leverages the fact that the protocol's regular language L is a prefix-closed set (See Section IV). It is based on a similar technique, employed in [13]. The optimization allows us to answer some membership queries immediately by the Learner without having to resort to symbolic execution. Every membership query w that was answered with $False$ is stored by the membership oracle in a cache. For every membership query w sent to the oracle, it is first checked whether there exists $x, y \in \Sigma^*$ such that $xy = w$ and x is in the cache. In such a case, the query immediately returns $False$. In other words, if a prefix of w is not in L, then by definition of prefix-closed set it must hold that $w \notin L$. Thus, we avoid unnecessary applications of symbolic execution.

When, during the discovery of new message types, an alphabet symbol is removed and L* is restarted, all queries w in the cache that contain a removed symbol are removed from the cache. These queries are invalid with the new alphabet and cannot be a prefix of a query over the new alphabet.

2) Fast Equivalence Queries: Let $w \in \Sigma^*$ be a query for which the membership oracle answered $True$, and let $Cont_A(w)$ be the returned set of alphabet candidates. We store w and its associated $Cont_A(w)$ in a cache called *continuations cache*. The equivalence oracle answers an equivalence query for DFA M by utilizing this cache. The oracle checks consistency of M with the continuations cache: for every w in the cache and for every $\sigma \in Cont_A(w)$, it checks whether M accepts $w \cdot \sigma$. If M rejects $w \cdot \sigma$, the equivalence oracle returns $False$ and returns $w \cdot \sigma$ as a counterexample. Thus it alleviates the need to run symbolic execution to answer the query. Note that, the cache stores alphabet symbols after resolving collisions, and not message type candidates. This is necessary so that the cache can return counterexamples over the current alphabet. When alphabet symbol is removed, all cache entries containing the removed symbol are erased.

The correctness of this optimization follows from the definition of $Cont_A(w)$. According to the definition, every state machine M that claims $L(M) = L$ should satisfy $w \cdot \sigma \in L(M)$.

3) Execution Cache: This optimization uses symbolic states s resulting at the end of the monitoring phase for w as initial states for query wx for any $x \in \Sigma^*$. All queries w for which the teacher returns $True$ are stored in a cache called "Execution Cache" with all active symbolic states resulting at the end of the monitoring phase for w. Then, whenever a query w' is sent, the teacher finds decomposition $w' = p.s, p = p_1 \ \dots \ p_k$ such that p is the longest word in the cache. Then, the monitoring phase for w' begins with the states saved for p, in the $i = k+1$ stage of the monitoring. We skip the first k stages because the states saved for p contains exactly all execution paths for sessions $p_1 \dots p_k$. The rest of the query remains the same as described in Section VI-E. We note that, when alphabet symbols are removed, all entries in the cache that include the removed symbol should also be removed.

VIII. Implementation, Results and Evaluation

In this section we present the details of our implementation of the presented method and explore its performance. We evaluated our method against various protocol implementations (including SMTP and other non-standard protocols), however due to space constraints we present here only an evaluation against Gh0st RAT's C&C protocol.

A. Implementation

The algorithm was implemented[2] as two independent modules for the Learner and the Teacher. The Learner is implemented as a Java program that communicates with the Teacher using local socket. The Teacher is implemented as a Python program that serves the Learner's queries. We base our implementation on two open source tools: (1) LearnLib [13] – implements the L* algorithm and its variations (for example, [14]); (2) angr [15] – a library that provides static analysis and symbolic execution engine for binary codes.

1) Learning Client (Learner): The Learner begins by initializing a learning process with LearnLib's implementation of L*. Membership queries are first checked with the prefix-closed cache (See Section VII-1). In case of a miss, the query is sent to the Teacher. If the Teacher answers that $w \in L$, then $Cont_A(w)$ is analyzed for new message types which are added to Σ. Intersections between message types are handled as described in Section VI-B.

Conjecture DFA is checked first against the continuations cache as described in Section VII-2. If the conjectured DFA is found to accept all continuations in the cache, an equivalence query approximation is triggered. A test suite is generated using the Wp-Method [11] and is tested as explained in Section VI-C. Missing message type are handled as described in Section VI-B. Counterexamples are handled by the internal implementation of L*.

We use the following features of LearnLib: Classical L* implementation; Support for growing alphabet in L*; Test suite generation with Wp-Method; Prefix-closed cache (See Section VII-1). On the other hand, we implemented the following modifications: (1) Alphabet symbols as tuples $\langle D, \mathcal{P} \rangle$; (2) Handling of alphabet changes and collisions (Section VI-B); (3) Continuations cache to support Fast equivalence queries. (Section VII-2); (4) Running tests suites to approximate modified equivalence queries (Section VI-C).

2) Symbolic Execution Server (Teacher): Our Teacher runs symbolic execution using angr, and is the only component that interacts with the binary code. The Teacher initializes symbolic execution for the binary code and setups the hooking of the send/receive functions the user provides. The Teacher receives membership queries in a loop, until the Learner finishes the learning. When a membership query is received, we first check the execution cache optimization (See Section VII-3),

[2]https://github.com/ron4548/pise

in case of a miss the monitoring phase executes as described in Section VI-E. If the query results with $True$, the probing phase runs and generates message type candidates. These candidates are collected and sent back to the Learner.

B. Gh0st RAT

Gh0st RAT is a well known malware[3]. Once an instance of Gh0st RAT is run on the victim's computer, the attacker has full control over the system. This includes access to the screen, microphone and camera. The attacker controls the malware using a C&C protocol. The source code of some variants of Gh0st is available on the web. We chose to work with one of them[4]. In this variant, the RAT runs in a multi-threaded process which connects to the attacker's server. When a command is received, a new thread and a new connection are created to handle the command and its further communications.

Initially, we applied our method on this variant. However, angr [15] is not well-suited for multi-threaded programs. In addition, angr does not fully support Windows API. This lead to difficulties with applying our method on the Gh0st RAT binary directly. To validate that the proposed method can infer a state machine as complex as that of Gh0st RAT, we opted for a different approach. We re-implemented most of the malware's C&C protocol with a simpler architecture that does not involve threads. We applied our method on this program.

We provided our method with two functions that the program uses in order to send and receive messages from the network: `get_message` and `send_message`. Both get a message buffer and its length. The full state machine is complex and contains 27 states and 52 transitions (without rejecting states). We show the full state machine and the discovered alphabet symbols in Appendix B.

In Figure 4 we show a branch of the state machine, that handles a command to stream the camera of the victim. In this branch, the attacker sends a command to open the camera stream (`[R] WEBCAM`). Then, the client sends information regarding the stream (`[S] BMPINFO`) and waits to receive from the attacker a command to begin streaming (`[R] NEXT`). From now on, the client sends periodically a bitmap of the webcam to the attacker's server (`[S] BMP`). By default, this bitmap is not compressed. The attacker can enable compression of the stream (`[R] COMPR_ON`) and disable it (`[R] COMPR_OFF`). When the compression is on, the bitmap is sent compressed (`COMPR_BMP`).

[3]https://attack.mitre.org/software/S0032/

[4]https://github.com/yuanyuanxiang/SimpleRemoter

Learning time:	142 seconds
Total Membership queries:	45488
Total Equivalence queries:	1
Prefix-Closed cache miss rate:	0.2184
Alphabet size:	45

TABLE I
GH0ST RAT LEARNING STATISTICS

Statistics of the learning process are shown in Table I. 45 message types were discovered. The learner issued about 45,000 membership queries; more than 78% of them were answered by the prefix-closed cache. Only a single equivalence query was issued. This shows the dramatic effectiveness of the continuations cache to reduce the number of costly equivalence queries. There are no discrepancies between the learnt DFA and the protocol's state machine.

IX. CONCLUSIONS

In this work we present a novel method for inferring the state machine of a protocol implemented by a binary with no a-priori knowledge of the protocol. Our method is based on extended symbolic execution and modified automata learning. The method assumes access to only the implementation of a single peer of the protocol.

We implemented and validated our method on several protocols implementations. As demonstrated by the Gh0st RAT use case, the method can infer complex protocols with dozens of message types. Nonetheless, this use case also highlighted that the method will perform as a good as the symbolic execution engine it relies on.

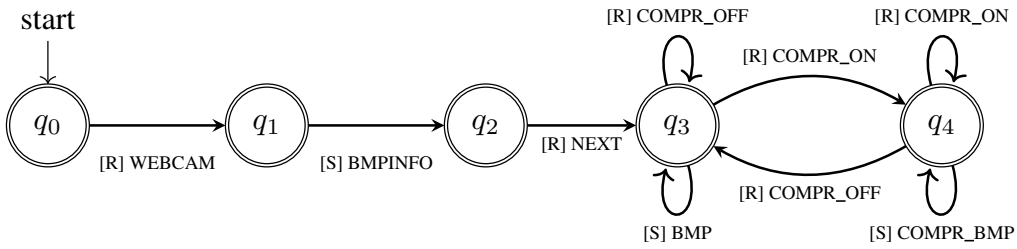

Fig. 4. The branch in Gh0st RAT C&C protocol that handles webcam streaming. The letter in the square brackets indicates whether the message is sent or received.

REFERENCES

[1] R. Alur, P. Cerný, P. Madhusudan, and W. Nam, "Synthesis of interface specifications for java classes," in *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, 2005, pp. 98–109.

[2] D. Angluin, "Learning regular sets from queries and counterexamples," *Inf. Comput.*, vol. 75, no. 2, pp. 87–106, 1987.

[3] J. Caballero and D. Song, "Automatic protocol reverse-engineering: Message format extraction and field semantics inference," *Computer Networks*, vol. 57, no. 2, pp. 451–474, 2013.

[4] J. Caballero, H. Yin, Z. Liang, and D. X. Song, "Polyglot: automatic extraction of protocol message format using dynamic binary analysis," in *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*, 2007, pp. 317–329.

[5] C. Y. Cho, D. Babić, P. Poosankam, K. Z. Chen, E. X. Wu, and D. Song, "MACE: Model-inference-Assisted Concolic Exploration for Protocol and Vulnerability Discovery," in *Proceedings of the 20th USENIX Security Symposium*, 8 2011.

[6] C. Y. Cho, D. Babic, E. C. R. Shin, and D. Song, "Inference and analysis of formal models of botnet command and control protocols," in *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*, 2010, pp. 426–439.

[7] P. M. Comparetti, G. Wondracek, C. Krügel, and E. Kirda, "Prospex: Protocol specification extraction," in *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA*, 2009, pp. 110–125.

[8] W. Cui, J. Kannan, and H. J. Wang, "Discoverer: Automatic protocol reverse engineering from network traces," in *Proceedings of the 16th USENIX Security Symposium, Boston, MA, USA, August 6-10, 2007*, 2007.

[9] W. Cui, M. Peinado, K. Chen, H. J. Wang, and L. Irún-Briz, "Tupni: automatic reverse engineering of input formats," in *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008*, 2008, pp. 391–402.

[10] M. Emmi, D. Giannakopoulou, and C. S. Pasareanu, "Assume-guarantee verification for interface automata," in *FM 2008: Formal Methods, 15th International Symposium on Formal Methods, Turku, Finland, May 26-30, 2008, Proceedings*, 2008, pp. 116–131.

[11] S. Fujiwara, G. von Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi, "Test selection based on finite state models," *IEEE Trans. Software Eng.*, vol. 17, no. 6, pp. 591–603, 1991.

[12] M. Gheorghiu, D. Giannakopoulou, and C. S. Pasareanu, "Refining interface alphabets for compositional verification," in *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings*, 2007, pp. 292–307.

[13] M. Isberner, F. Howar, and B. Steffen, "The open-source learnlib," in *Computer Aided Verification*, D. Kroening and C. S. Păsăreanu, Eds. Cham: Springer International Publishing, 2015, pp. 487–495.

[14] M. Shahbaz and R. Groz, "Inferring mealy machines," in *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings*, 2009, pp. 207–222.

[15] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Krügel, and G. Vigna, "SOK: (state of) the art of war: Offensive techniques in binary analysis," in *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*. IEEE Computer Society, 2016, pp. 138–157.

[16] M. Weiser, "Program slicing," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 4, pp. 352–357, 1984.

[17] G. Wondracek, P. M. Comparetti, C. Krügel, and E. Kirda, "Automatic network protocol analysis," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*, 2008.

APPENDIX

A. Example of probing message type candidates

To illustrate how a conditional branch reveals information on a symbolic value, consider the pseudo-code in Listing 1 of a binary code:

Listing 1. Pseudo-code to demonstrate the probing phase

```
1:  Send(Connect);
2:  msg = Receive();
3:  if (msg == "HelloV1") {
4:      Send("InitV1");
5:      ...
6:  } else if (msg == "HelloV2") {
7:      Send("InitV2");
8:      ...
9:  } else {
10:     abort();
11: }
```

Assume a query $w = \langle S, \mathcal{P} \rangle$ where $\mathcal{P} = (B_0B_1B_2B_3B_4B_5B_6 = \text{"Connect"})$. The monitoring phase for this query and this binary code is done with a single satisfiable state in line 2. The probing phase resumes this state. In line 2 the binary code receives msg in a state s. msg refers to a symbolic value with no constraints, as it is an input from the network. A state $s' = s$ is resumed with $msg_{s'} = msg$. The execution splits according to the conditional branches: a state $\widetilde{s_1}$ represents execution at line 4 with a constraint that $msg =$ "HelloV1", a state $\widetilde{s_2}$ represents execution at line 7 with a constraint that $msg =$ "HelloV2" and a state s_3 represents execution at line 10 which aborts and is discarded. Both $\widetilde{s_1}$ and $\widetilde{s_2}$ represent a call to send, which triggers the generation of message type candidate from $msg_{\widetilde{s_1}} = msg_{\widetilde{s_2}} = msg$. In the context of $\widetilde{s_1}$, the analysis is tied to the constraint $msg =$ "HelloV1" and generates a message type candidate $\langle R, \mathcal{P}_1 \rangle$ where:

$$\mathcal{P}_1 = (B_0B_1B_2B_3B_4B_5B_6 = \text{"HelloV1"})$$

In the context of $\widetilde{s_2}$ the analysis is tied to the constraint $msg =$ "HelloV2" and generates an alphabet symbol $\langle R, \mathcal{P}_2 \rangle$ where:

$$\mathcal{P}_2 = (B_0B_1B_2B_3B_4B_5B_6 = \text{"HelloV2"})$$

The set $Cont_A(w) = \{\langle R, \mathcal{P}_1 \rangle, \langle R, \mathcal{P}_2 \rangle\}$ is returned with the answer that $w \in L$.

B. Gh0st RAT Inference Results

The full state machine learnt by applying our method on Gh0st RAT C&C is presented in Figure 5. In the protocol a message type is determined by the first byte of the message and some message types provide additional information in the second byte. In Table II we present the predicate of the type as the prefix common to all the messages in that type.

MSG ID	Name	Prefix	MSG ID	Name	Prefix
[R:0]	SERVER_EXIT	0xcd	[R:1]	CMD_BYE	0xcc
[R:2]	CMD_TALK	0x34	[R:3]	CMD_REGEDIT	0x33
[R:4]	CMD_AUDIO	0x22	[R:5]	CMD_SHELL	0x28
[R:6]	CMD_SERVICES	0x32	[R:7]	CMD_SCREEN_SPY	0x10
[R:8]	CMD_CAM	0x1a	[R:145]	CMD_SCREEN_BLOCK_INPUT	0x15
[R:10]	CMD_SYSTEM	0x23	[S:277]	TOKEN_CLIPBOARD_TEXT	0x76
[S:12]	TOKEN_BITMAPINFO	0x73	[S:13]	TOKEN_AUDIO_START	0x79
[S:14]	TOKEN_SERVERLIST	0x81	[R:140]	CMD_SCREEN_SET_CLIPBOARD	0x19
[S:16]	TOKEN_WSLIST	0x7e	[S:17]	TOKEN_TALK_START	0x84
[S:19]	TOKEN_SHELL_START	0x80	[S:20]	TOKEN_CAM_BITMAPINFO	0x77
[S:21]	CMD_BYE	0xcc	[R:32]	CMD_SVCCFG/START	0x83 0x01
[R:24]	CMD_NEXT	0x1e	[R:30]	CMD_SVCCFG/DEMAND_START	0x83 0x04
[R:29]	CMD_SERVICELIST	0x82	[R:31]	CMD_SVCCFG/AUTO	0x83 0x03
[S:22]	SERVER_EXIT	0xcd	[R:33]	CMD_SVCCFG/STOP	0x83 0x02
[R:34]	CMD_REG_FIND	0xc9	[R:36]	CMD_WINDOW_CLOSE	0x00
[R:37]	CMD_PSLIST	0x24	[S:67]	TOKEN_FIRSTSCREEN	0x74
[S:68]	TOKEN_AUDIO_DATA	0x7a	[S:74]	TOKEN_CAM_DIB	0x78 0x00
[S:73]	TOKEN_TALKCMPLT	0x85	[R:75]	CMD_CAM_ENABLECOMPRESS	0x1b
[S:112]	TOKEN_PSLIST	0x7d	[R:76]	CMD_CAM_DISABLECOMPRESS	0x1c
[S:137]	TOKEN_NEXTSCREEN	0x75	[R:138]	CMD_SCREEN_GET_CLIPBOARD	0x18
[S:15]	TOKEN_REGEDIT	0xc8	[R:144]	CMD_SCREEN_CONTROL	0x14
[R:9]	CMD_WSLIST	0x25	[S:199]	TOKEN_CAM_DIB/COMPRESS	0x78 0x01
[R:11]	CMD_LIST_DRIVE	0x01			

TABLE II
LEARNT MESSAGE TYPES

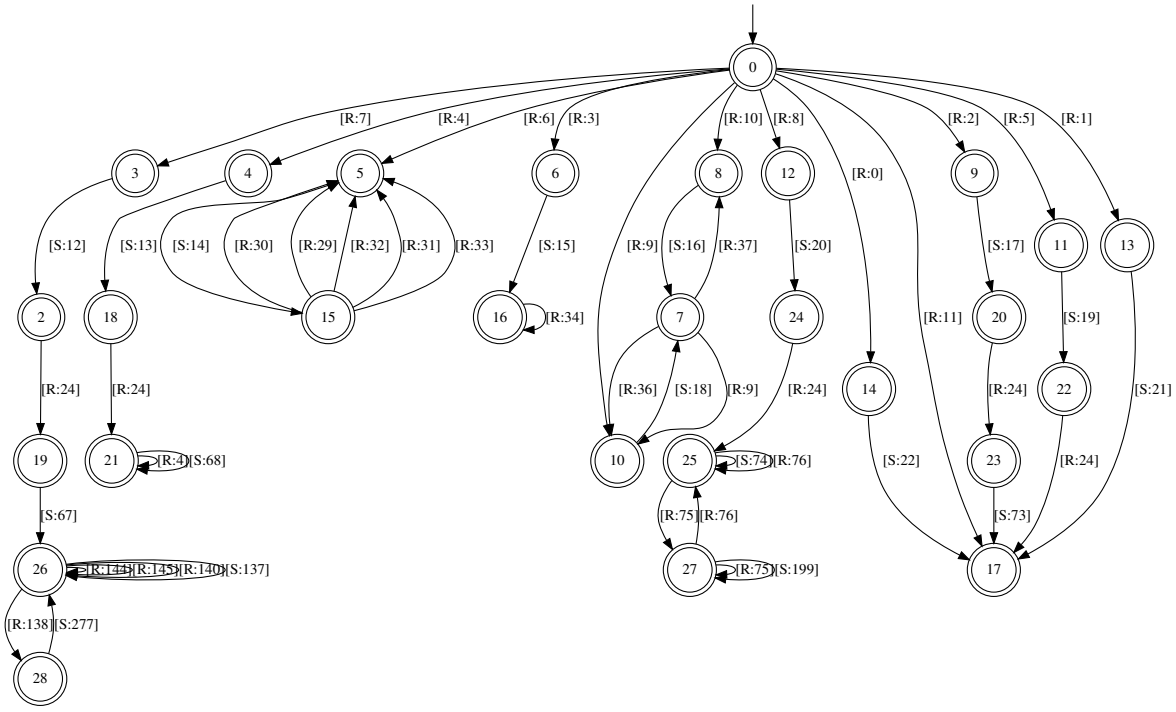

Fig. 5. Full state machine learnt by our method

C. End-to-end example

To illustrate the problem and our proposed method we utilize an example of inferring the state machine of a simple imaginary protocol.

The pseaudo-code in Listing 2 describes a client that implements this protocol. The client first retrieves the user id, and sends an `Hello` message along with the specified id. For simplicity, we assume that during symbolic execution, the method `GetIdFromUser()` returns an unconstrained symbolic value. Afterwards, the client expects to receive a `Welcome` message with the specified id. Then, the client repeatedly receives `Data` messages until an `End` message is received, in which case the client terminates successfully. If the client receives a message which is not expected, it terminates with the value of -1.

Listing 2. Pseudo-code of a client to infer

```
1:  ID = GetIdFromUser();
2:  Send("Hello" + ID);
3:  msg = Receive();
4:  if (msg != "Welcome" + ID) { Exit(-1); }
5:  while (True) {
6:      msg = Receive();
7:      if (msg == 'End') { break; }
8:      if (!msg.BeginsWith('Data')) {
9:          Exit(-1);
10:     }
11:     SaveData(data);
12: }
13: Exit(0);
```

When our method is applied to this client, we hook the `Send()` and `Receive()` methods. The process begins with $\Sigma = \emptyset$, thus the first query to be issued by the learner is $w_0 = \epsilon$. This query is answered with True since an empty session is valid for every protocol, as found by the monitoring phase (Section VI-E1). During the probing phase of w_0, we get examples of concrete messages that may follow w_0, e.g., `Hello 34`, `Hello 213` and `Hello 9102` (Section VI-E2), and find that the predicate for them is $(B_0B_1B_2B_3B_4 = \text{"Hello"})$ (Section VI-E2c). Therefore, the message type candidate is $\langle S, \text{"Hello"} \rangle$. Since no message types in Σ collide with this message type candidate, $\langle S, \text{"Hello"} \rangle$ is simply added to Σ (Section VI-B). Σ grew, therefore the observation table needs to be updated (Section V-3), thus the query $w_1 = \langle S, \text{"Hello"} \rangle$ is sent by the learner. w_1 is answered with True. The consequent probing phase reveals that a received message follows w_1 (line 3), therefore probing continues to the next message (`Receive()` at line 6) in order to collect constraints for the first received message (from line 3). In this case constraints are introduced based on the conditional branch in line 4. Message examples that answer those constraints are then generated, and message type candidate $\langle R, \text{"Welcome"} \rangle$ with predicate $(B_0B_1B_2B_3B_4B_5B_6 = \text{"Welcome"})$ is uncovered. $\langle R, \text{"Welcome"} \rangle$ is added to Σ.

According to the answer for w_1, the learner issues queries $w_2 = \langle S, \text{"Hello"} \rangle \cdot \langle S, \text{"Hello"} \rangle$ and $w_3 = \langle R, \text{"Welcome"} \rangle$ which are answered with False, and $w_4 = \langle S, \text{"Hello"} \rangle \cdot \langle R, \text{"Welcome"} \rangle$ which is answered with True and uncovers during the probing phase two message type candidates that are added to Σ: $\langle R, \text{"End"} \rangle$ with predicate $(B_0B_1B_2 = \text{"End"})$ and $\langle R, \text{"Data"} \rangle$ with predicate $(B_0B_1B_2B_3 = \text{"Data"})$.

A series of additional queries is issued by the original L* algorithm [2] to update the observation table such that the learner has a conjecture DFA to send in an equivalence query. The first conjectured DFA is shown in Figure 6. For simplicity, the rejecting state and its incoming transitions are omitted. Equivalence query generates a test suite to

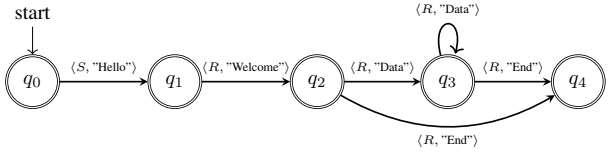

Fig. 6. The first conjectured DFA in the learning.

test this DFA against and validate the DFA's answer with a membership query (Section VI-C). For this conjecture, a test suite does not reveal any counterexamples or missing message types, and therefore the learning process terminates with this DFA and the set of message types: $\Sigma = \{\langle S, \text{"Hello"} \rangle, \langle R, \text{"Welcome"} \rangle, \langle R, \text{"Data"} \rangle, \langle R, \text{"End"} \rangle\}$

D. Scalabillity extensions

The core of our method is scalable as its complexity is determined by the complexity of the L* algorithm which is polynomial in the number of states and the maximum length of any counterexample provided by equivalence query [2]. Nonetheless, our method performs as good as the symbolic execution engine it uses. This may inhibit the method from being applied to binary programs for which the symbolic execution engine is not applicable.

One way to alleviate this limitation is replacing symbolic execution with dynamic symbolic execution, which combines symbolic execution and concrete execution. This way, modeled program may skip symbolic execution of complex parts by executing them concretely instead, avoiding symbolic state explosion on one hand but exploring all execution paths of the program on the other hand. Another way is to reduce the program using program slicing [16]. With program slicing, program instructions that do not affect the network interaction of the program can be omitted to create a simple, reduced program, that is simpler for symbolic execution. We argue that such technique can relieve the difficulty complex library functions pose.

Our method is also scalable in terms of its target protocols. Even through our method was developed and was demonstrated to work with classical network application protocols, its core ideas are abstract, and as such can be deployed on protocols of lower levels and different mediums, such as USB or Bluetooth, as long as they fit our minimal assumptions.

With that being said, if the user of our method chooses to terminate the algorithm before L* finishes, the state machine developed up until the termination under-approximates the state machine of the protocol. Message types discovered until that point describe valid messages of the protocol.