Automatic Transition System Model Identifications for Network Applications from Packet Traces

Zeynab Sabahi-Kaviani, Fatemeh Ghassemi, and Fateme Bajelan

School of Electrical and Computer Engineering,
University of Tehran, Tehran, Iran
z.sabahi@ut.ac.ir, fghassemi@ut.ac.ir, bajelan.fateme@ut.ac.ir

Abstract. A wide range of network management tasks such as balancing bandwidth usage, firewalling, anomaly detection and differentiating traffic pricing, depend on accurate traffic classification. Due to the diversity and variability of network applications, port-based and statistical signature detection approaches become inefficient and hence, behavioral classification approaches have been considered recently. However, so far, there is no automated general method to obtain the behavioral models of applications. In this research, we propose an automatic procedure to infer a transition system model from generated traffic of an application. Our approach is based on passive automata learning theory and evidence driven state merging technique using the rules of the network domain. We consider the behavior of well-known network protocols to generate the model which includes unobserved behaviors and excludes invalid ones as much as possible. To this aim, we present a new equivalence relation regarding the given protocol behaviors to induce proper state merging conditions. This idea has led the time complexity order of the algorithm to be linear rather than exponential. Finally, we apply the model of some real applications to evaluate the precision and execution time of our approach.

1 Introduction

The importance of traffic classification for network administration tasks such as ensuring the security and quality of service of applications in computer networks has long been acknowledged. The growing number of network applications and protocols has limited the efficiency of classical methods. In the past, packets were easily classified by their transport layer ports. As the use of random or non-standard ports is dramatically increasing, payload inspection [1,2] and statistical methods [3, 4] are proposed. However, drawbacks of these techniques such as insufficiency in encrypted traffic and their high computation cost lead to the emergence of behavioral classifiers. The merit of the behavioral classification is to use the behavioral pattern of an application instead of the content of packets or flow statistics. This point makes these classifiers useful for encrypted traffic or unknown protocols. But, so far, no automated method to obtain the behavioral

model of applications is provided which currently requires human inspection. There are a few studies for automating inference of the behavioral patterns which are application specific and cannot be widely used, for instance [5] has been presented for P2P-TV traffic.

To overcome the challenges of traffic classification and behavioral pattern detection approaches, we aim at providing an automatic approach to derive formal behavioral models, i.e., transition systems, for applications in the domain of the network. Our focus is mainly on programs at the application layer of the TCP/IP model [6]. We reduce our problem to the automata learning problem [7] which aims at inferring an automaton which accepts the set of given words. If the input traffic is considered as the given words of the language, then the desired model will be the identified automaton. However, the classical approaches in the literature of automata learning are not efficient to derive the most general model such that not only it subsumes valid unobserved traces as much as possible, but also disallows invalid traces. This is not achievable unless concepts of the domain are utilized to tailor the basic algorithm.

Intuitively, we assume that the behavior of an application can be identified in terms of how it executes well-known network protocols (below the application layer), abstracting the state variables of the application. Therefore, given the formal specification of well-known network protocols and execution traces of a program, we automatically generate a transition system. Hence, we customize the automata learning algorithm of [8] using rules in the network context to derive the most general model. Noting to the fact that each trace of a program is an interleaving of network protocol execution traces, called *flows*, the inferred model must preserve the behavior of each network protocol. In other words, the model of various applications differ in how they interleave the traces of well-known network protocols. Therefore, we take advantage of a behavioral pre-order relation in the theory of transition systems to conduct the process of model generation such that invalid traces are prohibited. Due to our abstraction (of application variables), the states of the inferred model which identify the same state with the same number of the flows for each network protocol, can be aggregated together using the counter abstraction technique [9] to include not observed behaviors.

To illustrate the applicability of our approach, we have implemented our algorithm in a tool and applied it on two version control system applications and two remote desktop sharing programs. Our results indicate that the techniques that are used to generalize the model, are sufficiently conservative such that no invalid trace is included and unobserved behaviors are covered with a high precision. Furthermore, the worst case time complexity order of our algorithm is linear rather than exponential in contrast to the related automata learning techniques.

2 Preliminaries

We first explain the network concepts utilized in our proposed method in Section 2.1. We provide an overview of automata learning in Section 2.2. We define the main concepts related to transition systems in Section 2.3. The counter abstraction technique is introduced in Section 2.4 which is used to find the equivalent states.

2.1 Network Background

Each packet transferred across a network is composed of two parts: the header and the content. The header includes the control information needed by the corresponded protocol and is appended to the beginning of the content. Protocols defined over the Internet follow the *TCP/IP* layered architecture [6]. This model consists of four layers: *Application*, *Transport*, *Internet*, and *Link*. The layered architecture means that the packet content of each layer is the built packet of its upper layer.

To send a message over the network, at first, the Application layer receives the user message from the software which is running (e.g., email client, web browser, instant messaging software, etc.), and passes it to the lower layer. The Transport layer segments data from the upper levels, then establishes a connection between the packet's point of origin and where it has to be received, and ensures that the packets are reassembled in the correct order [10]. The Network layer is responsible for the packet's addressing and routing. Finally, the Link layer manages the formats of packets based on the mediums being used in transmitting the packets. For each layer, a number of different protocols is standardized. Protocols are divided into connection-less and connection-oriented categories. Connection-oriented are those that need to establish a connection before data transmission. Thus there are handshake (initialization) and finalization phases in these protocols. These phases are not required in connection-less protocols. They just send a request packet for each desired data. A sequence of packets which have the same value for the parameters source IP, source port, destination IP, destination port, and the protocol name is called *flow*. An execution of an application gives rise to initiating a number of flows. These flows are the connections which are established between the initiator system and the other end systems.

2.2 Automata Learning

There are equivalent keywords in the literature to automata learning such as grammar inference or regular inference, language or automata identification. The goal of automata learning is to find a (non-unique) smallest automaton which is consistent with the set of given examples [11]. Gold has proved that this problem when the alphabet is finite, the two input sets of positive and negative samples are given, and the number of the states of the output automaton is determined, is a NP-complete problem [7]. If all of the words with the size equal and less than n are given, then it is possible to solve the problem in the polynomial time.

The algorithms of this problem can be divided into the two categories: active and passive.

The active techniques are based on Angluin L^* algorithm which solves the problem in polynomial time by asking some membership or equivalence queries [12]. It is assumed that there is an oracle than answer the required queries. Passive techniques tend to build tree-like automata, called prefix tree automata, from input examples and then by merging their states according to some heuristics evidence, achieve the smallest deterministic finite automata (this technique is called *Evidence Driven State Merging (EDSM)* [8]). In this category, there is no oracle and the algorithm should find the solution only from the positive and negative words of the language.

Since we aim to infer the behavioral model only from the input traces when there is no oracle system, our solution in this paper is based on the passive techniques. The most important challenge of these techniques is how to find the candidate states which should be merged to include unobserved traces. KTail algorithm merges states which have K common future (i.e., states that accept the same set of strings of length K) [13]. Several research has been conducted to decrease the $O(n^2)$ search space of the states which should be merged. Red-Blue is one of them and becomes a popular framework which limits the number of pairs of states by determining sufficient conditions on their colors.

2.3 Transition Systems

In this section we define the concepts used in the proposed methodology which are related to transition systems. These definitions have been adapted from [14].

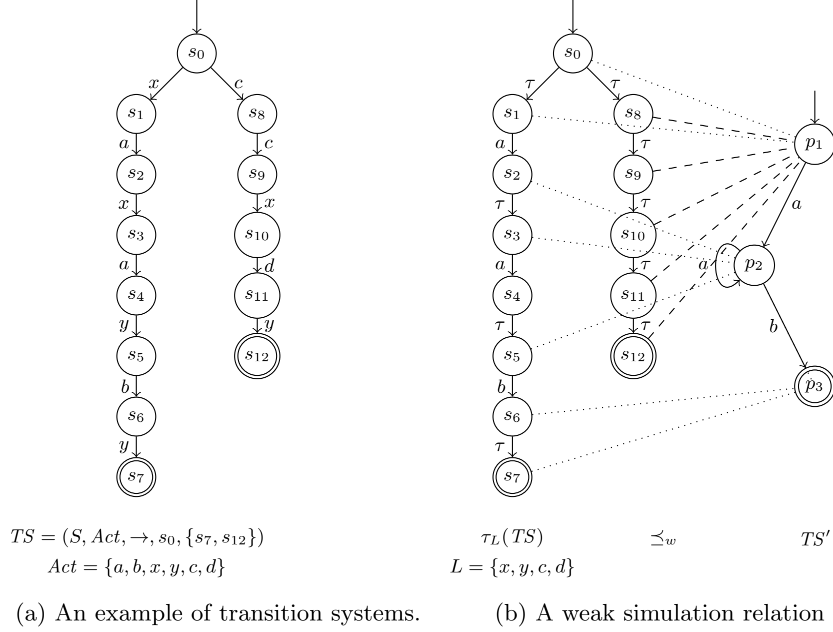
Definition 1 (Transition System). *A transition system is a tuple* $TS = (S, Act, \rightarrow, s_0, \downarrow)$ *where* S *is a set of states,* Act *is a set of actions,* $\rightarrow \subseteq S \times Act \times S$ *is a transition relation,* s_0 *is the initial state, and* $\downarrow \subseteq S$ *is a set of final states. We use* $s \overset{\alpha}{\rightarrow} t$ *to denote* $(s, \alpha, t) \in \rightarrow$.

TS *is called action-deterministic if for all* $s \in S$*, there are not* $(s, \alpha, t) \in \rightarrow$ *and* $(s, \alpha, v) \in \rightarrow$*, where* $\alpha \in Act$ *and* $t, v \in S$*, such that* $t \neq v$.

From this definition, the transition system of Fig. 1a is action-deterministic.

A finite *execution fragment* $\eta = s_0\alpha_0 s_1\alpha_1 \ldots \alpha_n s_{n+1}$ of TS is an alternating sequence of states and actions starting with the initial state and ending with a final state such that $(s_i, \alpha_i, s_{i+1}) \in \rightarrow$ where $0 \leq i \leq n$. A finite sequence of actions $\varrho = \alpha_0\alpha_1 \ldots \alpha_n$ of TS is an *execution trace* if $\exists s_0, \ldots, s_{n+1} \in S$ such that $\eta = s_0\alpha_0 s_1\alpha_1 \ldots \alpha_n s_{n+1}$ is an execution fragment. For instance, $s_0\ x\ s_1\ a\ s_2\ x\ s_3\ a\ s_4\ y\ s_5\ b\ s_6\ y\ s_7$ and $s_0\ c\ s_8\ c\ s_9\ x\ s_{10}\ d\ s_{11}\ y\ s_{12}$ are the execution fragments of the transition system in Fig. 1a. By eliminating states from these sequences, the execution traces are generated ($x\ a\ x\ a\ y\ b\ y$ and $c\ c\ x\ d\ y$ respectively).

There is an abstraction operator which has the responsibility to hide some actions of a transition system to make them internal and thus unobservable to external entities. We formally define the abstraction operator in the following definition:



(a) An example of transition systems. (b) A weak simulation relation

Fig. 1: A transition system, its abstraction, and its weak simulation relation.

Definition 2 (Abstraction Operator). *Let $TS = (S, Act, \to, s_0, \downarrow)$ be a transition system. The abstraction of TS via a set of actions $L \subseteq Act$, denoted by $\tau_L(TS)$, is $(S, Act \setminus L, \to', s_0, \downarrow)$ such that: $\to' = \{(s, \alpha, t) \mid (s, \alpha, t) \in \to, \alpha \notin L\} \cup \{(s, \tau, t) \mid (s, \alpha, t) \in \to, \alpha \in L\}$*

The left transition system in the Fig. 1b is the result of applying an abstraction on the transition system what has been abstracting $\{x, y, c, d\}$ of Fig. 1a.

To compare the behavior of transition systems, several behavioral pre-order and equivalence relations have been proposed ranging from strict to liberal ones. The *Simulation* relation is a finest pre-order relation which requires a transition system to precisely mimic transitions of another one [15]. In the case of existing internal actions in the system, the *Weak Simulation* relation is defined to relax the conditions only for the observable actions.

Let $\xrightarrow{\tau}^*$ be reflexive and transitive closure of τ-transitions:

- $t \xrightarrow{\tau}^* t$;
- $t \xrightarrow{\tau}^* s$ and $s \xrightarrow{\tau} r$, then $t \xrightarrow{\tau}^* r$.

Definition 3 (Weak Simulation Relation). *A binary relation R on the set of states S is a weak simulation relation if for any s_1, s_1', and $t_1 \in S$ and $\alpha \in Act$, $s_1 \mathcal{R} t_1$ implies:*

- $s_1 \xrightarrow{\alpha} s_1' \Rightarrow (\alpha = \tau \wedge s_1' \mathcal{R} t_1) \vee (\exists t_1', t_1'', t_1''' \in S : t_1 \xrightarrow{\tau}^* t_1''' \xrightarrow{\alpha} t_1'' \xrightarrow{\tau}^* t_1' \wedge s_1' \mathcal{R} t_1')$;

– $s_1 \in \downarrow \Rightarrow (\exists\, t'_1 \in \downarrow : t \xrightarrow{\tau}{}^{*} t')$.

For the given transition systems $TS_i = (S_i, Act_i, \rightarrow_i, s_{0i}, \downarrow_i)$*, where* $i \in \{1, 2\}$*,* TS_1 *is weakly simulated by* TS_2 *or* TS_2 *simulates* TS_1*, denoted by* $TS_1 \preceq_w TS_2$*, if* $s_{01}\ \mathcal{R}\ s_{02}$ *for some weak simulation relation* $\mathcal{R}$*.*

A weak simulation relation $\mathcal{R}$ is minimal, if for all simulation relation $\mathcal{R}'$ witnessing $TS_1 \preceq_w TS_2$, $\mathcal{R} \subseteq \mathcal{R}'$. Hence, a minimal weak simulation relation $\mathcal{R}$ is not necessarily unique. An example of weak simulation relations between two transition systems has been illustrated in Fig. 1b.

2.4 Counter Abstraction

The *Counter Abstraction* is a technique to abstract states of a system. The idea is to represent each state as a vector of counters one per each value instead of a vector of state variables. For instance, consider there are three integer variables x, y and z. The states $\{x = 1, y = 2, z = 1\}$, $\{x = 2, y = 1, z = 1\}$ and $\{x = 1, y = 1, z = 2\}$ are equivalent because they have the identical counter abstracted state $\{count(1) : 2, count(2) : 1\}$. This technique has been used in various applications. In symmetry reduction which is a technique to avoid state space explosion problem, the counter abstraction has the role of finding identical clusters of states space so as to reduce the symmetry states and decrease the cost of model checking [16]. This concept is also used in [9] in order to abstract a parameterized system of an unbounded size into a finite-state system to be verifiable.

3 Methodology

In this section, our proposed methodology for learning the network behavioral model of an application is discussed. To derive this model, it is assumed that the only application-dependent input is the set of network traffic captured during different executions of the application. We develop our solution based on the automata learning techniques and provide an evidence driven state merging algorithm based on rules in the network context to derive the most general model which subsumes unobserved traces as much as possible. This algorithm is based on the specification of well-known network protocols.

3.1 Problem Statement

The captured traffic is the sequence of packets sent or received as the result of the execution of an application during a specified time. Each packet contains data and headers of layers as the result of encapsulation. We only consider information of the upper layer instead of the whole headers and data (for instance, we only take into account information of the application layer of HTTP packets while they subsume information of the TCP layer).

To facilitate the processing of each packet content and close up the concept to the automata theory, a function is exploited which corresponds each packet to its equivalent action-like abstract representation. This function is defined as $PMapper : \mathit{Packets} \rightarrow \mathit{Act}$ where *Packets* is the set of possible packets captured through the pre-processing step. For example, a received TCP packet in the handshake phase is mapped to *TCPInitI* which is a member of the model actions set. In Section 4 we explain how the action set is defined in terms of the packet information. Therefore, by applying the *PMapper* function to each captured packet, we obtain a trace of actions. Hence, one input of our problem is N executions of an application which are transformed by the *PMapper* into the N *action traces* (packet trace) denoted by PT, ranged over by π. Let π_i indicate the ith action of the trace π, and $\mathit{len}(\pi)$ show the length of the trace. We remark that the length of each input execution is arbitrary, and in potentially independent of the length of other traces.

Besides the packet traces, another important input of our problem is the specifications of K network protocols. We assume that the specifications are provided in the form of action-deterministic transition systems $P_i = (S_i, \mathit{Act}_i, \rightarrow_i, s_{0i}, \downarrow_i)$ where $1 \leq i \leq K$, $\tau \notin \mathit{Act}_i$, and $\forall i, j \leq K : (\mathit{Act}_i \cap \mathit{Act}_j = \emptyset)$. We remark that each action trace π is the interleaving of a set of flows $f_1,\ f_2, \ldots$ where f_i is an execution trace of P_j $(j \leq K)$.

The goal of our problem is to derive a model in the form of transition system, i.e., $M = (S_M, \mathit{Act}_M, \rightarrow_M, s_{0M}, \downarrow_M)$ such that $\mathit{Act}_M = \bigcup_{\pi \in PT} \{\pi_i \mid 1 \leq i \leq \mathit{len}(\pi)\}$, and $\pi \in \mathit{Traces}(M)$, where $\mathit{Traces}(M)$ is the set of the execution traces of M. In fact, each of the input traces is an execution trace of the desired transition system.

3.2 Projection Relation

Initially, a tree-like automaton which consists of all action traces is generated. Intuitively, each application needs to establish a number of connections with other systems in order to perform each of its functionality. Each connection follows a protocol specification. For instance, an execution of the *Map* application of *Windows 8* contains four flows where two are for the DNS protocol, one for the TCP and one for the TLS protocols. Hence, each state of the initial transition system can be considered as a vector of states, each of which identifies a state of the corresponding protocol. Note that the size of the vector is equal to the number of flows. To generalize the initial transition system to cover more behavior, some states are selected to merge together. Hence, the new model accepts extra not observed valid behavior. Merged states are those called *project equivalent*. Two states are project equivalent if their vectors (of flow states) are identical with respect to the counter abstraction technique. For the sake of efficiency, the resulting transition system is determined.

Before describing the method, we mention some definitions and theorems. As we explained, each packet from the application execution belongs to a flow. We assume the total number of the flows of the all input traces is denoted by F. Furthermore, the auxiliary function $\mathit{Flow} : S \times \mathit{Act} \times S \rightarrow Nat$, defined over the

initial transition system, maps each packet, specified by the transition with the corresponding action of the packet, to its flow number such that $Flow(s, \alpha, t) \leq F$, where $s, t \in S$ and $\alpha \in Act$. From the flow definition each flow has a protocol attribute. Let function *Protocol* identify the protocol name of a flow, denoted by $Protocol : Nat \rightarrow Nat$, such that $\forall f \leq F : Protocol(f) \leq K$.

Definition 4 (Projection Relation). *Let $TS_i = (S_i, Act_i, \rightarrow_i, s_{0i}, \downarrow_i)$, for $i = 1, 2$, be transition systems such that $TS_1 \preceq_w TS_2$ witnessed by a minimal weak simulation relation $\mathcal{R}$. Two states s_1 and s_2 of S_1 have projection relation under TS_2 if $\exists\, t \in S_2 : s_1 \mathcal{R}\ t \wedge s_2 \mathcal{R}\ t$. Then, we say that s_1 and s_2 are the same projection of t under the transition system TS_2, denoted by $s_1 \sim_{\downharpoonleft TS_2} s_2$.*

To define states that are project equivalent, the following lemma identifies the conditions under which the project relation can act as an equivalence relation, and consequently can partition states. If a transition system has a *tree-like* structure, any of its two states can be connected by a unique simple path.

Lemma 1. *Let $TS_i = (S_i, Act_i, \rightarrow_i, s_{0i}, \downarrow_i)$, for $i = 1, 2$, be transition systems such that TS_1 is a tree-like transition system and TS_2 is an action-deterministic transition system without any τ-transition (i.e., $\tau \notin Act_2$). If TS_1 is weakly simulated by TS_2, witnessed by a minimal weak simulation $\mathcal{R}$, then each state of S_1 relates to only one state of S_2 under $\mathcal{R}$:*

$$\forall\, s \in S_1, \forall\, t_1, t_2 \in S_2 : s\ \mathcal{R}\ t_1\ \wedge\ s\ \mathcal{R}\ t_2 \Rightarrow t_1 = t_2$$

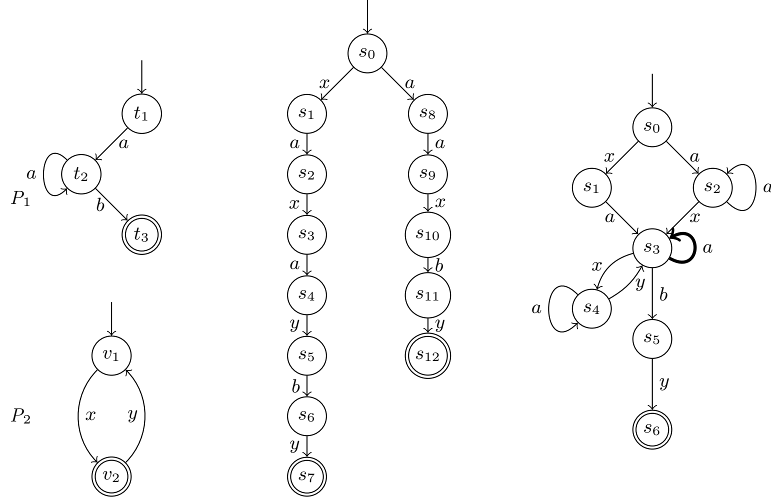
Theorem 1. *Let $TS_i = (S_i, Act_i, \rightarrow_i, s_{0i}, \downarrow_i)$, for $i = 1, 2$, be transition systems such that TS_1 is a tree-like transition system and TS_2 is an action-deterministic transition system without any τ-transition (i.e., $\tau \notin Act_2$). The projection relation under the transition system TS_2 over the states of TS_1 is an equivalence relation.*

See Appendix A.1 for the proof of Lemma 1 and Appendix A.2 for the proof of Theorem 1. As a consequence of Theorem 1, the states of a transition system can be partitioned into equivalance classes by a projection relation. The equivalence class for projection relation is defined in the following definition.

Definition 5 (Projection Relation Partitioning). *Let $TS_i = (S_i, Act_i, \rightarrow_i, s_{0i}, \downarrow_i)$, for $i = 1, 2$, be transition systems such that TS_1 is a tree-like transition system and TS_2 is an action-deterministic transition system without any τ-transition (i.e., $\tau \notin Act_2$). States of TS_1 are partitioned under the projection relation under TS_2 into the equivalence classes each of which is identified by the unique state $t \in S_2$ such that:*

$$[t]_{TS_1 \sim_{\downharpoonleft TS_2}} = \{s \in S_1\ \mid\ s\ \mathcal{R}\ t\}$$

where $\mathcal{R}$ is a minimal weak simulation relation.



(a) Specification of Protocols

(b) The initial transition system.

(c) After completing transitions.

Fig. 2: Applying the steps of proposed method on an example.

Running Example. Consider the specifications of two sample protocols in Fig. 2a, we assume that two input traces $x\ a\ x\ a\ y\ b\ y$ and $a\ a\ x\ b\ y$ are given. In the first trace, there are three flows, two are of the protocol P_2 and one of the protocol P_1. They are given such that the first x and the last y belongs to a flow and the second x and the first y are related together. In the second trace, there are two flows each of which is instantiated from each protocol. It is assumed that the flows are enumerated by the order of their first packets. We use this example in the rest of this section.

3.3 Step 1: Building the Initial Transition System

From Section 2, there is an execution fragment for each execution trace. We generate for each input trace π, its corresponding execution fragment $\eta^\pi = s_0 \pi_1 s_1^\pi \pi_2 \dots \pi_{len(\pi)} s_{len(\pi)}^\pi$. Note that the initial state in the fragments of the all traces are intentionally identical. In the first step, the tree-like transition system M_0 is built from aggregating the execution fragments of the input traces. Therefore, the initial transition system $M_0 = (S, Act, \rightarrow, s_0, \downarrow)$ is obtained as follows:

- $S = \{s_i^\pi \mid 1 \leq i \leq len(\pi),\ \pi \in PT\} \cup \{s_0\}$,
- $Act = \{\pi_i \mid 1 \leq i \leq len(\pi),\ \pi \in PT\}$,
- $\rightarrow = \bigcup_{\pi \in PT}(\{(s_k^\pi, \pi_{k+1}, s_{k+1}^\pi) \mid 1 \leq k \leq len(\pi)\} \cup \{(s_0, \pi_1, s_1^\pi)\})$,
- $\downarrow = \{s_{len(\pi)}^\pi \mid \pi \in PT\}$.

Fig. 2b is the result of performing this step. This initial transition system does not cover new execution traces of the application which are not given in the input. Therefore, some operations are needed to generalize the initial transition system and cover more execution traces. Next steps (step 2 and 3) are the efforts to reach this goal.

3.4 Step 2: Generalizing by Counter Abstraction

The generalization method in this step is addressed in two sub-steps.

1. Finding Equivalent States Intuitively, two states are equivalent under the flow f, if they belong to the same equivalence class based on the projection relation under the transition system of the attributed protocol of f, i.e., $Protocol(f)$. To this aim, we introduce the flow-based abstraction operator, which renames actions not included in the flow f to τ. By generalizing this intuition, two states s_1 and s_2 of transition system M_0 are equivalent if and only if they are equivalent under all flows of the initial transition system M_0.

Definition 6 (Flow-based Abstraction Operator). *Let* $TS = (S, Act, \rightarrow, s_0, \downarrow)$ *be a transition system. Then* $\tau_{\bar{f}}(TS) = (S, Act', \rightarrow', s_0, \downarrow)$ *such that:* $\rightarrow' = \{(s, \alpha, t) \in \rightarrow \mid Flow(s, \alpha, t) = f\} \cup \{(s, \tau, t) \mid \exists (s, \alpha, t) \in \rightarrow (Flow(s, \alpha, t) \neq f)\}$ *and* $Act' = Act_i$ *where* $Protocol(f) = i$ *and* $P_i = (S_i, Act_i, \rightarrow_i, s_{0i}, \downarrow_i)$.

For instance, the abstracted transition system of Fig. 2b under flow f_1 (contains $a\ a\ b$ and has been illustrated in the left side of Fig. 1b. We remark that if the abstraction operator is defined under protocol actions instead of a flow, then the resulting abstracted transition system may not preserve the protocol behavior due to interleaving of flows. For instance, the abstraction of the transition system in Fig. 2b under the protocol P_2 contains the sequence of $x\ \tau\ x\ \tau\ y\ \tau\ y$ at its left branch which does not have any weak simulation relation with P_2.

Let $count(s, t)$ denote the number of flows like f_i that the state t of the protocol P_j weakly simulates s in the abstraction of M_0 under f_i:

$$count(s, t) = \ \left| \{ f_i \leq F \mid s \in [t]_{\tau_{\bar{f}_i}(TS(M_0)) \sim_{\downarrow P_j}} \} \right| .$$

We remark that each state s is uniquely simulated by a state t as the result of our projection relation.

The two states s_1 and s_2 can be aggregated together under the counter abstraction technique if and only if $\forall\, j \leq K, t \in S_j : count(s_1, t) = count(s_2, t)$.

The results of applying the counter abstraction on the states of the initial transition system of the running example are presented in Table 1. To obtain each row, at first, the projection relation under abstraction of each flow is computed. After that, the number of flows in each state of protocols is counted. First row shows that all the flows are related to the initial states of the protocols. Because of the transition (s_0, x, s_1) of flow f_1, the state of s_1 is simulated by the state v_2 of the protocol P_2, and hence, the counter of flows in the state v_1

Table 1: Valuation of *count* function in each states of the initial transition system. Function c is the abbreviation of *count*.

state	$c(s,t_1)$	$c(s,t_2)$	$c(s,t_3)$	$c(s,v_1)$	$c(s,v_2)$	state	$c(s,t_1)$	$c(s,t_2)$	$c(s,t_3)$	$c(s,v_1)$	$c(s,v_2)$
s_0	5	0	0	5	0	s_7	4	0	1	5	0
s_1	5	0	0	4	1	s_8	4	1	0	5	0
s_2	4	1	0	4	1	s_9	4	1	0	5	0
s_3	4	1	0	3	2	s_{10}	4	1	0	4	1
s_4	4	1	0	3	2	s_{11}	4	0	1	4	1
s_5	4	1	0	4	1	s_{12}	4	0	1	5	0
s_6	4	0	1	4	1						

decreases by one and the counter of flows in the state v_2 increases by one. After calculating the counters for each state, the set of equivalent states are achieved: $\{(s_2, s_5, s_{10}), (s_3, s_4), (s_6, s_{11}), (s_7, s_{12}), (s_8, s_9)\}$.

2. Merging Equivalent States After finding the set of equivalent states of M_0, merging process should be done. Let $[s]$ donote the equivalence class of the state s, i.e., $\forall s' \in S : s' \in [s] \Leftrightarrow s' \equiv s$. A merged state inherits the union of the incoming and outgoing transitions of its origin states. By applying all merge candidates, the final transition system $M_1 = (S', Act, \rightarrow', s'_0, \downarrow')$ is obtained, where $S' = \{[s] \mid \forall s \in S\}$, $\rightarrow' = \{([s], \alpha, [t]) \mid \exists s, t \in S : (s, \alpha, t) \in\rightarrow\}$, $s'_0 = [s_0]$, and $\downarrow' = \{[s] \mid \forall s \in\downarrow\}$. Fig. 2c (without the tick transition a on the state s_3) is the final result of performing this step.

Remark 1. After this step, the resulting transition system is action-deterministic. We have proved this fact in the appendix.

3.5 Step 3: Generalizing by Completing Transitions

The next generalization idea is completing the transitions set according to the transition systems of the network protocols. We add self-loops of each protocol state $t \in P_i$, for some $i \leq K$, to the state $[s]$ if $count(s,t) > 0$. Adding such transitions does not affect the equivalent classes of M_1. Then, after applying this step, the resulting generalized transition system is $M_g = (S', Act, \rightarrow_g, s'_0, \downarrow')$ such that:

$$\rightarrow_g = \rightarrow' \cup \{([s], \alpha, [s]) \mid \forall j \leq K, t \in S_j, \forall s \in S : count(s,t) > 0 \wedge (t, \alpha, t) \in\rightarrow_j\}.$$

After applying this step, the tick transition a on state s_3 is added to the Fig. 2c. The time complexity of the algorithm is linear in the size of the input. See Appendix A.4 for the psuedocode of the algorithm and a discussion of the time complexity.

4 Evaluation

To evaluate the proposed method, we have implemented our algorithm in Java and applied it to some applications. Two categories of applications,*version control system* and *remote desktop sharing*, are selected for testing our methodology. For the first category, two applications TortoiseSVN client of SVN [1]and Source Tree Client of GIT [2] are selected. The traffic of the *update* command of these applications are gathered as their captured packet traces. Also, we have selected two remote desktop sharing applications, namely *TeamViewer* [3] and *JoinMe* [4], for which their traffic is encrypted. Hence, they cannot be easily identified by signature based approaches on the content of packets. Each one has run for 100 times and their network traces are captured via the Wireshark [5] tool. Packets of the application layer protocols (used by these programs), namely *TCP*, *SSL*, *SSLv2*, *TLSv1*, *TLSv1.2*, *HTTP* and *UDP* have been considered and the others are filtered. The more protocols are considered, the more precision will be achieved. Some preprocessing operations have been performed to eliminate the repetitive and truncated packets. Also, we have reassembled segments of fragmented packets. The mapper function which is responsible for translating the packets to their corresponding actions is defined such that it assigns the concatenation of the packet protocol name, the control phase and the direction to each packets. We divide the operation of each protocol into a set of phases to abstractly consider its progress. The control phases are assumed to be *Init*, *Data*, and *Fin* for connection-oriented protocols and *Init* and *Data* for connection-less ones. Intuitively, *Init* indicates to the establishment of the connection, *Data* to the transmission of data, and *Fin* to the termination of the connection. The direction is a binary tag which can be either I or O to indicate that the packet is sent or received, respectively. The amount of detail about packets embedded in their corresponded actions, shows how much the final generated model is sensitive to packet variations. By this mapper function, different manners of each control phase (initialization/ transferring data/ finalization) are considered to be the same.

We assume that the specifications of protocols are given in the form of transition systems and defined according to the mapper function abstraction level. By applying the mapper function on the packet traces, 100 action traces have been obtained for each application. These traces are divided into the train and test sets. The train traces are the input of our proposed method to infer the behavioral model which should accept the test traces. The overall scheme of an obtained model is shown in [**?**].We use the cross validation technique for 100 times to calculate the average value of precision with a reasonable confidence interval. Table 2 shows the final result of our experiments. Regarding to impossibility of measuring the real value of false positive rate (because it is not

[1] https://tortoisesvn.net/
[2] https://www.atlassian.com/software/sourcetree
[3] https://www.teamviewer.com/en/
[4] https://www.join.me/
[5] https://www.wireshark.org/

possible to gather all negative traces), researchers tend to consider the traces of the other applications which have the same functionality. Thus, we use traces of applications in the same category crossly to calculate the false positive rates.

Table 2: The average result of applying the proposed approach step by step, run on system with CPU Corei7 and 2G RAM. TPR stands for true positive rate.

Step	App	States Num	FP	TPR (observed)	TPR (unobserved)	Train time	Test time
Initial Transition System	SVN	3982	100%	100%	2 %	< 5sec	< 1sec
	GIT	4115	100%	100%	1 %		
	TeamViewer	8637	0	100%	0%		
	JoinMe	34484	0	100%	0%		
Applying Counter Abstraction	SVN	78	100%	100%	55 %	< 2min	< 1sec
	GIT	45	100%	100%	100%		
	TeamViewer	407	0	100%	36%		
	JoinMe	5458	0	100%	25 %		
Completing Self-Loop Transitions	SVN	78	100%	100%	100%	< 5min	< 1sec
	GIT	45	100%	100%	100%		
	TeamViewer	407	0	100%	98 %		
	JoinMe	5458	0	100%	56 %		
Relaxing Unnecessary Orders*	SVN	*	100%	100%	100%	*	*
	GIT	*	100%	100%	100%		
	TeamViewer	*	0	100%	100 %		
	JoinMe	*	0	100%	91 %		

The major point is that by applying our proposed generalization steps, the false positive rate does not grow. This means that our conservative approach prevents over-generalization from occurring. Each generalization step improves the completeness of the model. Note that since the update command of SVN and GIT generates a short packet trace, their captured traffic are similar and misclassified. As a future work, we plan to map packets to parametric actions in order to enhance the precision of the classifier. Adding (self-loop) transitions has increased our precision by 31 percent in the worst case. In the next step, we aim to relax unnecessary interleaving which stems from the concurrent development of applications or parallel network connections. Such a step which is our future work increases our precision to 100 or 91 percent. Now, we have applied the step manually, by examining the counter examples of the previous step. Those traces which can be covered by the generated model via modifying the orders of packets, is counted as the successful result for this step. We plan to automate this idea so as to automatically induce strict orders among transitions and relax the unnecessary ones in our future work. Our approach fails to recognize 9 percent of test traces (the last row of the Table 2) which are mainly those that include new unpredictable subsequences based on the train set.

4.1 Comparison with other packet classification methods.

To clarify the applicability of our methods, it should be compared with other packet classification techniques which we have described in Section 1. **Port-based** detection method does not have the ability of detecting most of the current applications because that they tend to use random or non-standard ports. Due to growing usage of encrypted traffic **payload inspection** methods become useless and it can not be used in our dataset. Furthermore, the proposed **behavioral classification** methods are application specific (e.x. for P2P applications) and they are not enough general to apply to our selected applications. Finally **statistical classification methods** are the only related work which we can compare our work with. To this aim, Netmate [6] is used to obtain the feature vectors of flows of captured traffic. Then, using Weka tool-set[7], the average precision of classification and false positive metrics among three algorithms SVM, Native Bayes and C4.5 were measured. The final result of these metrics are reported in Table 3.

Table 3: The result of statistical classification

Method	FP	TPR	Train Time	Test Time
TeamViewer	0.12 %	83 %	3 sec	< 1 sec
JoinMe	0.10 %	87 %	5 sec	< 1 sec

5 Related Work

Two research areas are related to our problem. In the following we explore related work in each area.

Automata Learning. Some research has been conducted to extend the expressiveness of inferred models. The KTail algorithm is extended in [17] with the aim to generate models from methods invocation traces. This approach is conducted in four steps. At first, the traces with the identical sequence of methods (those differ in the values of parameters) are merged together. Next, constraints on parameters are obtained via Daikon invariant detector [18]. At the third step, a prefix tree automaton is built. Finally, the states are merged according to a criterion which can be equivalence of method and parameters, weak subsumption or strong subsumption for their next k actions. In [19], the authors extends the Angluin L^* algorithm to infer relationships between input and output parameters in the form of the Mealy machines. In [20], the automate learning problem is extended to infer deterministic timed automata.

Some studies address the application of automata learning problem. Among them, [21] is the most related work to ours which elaborates on inferring mealy

[6] https://dan.arndt.ca/projects/netmate-flowcalc/
[7] http://www.cs.waikato.ac.nz/ml/weka/

machine models of communication protocols. The authors indicate that the parameters in the message format of protocols such as sequence number, configuration parameter and session id, result in infinite-states model. To minimize the state space, the abstract representation of protocol states are derived automatically in terms of operations that a requester and responder may perform. Hence, they have a similar assumption to ours which is the existence of protocol specifications. Their algorithm is based on query evaluation (active automata learning), while, we have extended the passive automata learning. Also, there are other applications of automata learning in different areas, especially in software specification mining [22, 23] which are not directly related to our work and we do not elaborate on.

Reverse Engineering of Protocol Specification. In this part we enumerate the works that focus on inferring protocol specifications from traffic. These works are related to ours because of their restriction on inferring a model by observing the behavior of the application in a black-box style. In [24], a probabilistic method was investigated to obtain a finite state machine of a protocol. It was assumed that the format of protocol messages is not determined. At the first step, messages are segmented into l-length bytes and clustered with the aim of recognizing their control parts. Next, the most frequent patterns are selected as message units by statistical analysis. Then, the main messages of the protocols are defined by computing the centers of the clusters. Finally, the finite state machine is constructed whose states are the main messages and probabilistic transitions are the frequencies of each pairs of messages.

In ReverX algorithm [25], a prefix tree automaton is built from traces and then the states which are the destination of identical transitions, are merged. Therefore, transitions with the same source and destination are created. They claim that if these transitions are merged the parameters of message headers are induced. Actually, despite their work is similar to us in using passive automata learning, we differ in the conditions for state merging. If the states are just similar in their 1-future action, they merge them, while we have investigated a domain specific condition based on well-known protocol.

6 Conclusion

The classical methods which identify the traffic based on packet header information or statistical metrics, are not effective anymore. Classification approaches based on the behavioral patterns of applications are of a new trend to this problem. No general and automated method to derive behavioral models has been provided. We proposed a method to reach this goal based on the automata identification problem and evidence driven state merging technique combined by transition system theories, namely behavioral pre-order relations and the counter abstraction. Intuitively, we assumed that the behavior of an application can be identified in terms of how it executes well-known network protocols, abstracting the state variables of the application. Hence, we have introduced our merging conditions to identify the equivalent states based on the specification of a set of

well-known network protocols such as TCP, TLS, SSL, etc. To this aim, we have provided the projection relation to identify the states with the same number of the flows for each network protocol which can be counted together using the counter abstraction technique.

We have presented two extra steps to complete the inferred model to cover unobserved behaviors of the software. At first, the model is completed by including the self-loop behaviors of the network protocols. After that, the possible valid interleaving of the packets based on the repetition of their orders is predicted. The model is extended to subsume such predicted orders. We also implemented and evaluated our procedure which does not require human inspection. The experiments show very encouraging results that the generalization steps significantly increase the accuracy from 0% to 91% in the worst case. The future work is to mechanize the last step which induces the essential orders with the aim of relaxing the unnecessary ones. We plan to extend our case study and compare the result of our method with the real traffic classification tools.

References

1. A. W. Moore and K. Papagiannaki, "Toward the accurate identification of network applications," in *Passive and Active Network Measurement*. Springer, 2005, pp. 41–54.
2. S. Sen, O. Spatscheck, and D. Wang, "Accurate, scalable in-network identification of p2p traffic using application signatures," in *Proceedings of the 13th international conference on World Wide Web*. ACM, 2004, pp. 512–521.
3. A. W. Moore and D. Zuev, "Internet traffic classification using bayesian analysis techniques," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 33, no. 1. ACM, 2005, pp. 50–60.
4. A. McGregor, M. Hall, P. Lorier, and J. Brunskill, "Flow clustering using machine learning techniques," in *Passive and Active Network Measurement*. Springer, 2004, pp. 205–214.
5. P. Bermolen, M. Mellia, M. Meo, D. Rossi, and S. Valenti, "Abacus: Accurate behavioral classification of p2p-tv traffic," *Computer Networks*, vol. 55, no. 6, pp. 1394–1411, 2011.
6. K. R. Fall and W. R. Stevens, *TCP/IP illustrated, volume 1: The protocols*. addison-Wesley, 2011.
7. E. M. Gold, "Language identification in the limit," *Information and Control*, vol. 10, no. 5, pp. 447 – 474, 1967.
8. K. J. Lang, B. A. Pearlmutter, and R. A. Price, *Grammatical Inference: 4th International Colloquium, ICGI-98 Ames, Iowa, USA, July 12–14, 1998 Proceedings*. Springer Berlin Heidelberg, 1998, ch. Results of the Abbadingo one DFA learning competition and a new evidence-driven state merging algorithm, pp. 1–12.
9. A. Pnueli, J. Xu, and L. D. Zuck, "Liveness with (0, 1, infty)-counter abstraction," in *Proceedings of the 14th International Conference on Computer Aided Verification*, ser. CAV '02. London, UK, UK: Springer-Verlag, 2002, pp. 107–122.
10. C. Parsons, *Deep Packet Inspection in Perspective: Tracing Its Lineage and Surveillance potentials*. Citeseer, 2008.
11. M. J. Heule and S. Verwer, "Exact DFA identification using SAT solvers," in *Grammatical Inference: Theoretical Results and Applications*. Springer, 2010, pp. 66–79.
12. D. Angluin, "Learning regular sets from queries and counterexamples," *Inf. Comput.*, vol. 75, no. 2, pp. 87–106, Nov. 1987.
13. A. Biermann and J. Feldman, "On the synthesis of finite-state machines from samples of their behavior," *Computers, IEEE Transactions on*, vol. C-21, no. 6, pp. 592–597, June 1972.
14. C. Baier, J.-P. Katoen *et al.*, *Principles of model checking*. MIT press Cambridge, 2008, vol. 26202649.
15. R. J. van Glabbeek, *The linear time - branching time spectrum*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1990, pp. 278–297. [Online]. Available: http://dx.doi.org/10.1007/BFb0039066
16. E. A. Emerson and R. J. Trefler, "From asymmetry to full symmetry: New techniques for symmetry reduction in model checking," in *Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, ser. CHARME '99. London, UK, UK: Springer-Verlag, 1999, pp. 142–156. [Online]. Available: http://dl.acm.org/citation.cfm?id=646704.702007

17. D. Lorenzoli, L. Mariani, and M. Pezzè, "Automatic generation of software behavioral models," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 501–510.
18. M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Trans. Softw. Eng.*, vol. 27, no. 2, pp. 99–123, Feb. 2001.
19. A. Khalili and A. Tacchella, "Learning nondeterministic mealy machines," in *Proceedings of the 12th International Conference on Grammatical Inference, ICGI 2014, Kyoto, Japan, September 17-19, 2014.*, 2014, pp. 109–123.
20. S. E. Verwer, "Efficient identification of timed automata: Theory and practice," Ph.D. dissertation, TU Delft, Delft University of Technology, 2010.
21. F. Aarts, B. Jonsson, and J. Uijen, "Generating models of infinite-state communication protocols using regular inference with abstraction," in *Testing Software and Systems*. Springer, 2010, pp. 188–204.
22. N. Walkinshaw, J. Derrick, and Q. Guo, "Iterative refinement of reverse-engineered models by model-based testing," in *Proceedings of the 2Nd World Congress on Formal Methods*, ser. FM '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 305–320. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-05089-3_20
23. D. Lo and S. Maoz, "Scenario-based and value-based specification mining: Better together," *Automated Software Engineering*, vol. 19, no. 4, pp. 423–458, 2012.
24. Y. Wang, Z. Zhang, D. D. Yao, B. Qu, and L. Guo, *Applied Cryptography and Network Security: 9th International Conference, ACNS 2011, Nerja, Spain, June 7-10, 2011. Proceedings.* Springer Berlin Heidelberg, 2011, ch. Inferring Protocol State Machine from Network Traces: A Probabilistic Approach, pp. 1–18.
25. J. Antunes, N. Neves, and P. Verissimo, "Reverse engineering of protocols from network traces," in *Reverse Engineering (WCRE), 2011 18th Working Conference on*, Oct 2011, pp. 169–178.

A Appendix

A.1 Proof of Lemma 1: The minimal weak simulation relation relates each states to only one state.

Proof by contradiction: It is assumed that this does not hold ($t_1 \neq t_2$). The tree-like structure of TS_1 indicates that each state of S_1 is the target of exactly one transition which is not a self-loop. Therefore, since s essentially has a unique parent, denoted by $p(s)$, two cases can be distinguished: 1) either it is reachable through a τ-transition, then $\tau \notin Act_2$, minimality of $\mathcal{R}$, and Definition 3 imply that $p(s)\ \mathcal{R}\ t_1$ and $p(s)\ \mathcal{R}\ t_2$; 2) or it is reachable through an observable action, then minimality of $\mathcal{R}$, and Definition 3 together imply that the parents of t_1 and t_2 must have relation with $p(s)$, i.e., $p(s)\ \mathcal{R}\ p(t_1)$ and $p(s)\ \mathcal{R}\ p(t_2)$. By the same reasoning, if we move backward one by one, their parents become the initial states of both TS_1 and TS_2. However, the root of TS_2 is reachable through transitions with the same observable action, which it is in contradiction with the action-deterministic assumption. Therefore, the premise $t_1 \neq t_2$ is violated.

A.2 Proof of Theorem 1: The projection relation is an equivalence relation.

We prove that the projection relation has three reflexive, symmetric, and transitive properties. The reflexive property results from the fact that $(s,t) \in \mathcal{R} \wedge (s,t) \in \mathcal{R} \Rightarrow s \sim_{\downharpoonleft TS} s$. To prove the symmetric property, $s_1 \sim_{\downharpoonleft TS} s_2$ implies $\exists\, t \in S_2 : (s_1,t) \in \mathcal{R} \wedge (s_2,t) \in \mathcal{R}$. Consequently, $(s_2,t) \in \mathcal{R} \wedge (s_1,t) \in \mathcal{R}$ implies $s_2 \sim_{\downharpoonleft TS} s_1$. The transitive property of the relation is concluded by the facts that $s_1 \sim_{\downharpoonleft TS} s_2 \Rightarrow \exists\, t \in S_2 : (s_1,t) \in \mathcal{R} \wedge (s_2,t) \in \mathcal{R}$, and $s_2 \sim_{\downharpoonleft TS} s_3 \Rightarrow \exists\, r \in S_2 : (s_2,r) \in \mathcal{R} \wedge (s_3,r) \in R$. According to Lemma 1, it holds that $t = r$, and hence, $s_1 \sim_{\downharpoonleft TS} s_3$.

A.3 Proof of Remark 1: After generalization by counter abstraction, the resulting transition system is action-deterministic.

The resulting transition system after merging the equivalent states is action-deterministic, since the network protocol specifications are action-deterministic. Suppose there is $[s] \xrightarrow{\alpha} [t_1]$ and $[s] \xrightarrow{\alpha} [t_2]$ such that $t_1 \not\equiv t_2$. Since $\alpha \in \mathit{Act}_j$, then there exists $t \in S_j$ such that $\mathit{count}(s,t) > 0$ which weakly simulates s in the abstraction M_0 under flow f, where $\mathit{Protocol}(f) = j$. By Definitions 3 and 5, there exists $t'_1, t'_2 \in S_j$ that $t \xrightarrow{\alpha} t'_1$ and $t \xrightarrow{\alpha} t'_2$ where $t_1 \in [t'_1]_{\sim_{\downharpoonleft P_j}}$ and $t_2 \in [t'_2]_{\sim_{\downharpoonleft P_j}}$, respectively, which contradicts to the assumption that P_j is action-deterministic.

A.4 Pseudocode of behavioral model identification algorithm

Algorithm 1 indicates the pseudocode of our methodology. Obviously, building the initial transition system takes linear time based on the number of packets (lines 7-12). For each input trace π, the projection partitioning of its states is computed for all flows in $O(\mathit{len}(\pi))$. Therefore, the corresponding state of the protocols for all the states is achieved in $O(\Sigma_{\pi \in PT} \mathit{len}(\pi))$ (lines 14-22). Due to the tree-like structure of the initial transition system and the lack of both τ and nondeterministic actions in transition systems of protocols, it can be shown that a minimal weak simulation relation can be implicitly achieved by Simultaneously traversing of the initial transition system and the transitions systems of protocols (lines 18 - 22).

The counting operation is performed to calculate the number of flows are in the same state of the protocol for each state during the previous step (line 22). Finding the equivalent states and merging them is fulfilled with a hash structure, so it takes linear time in size of the states (lines 24-30) .

The last step to add the self-loops is of $O(SL \times \Sigma_{\pi \in PT} \mathit{len}(\pi))$, where SL is the number of self-loops in all the protocols and it is assumed that the self-loops are founded as the result of a preprocess (lines 32-35).

We can find an upper bound for the number of flows for each functionality of applications (e.g., M) which is established for each software ($M = \mathit{Max}(F_\pi)$,

where Max is the maximum operator). Then, the time complexity of the approach equals $O((M + SL) \times \Sigma_{\pi \in PT} len(\pi))$. Since M can be assumed as a constant variable for an application and SL is not application-dependent, the total time complexity of the algorithm is linear in the size of the input.

Algorithm 1 Pseudocode of transition system identification

Input: Sequences of packets(η^{π}), Transition systems of protocols $TS_i = (S_i, Act_i, \rightarrow_i, s_{0i}, \downarrow_i)$, The set of initiated flows $Flows$

Output: Transition system $TS = (S, Act, \rightarrow, s_0, \downarrow)$

1: $s_0 := NewState()$
2: $S := \{s_0\}$
3: $Act := \cup Act_i$
4: $\rightarrow :=$ Undefined for all domain
5: $\downarrow := \emptyset$
6: ▷ Building the initial transition system
7: **for all** $\eta^{\pi} = s_0 \pi_1 s_1^{\pi} \pi_2 \ldots \pi_{len(\pi)} s_{len(\pi)}^{\pi} \in \eta$ **do**
8: $\quad \rightarrow := \rightarrow \cup \{(s_0, \pi_{j+1}, s_1^{\pi})\}$
9: $\quad$ **for all** $j \in [0, len(\pi) - 1]$ **do**
10: $\quad\quad S := S \cup \{s_{j+1}\}$
11: $\quad\quad \rightarrow := \rightarrow \cup \{(s_j, \pi_{j+1}, s_{j+1}^{\pi})\}$
12: $\quad \downarrow := \downarrow \cup \{s_{len(\pi)}\}$
13: ▷ Finding the equivalent states
14: **for all** $s \in S$ **do**
15: $\quad v_s :=$ a new vector with the size of $|Flows|$
16: $\quad$ **for all** $f_k \in Flows$ **do**
17: $\quad\quad cnt(s, v_s(f_k)) := 0$
18: **for all** $\eta^{\pi} = s_0 \pi_1 s_1^{\pi} \pi_2 \ldots \pi_{len(\pi)} s_{len(\pi)}^{\pi} \in \eta$ **do**
19: $\quad$ **for all** $j \in [1, len(\pi)]$ **do**
20: $\quad\quad$ **for all** $f_k \in Flows$ **do**
21: $\quad\quad\quad v_{s_j}[f_k] := StateOfProtocol(v_{s_{j-1}}[f_k], Protocol(f_k), \pi_j)$
22: $\quad\quad\quad cnt(s_j, v_{s_j}(f_k)) := cnt(s_j, v_{s_j}[f_k]) + 1$
23: ▷ Merging the equivalent states
24: **for all** $s \in S$ **do**
25: $\quad key =$ a new vector with the size of $| \cup_{i \in |Protocols|} S_i|$
26: $\quad$ **for all** $f_k \in Flows$ **do**
27: $\quad\quad key[v_s[f_k]] = cnt(s, v_s[f_k])$
28: $\quad HashMap.add(key, s)$
29: **for all** $key \in HashMap$ **do**
30: $\quad Merge$(states in $HashMap.get(key)$)
31: ▷ Completing the transitions by adding self-loops
32: **for all** (state sl, transition α) $\in selfloops$ **do**
33: $\quad$ **for all** $s \in S$ **do**
34: $\quad\quad$ **if** $cnt(s, sl) > 0$ **then**
35: $\quad\quad\quad \rightarrow := \rightarrow \cup \{(s, \alpha, s)\}$

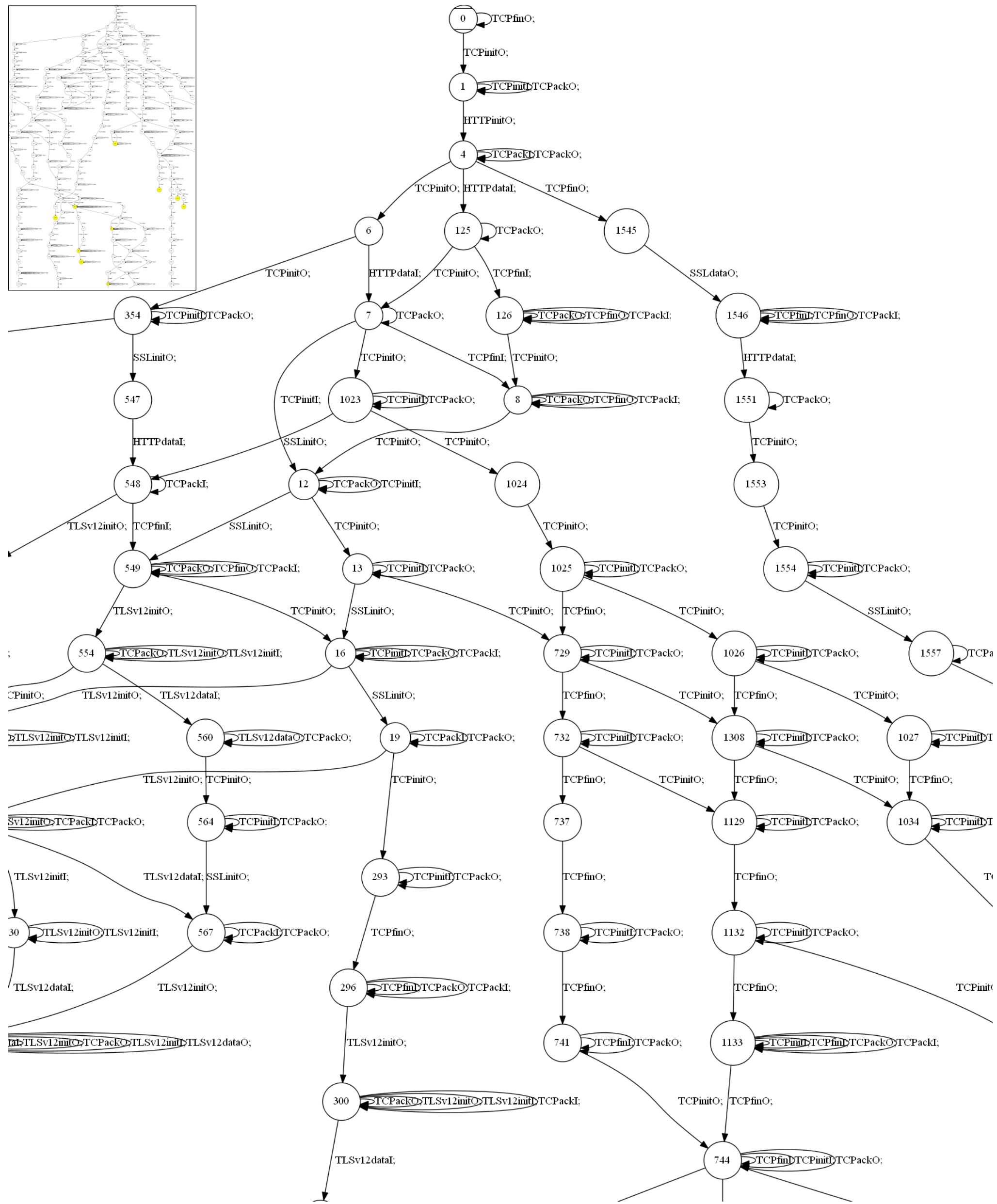


Fig. 3: Overall view of an identified model for one of the case studies.