Learning Fragments of the TCP Network Protocol

Paul Fiterău-Broştean*, Ramon Janssen, and Frits Vaandrager

Institute for Computing and Information Sciences
Radboud University Nijmegen
P.O. Box 9010, 6500 GL Nijmegen, the Netherlands
{P.FiterauBrostean,f.vaandrager}@cs.ru.nl, ramon.janssen@student.ru.nl

Abstract. We apply automata learning techniques to learn fragments of the TCP network protocol by observing its external behavior. We show that different implementations of TCP in Windows 8 and Ubuntu induce different automata models, thus allowing for fingerprinting of these implementations. In order to infer our models we use the notion of a mapper component introduced by Aarts, Jonsson and Uijen, which abstracts the large number of possible TCP packets into a limited number of abstract actions that can be handled by the regular inference tool LearnLib. Our work improves upon previous work by Uijen, who learned an overapproximation of the behavior of a TCP implementation in the ns-2 simulator.

1 Introduction

Our society has become reliant on the security and application of protocols used for various operations. Standards describing these protocols typically fail to specify what an agent should do in case another agent does not follow the rules of the protocol, which can result in exploits by hackers. Moreover, implementations of these standards can differ, and may deviate slightly from the official standard, resulting in security vulnerabilities. Automata learning techniques can help expose and/or mitigate such problems through tools that help generate state models for these systems.

Learning techniques enable the inference of state models for systems available as black boxes. Inferring such models is important not only for understanding these systems, but also for model checking and model based testing. To this end, several learning algorithms and tools have been developed, such as those presented in: [5, 19, 16, 3, 21, 12].

Whereas learning algorithms such as L* [5], work for systems with limited numbers of abstract inputs and outputs, many protocols make use of messages with parameters, for instance sequence numbers or flags. Moreover, network protocols may remember variables. For example, the TCP protocol requires remembering several variables for maintaining a connection. Efforts have been made to

* Supported by NWO project 612.001.216: Active Learning of Security Protocols (ALSEP).

develop techniques to learn these more complex systems. In particular, building on the extension of the L* algorithm used to learn Mealy machines (Niese [11]), F. Aarts et al. describe in [4] a methodology for learning systems via abstraction. This method entails introducing a *mapper* component in-between the protocol and the learner. The *mapper* reduces the parameters and state variables implied by the protocol to a small number of abstract values, on which learning algorithms can then be applied. By using this technique, they were able to infer state models of simulated versions of the Transmission Control Protocol (TCP) and the Session Initiation Protocol (SIP).

Advancements have been made in constructing the *mapper* automatically. Tomte [1] is a tool that builds mappers automatically for a subset of scalarset Mealy Machines. State machines from this subset can only test the equality of existing parameters and outputs, which is insufficient to learning systems that implement operations on data. (for instance, sums or other linear operations)

The goal of this work is to use abstraction to learn implementations of the TCP-protocol for different operating systems. We use abstraction based on the approach described in [1], but extend it to include the increment operator which is needed to learn the TCP protocol. While our setup specifically targets TCP, it can be easily adapted to learn other protocols.

Related work. In addition to software simulations of network protocols, the methodology described by Aarts et al. [4] was also used to infer state diagrams of embedded control software [20] and banking cards [2]. Dawn Song et al. [7] developed techniques to learn the state diagram of a network protocol used to control botnets. Learning techniques were also used to learn HTTP interaction models for web applications, as part of the SPaCIoS Project [6].

Organization. The paper is structured as follows: Section 2 gives a brief description of the TCP network protocol, Section 3 sets the context of regular inference with abstraction. Section 4 presents the framework we implemented to learn the TCP network protocol. Section 5 explains how the setup implements abstraction. Section 6 explains difficulties encountered and how we managed them. Section 7 presents experiments carried out to learn TCP. Section 8 outlines conclusions and related work.

2 The TCP Network Protocol

The transmission control protocol [14], or TCP, is a connection-based network protocol that allows two application programs to transfer data bidirectionally, in a reliable and orderly manner. The programs can run on the same or on separate machines. TCP supports data transfer through the *connection* abstraction where a *connection* comprises two endpoints associated with each of the two programs. A connection progresses from one state to the next following events, which are user actions, incoming *segments* containing flags, sequence and acknowledgement numbers, and timeouts. Connection progression is depicted in Figure 1.

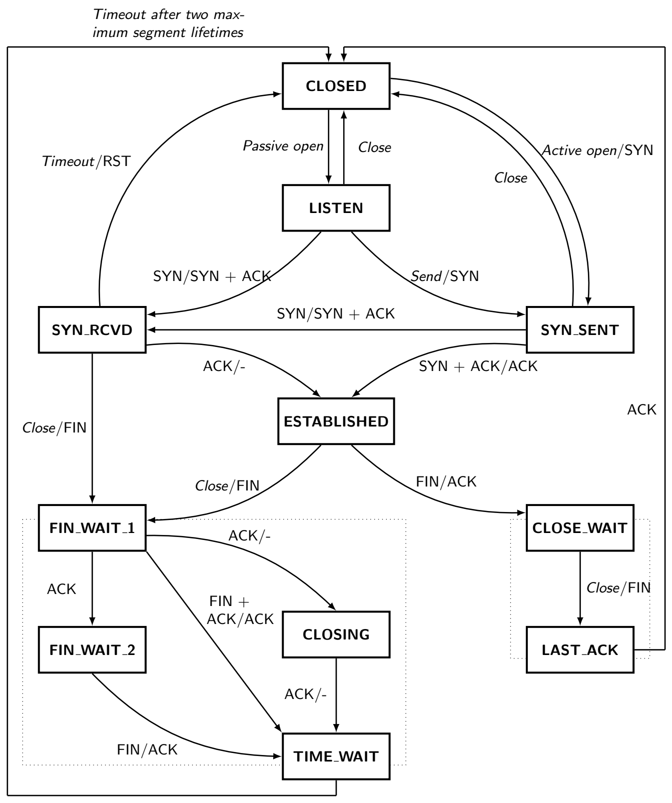


Fig. 1. A state diagram describing TCP [1]

We give a brief example of how the protocol functions from connection initiation to termination. For brevity, we use FLAGS-segment as shorthands for segments having the mentioned control flags activated.

The two systems communicating through TCP have different roles: one system acts as a *server* and the other as a *client*. Connection between *server* and *client* is established through the three-way handshake. Assuming the *server* is in the LISTEN-state, waiting for a *client* to connect to it, the *client* sends a SYN-segment on an ACTIVE OPEN-action and transitions to the SYN_SENT-state. On receiving this segment, the *server* responds with a SYN+ACK-segment, transitioning to the SYN_RCVD-state. The *client* then acknowledges the *server*'s segment with an ACK-segment and transitions to the ESTABLISHED-state. On receiving this segment, the *server* also transitions to the ESTABLISHED-state, connec-

[1] Code to generate this model was retrieved from `http://www.texample.net/tikz/examples/tcp-state-machine/`. Copyright 2009 Ivan Griffin. Reprinted under the LaTeX Project Public License, version 1.3.

tion is established and data can be transferred, thus concluding the three-way handshake.

When either side has finished sending data, that side sends a segment with a FIN-flag on an ACTIVE CLOSE-action, signaling that it has no more data left to send. This can be acknowledged with a FIN+ACK-segment if the other device also wants to close the connection, or with a ACK-segment if that device still wants to send data. Once the device has sent all data, it sends a FIN-message, closing the connection.

Notice that the state diagram in Figure 1 does not fully specify the behaviour of the TCP-implementation. More specifically, the model does not reveal what response is given in case either side receives a segment for which no transition is defined (for instance, if a RST-segment is received in the SYN_RCVD-state). Moreover, the model abstracts away from the sequence and acknowledgment numbers found in TCP packets. Many of these details can be inferred from the protocol standard. There are, however, some details which are implementation specific, with each operating system providing its own TCP implementation. Hence, inferring models of these TCP implementations represents a valuable asset in analyzing their concrete behavior.

3 Regular Inference Using Abstraction

In this section, we recall the definition of a Mealy machine, the basic ideas of regular inference in Angluin-style, and the notion of a mapper which allows us to learn "large" models with data parameters.

3.1 Learning Mealy Machines

We will use Mealy machines to model TCP protocol entities. A Mealy machine $\mathcal{M}$ is the tuple $\mathcal{M} = \langle I, O, Q, q_0, \rightarrow \rangle$, where

- I, O, and Q are nonempty sets of *input symbols*, *output symbols*, and *states*, respectively,
- $q_0 \in Q$ is the *initial state*, and
- $\rightarrow \subseteq Q \times I \times O \times Q$ is the *transition relation.*

Transitions are tuples of the form $(q, i, o, q') \in \rightarrow$. A transition implies that, on receiving an input $i \in I$, when in the state $q \in Q$, the machine jumps to the state $q' \in Q$, producing the output $o \in O$. Mealy machines are deterministic if for every state and input, there is exactly one transition. A Mealy machine is finite if I and Q are finite sets.

Angluin described L* in [5], an algorithm to learn deterministic finite automata. Niese [11] adapted this algorithm to learning deterministic Mealy machines. Improved versions of the L^* algorithm were implemented in the LearnLib tool [15, 13], developed at the Technical University of Dortmund. A graphical model of the basic learning setup is given in Figure 2.

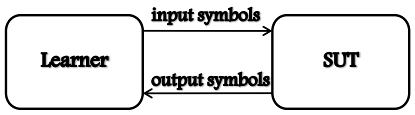


Fig. 2. Overview of the learner and the SUT

We assume an implementation, or System Under Test (*SUT*), and postulate that its behavior can be described by a deterministic Mealy Machine $\mathcal{M}$. The *learner*, connected to the *SUT*, sends *inputs* (or queries) to the *SUT* and observes resulting *outputs*. After each observation, the *learner* sends a special *reset* message, prompting the reset of the implementation. Based on the observation of the outputs, it builds a hypothesis $\mathcal{H}$. The hypothesis is then tested against the implementation. Testing involves running a number of test sequences which determine whether the hypothesis conforms to the *SUT*. The hypothesis is returned if all test sequences show conformation, otherwise it is further refined on the basis of the new counterexample. This process is repeated until all equivalence queries are passed.

3.2 Inference using Abstraction

Existing implementations of inference algorithms only proved effective when applied to machines with small alphabets (sets of input and output symbols). Practical systems like the TCP protocol, however, typically have large alphabets, e.g. inputs and outputs with data parameters of type integer or string.

A solution to this problem was proposed by Aarts et al in [4]. In this work, the concrete values of every parameter are mapped to a small domain of abstract values in a history-dependent manner. A *mapper component* is placed in-between the *learner* and the *SUT*. The *learner* sends abstract inputs comprising abstract parameter values to this component. The mapper component then turns the abstract values into concrete values (by taking the inverse of the abstraction function), forming concrete inputs, and sends them to the *SUT*. The concrete outputs received from the *SUT* are subsequently transformed back to abstract outputs and are returned to the *learner*. Reset messages sent by the *learner* to the *SUT* also reset the mapper component. A graphical overview of the *learner* and mapper component is given in Figure 3.

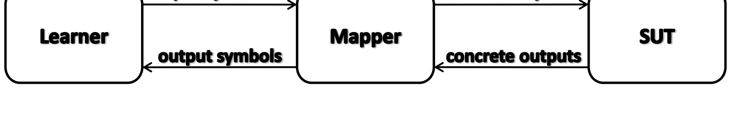


Fig. 3. Overview of the learner, the mapper and the SUT

Formally, the behavior of the intermediate component is fully determined by the notion of a *mapper* $\mathcal{A}$, which essentially is just a deterministic Mealy machine. A mapper encompasses both concrete and abstract sets of input and output symbols, a set of states, an initial state, a transition function that tells us how the occurrence of a concrete symbol affects the state, and an abstraction function which, depending on the state, maps concrete to abstract symbols. Each mapper $\mathcal{A}$ induces an abstraction operator $\alpha_{\mathcal{A}}$, which transforms a concrete Mealy machine with concrete inputs I and outputs O into an abstract Mealy machine with abstract inputs X and outputs Y. If the behavior of the SUT is described by a Mealy machine $\mathcal{M}$ then the SUT and the mapper component together are described by the Mealy machine $\alpha_{\mathcal{A}}(\mathcal{M})$. Dually, each mapper also induces a concretization operator $\gamma_{\mathcal{A}}$, which transforms an abstract hypothesis Mealy machine $\mathcal{H}$ with inputs X and outputs Y into a concrete Mealy machine with inputs I and outputs O. A key result proved by Aarts et al [4] is that $\alpha_{\mathcal{A}}(\mathcal{M}) \leq \mathcal{H}$ implies $\mathcal{M} \leq \gamma_{\mathcal{A}}(\mathcal{H})$, where $\leq$ denotes behavioral inclusion of Mealy machines. This result allows us to transform an abstract model $\mathcal{H}$, inferred through interaction with a mapper component, into a concrete model that over-approximates the behavior of the SUT.

4 Learning Setup

In the case of TCP, the SUT is the *server* in the TCP communication. On the other side, the *learner* and *mapper* simulate the *client*. On the client side we also introduce the *adapter*, a component that translates between messages and segments that are to be sent over the network. More specifically, it builds request segments from concrete inputs, sends them to the server, retrieves the response segments and infers the respective concrete outputs, which it delivers to the *learner-mapper* assembly. It is important to make distinction between the *mapper* and *adapter*. Whereas the *mapper* implements mapping between abstract and concrete messages, the *adapter* transforms these concrete messages to a format that is readable by the SUT.

With that said, we present in Figure 4 the framework implemented to learn fragments of the TCP implementation. On the learner side, we use *LearnLib* [16] and *Tomte* [1], two Java based learning tools. LearnLib provides the Java implementation of the L* based learning algorithm, while we use some of Tomte's libraries to connect the *learner* to a Java based *mapper* via direct method calls. A Python adapter based on *Scapy* [17] is used to craft, send request packets and retrieve response packets. Communication between the *mapper* and *adapter* is done over sockets.

We conducted our experiments on both a single and on two separate machines. The *client* and *server* reside in separate operating systems. When learning a model of the *server*, the resulting model describes the implementation of TCP for the operating system on which the *server* resides. Each operating system enables the user to configure parameters involved in TCP. These parameters can also have an influence over the resulting model. We used *Wireshark*

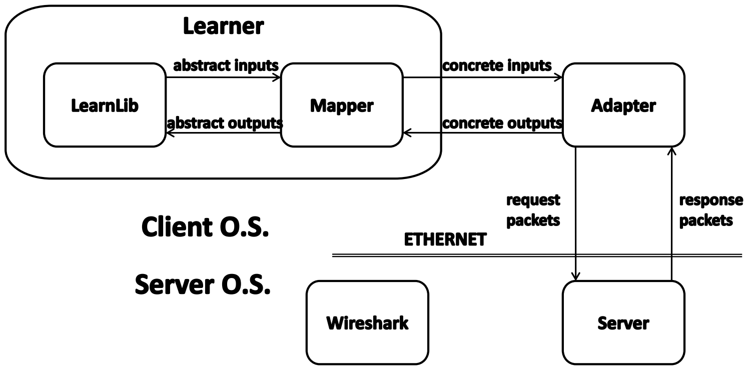


Fig. 4. Overview of the experimental setup

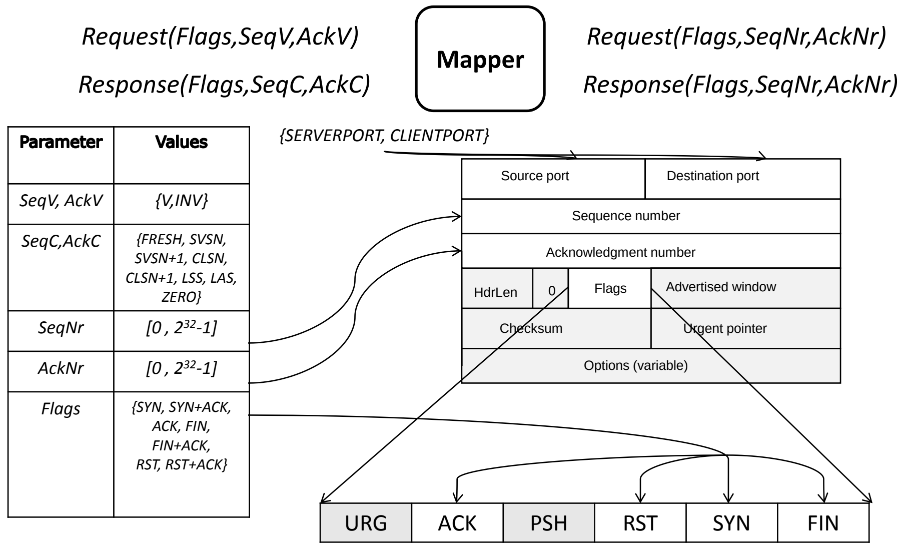
to monitor communication between *client* and *server*. Note that Scapy cannot communicate locally over the localhost(127.0.0.1) interface [18]. Consequently, communication is held over an *Ethernet* connection.

The experiments were carried out with the *server* deployed on Windows 8 and Ubuntu 13.10 respectively. The *server* passively listens for incoming connections on a port while the *learner*, acting as a "fake client", sends messages to the *server* through its own port.

5 Messages and Abstraction

As mentioned previously, we use a mapper component that translates abstract input messages into concrete input messages, and concrete output messages into abstract output messages. More specifically, parameters contained in messages are mapped from a concrete to an abstract domain, and vice versa. Figure 5 shows the concrete and abstract parameters used in learning. Also shown is how the concrete parameters are then associated with fields within TCP segments. Our message selection is based on the work of Aarts et al. in [4]. Like in their work, both inputs and outputs are generated based on the sequence number, acknowledgement number and flags found in each TCP segment.

Both the concrete and the abstract alphabets comprise *Request* inputs and *Response* outputs. Each of these inputs and outputs takes 3 parameters corresponding to the sequence number, acknowledgment number and the TCP flags. The concrete parameters *SeqNr* and *AckNr* are defined as 32 bit unsigned integers, while their corresponding abstract parameters *SeqV* and *AckV* are either *valid* or *invalid*. The *Flags* parameter can have the values ACK, SYN, FIN, RST, or any valid combination as listed in Figure 5. These flags correspond to bitfields of the control register in the TCP-frame, in which flags are either set or unset. In other words, each element of *flags* defines which flags have been set: all flags mentioned are set, all other flags are not.



Parameter	Values
SeqV, AckV	*{V,INV}*
SeqC,AckC	*{FRESH, SVSN, SVSN+1, CLSN, CLSN+1, LSS, LAS, ZERO}*
SeqNr	*[0 , 2^{32}-1]*
AckNr	*[0 , 2^{32}-1]*
Flags	*{SYN, SYN+ACK, ACK, FIN, FIN+ACK, RST, RST+ACK}*

Fig. 5. Message scheme

We abstract away from the sequence and acknowledgement numbers sent to the *server* by way of validity. We define validity based on whether the sequence and acknowledgement numbers comply to the standard TCP flow. Here the sequence number sent is equal to the last acknowledgement number received while the acknowledgement number sent is equal to the last sequence number received (the *server* sequence number) plus the length of the data that the *client* expects to receive plus 1 in case the segment carries a SYN or a FIN flag. ACK flags do not lead to any increase. Numbers that comply to this standard are *valid*, those that don't are *invalid*. We except from this rule whenever the server is in the LISTENING-state with no connection set up. In this case, new sequence numbers are generated via ISN (Initial Sequence Number), a number generation algorithm. We abstract away from this algorithm by deeming all generated numbers *valid* and none *invalid* at this stage. Consequently, we ignore any messages containing *invalid* parameters that the *learner* generates at this point. (we do not create and send segments for these inputs)

We abstract away from the sequence and acknoweldgement numbers received from the *server* by comparing them with values encountered in the communication up to that point. These values are stored in state variables which are maintained by the *mapper*. We also define the abstract output *timeout* for the case when no response segment is received.

In order to map between abstract and concrete the *mapper* maintains the following state variables:

- *lastSeqSent* stores the last sequence number sent by the client
- *lastAckSent* stores the last acknowledgment number sent by the client

– *lastFlagSent* stores the last flag sent by the client
– *valClientSeq* stores the last valid sequence number sent by the client
– *lastAbsSeq* stores whether the last sequence number sent by the client was valid
– *lastAbsAck* stores whether the last acknowledgement number sent by the client was valid
– *InitServSeq* stores the first sequence number from the server
– *INIT* records whether the server is in its initial state.

Variables *lastSeqSent*, *lastAckSent*, *lastFlagSent*, *lastAbsSeq*, *lastAbsAck* and *InitServSeq* store the first or most recent occurrence of certain message parameters. Variable *valClientSeq* stores the last valid sequence number: its definition is based on knowledge of the protocol. Variable *INIT* records whether the server is in its initial state.

On the basis of these variables, we define below the functions for *Request* transmission, *Response* receival, and *Timeout*. For symmetry, we use *SeqAbs* and *AckAbs* as notations for both abstract input and abstract output parameters.

```
function REQUEST($Flags$, $SeqNr$, $AckNr$)
    $lastFlagSent \leftarrow Flags$
    if $INIT \lor SeqNr == valClientSeq$ then
        $SeqAbs \leftarrow valid$
        $valClientSeq \leftarrow SeqNr$
    else
        $SeqAbs \leftarrow invalid$
    end if
    $lastSeqSent \leftarrow SeqNr$
    $lastAbsSeq \leftarrow SeqAbs$
    if $INIT \lor AckNr == InitServSeq + 1$ then
        $AckAbs \leftarrow valid$
    else
        $AckAbs \leftarrow invalid$
    end if
    $lastAckSent \leftarrow AckNr$
    $lastAbsAck \leftarrow AckAbs$
    return $Request(Flags, SeqAbs, AckAbs)$
end function
```

In the context of *Response* outputs, we compare values found in server responses to a set of reference values. Note the similarity between the conditions we chose and the conditions used in nmap [10] to perform OS fingerprinting. Also note that we assume no collision between different reference values. It is indeed possible that the client sequence number is equal to the server sequence number. However, the likelihood is very low due to the extended range these numbers can take values in.

```
function RESPONSE($Flags$, $SeqNr$, $AckNr$)
    if $INIT$ then
        $SeqAbs \leftarrow$ FRESH
```

```
            $\mathit{InitServSeq} = \mathit{SeqNr}$
        **else**
            $\mathit{SeqAbs} \leftarrow$ ABSTRACT($\mathit{SeqNr}$)
        **end if**
        $\mathit{AckAbs} \leftarrow$ ABSTRACT($\mathit{AckNr}$)
        **if** $\mathit{AckAbs} == \mathbf{CLSN + 1}$ **then**
            $\mathit{InitServSeq} \leftarrow \mathit{SeqNr}$
        **end if**
        $\mathit{INIT} \leftarrow \mathit{IsInitial}()$
         **return** $\mathit{Response}(\mathit{Flags}, \mathit{SeqAbs}, \mathit{AckAbs})$
    **end function**
    **function** ABSTRACT($\mathit{concVal}$)
        **if** $\mathit{concVal} == \mathit{valClientSeq} + 1$ **then**
            $\mathit{absVal} \leftarrow \mathbf{CLSN + 1}$
        **else if** $\mathit{concVal} == \mathit{valClientSeq}$ **then**
            $\mathit{absVal} \leftarrow \mathbf{CLSN}$
        **else if** $\mathit{concVal} == \mathit{InitServSeq}$ **then**
            $\mathit{absVal} == \mathbf{SVSN}$
        **else if** $\mathit{concVal} == \mathit{InitServSeq} + 1$ **then**
            $\mathit{absVal} \leftarrow \mathbf{SVSN + 1}$
        **else if** $\mathit{concVal} == \mathit{lastSeqSent}$ **then**
            $\mathit{absVal} \leftarrow \mathbf{LSS}$
        **else if** $\mathit{concVal} == \mathit{lastAckSent}$ **then**
            $\mathit{absVal} \leftarrow \mathbf{LAS}$
        **else if** $\mathit{concVal} == 0$ **then**
            $\mathit{absVal} \leftarrow \mathbf{ZERO}$
        **else**
            $\mathit{absVal} \leftarrow \mathbf{INV}$
        **end if**
         **return** $\mathit{absVal}$
    **end function**
    **function** TIMEOUT
        $\mathit{INIT} \leftarrow \mathit{IsInitial}()$
    **end function**
```

One important aspect is the detection of the initial state (ie. the LISTENING-state in the TCP protocol, *INIT* in the mapper definition). Such awareness is necessary in order to follow the TCP flow, wherein transitions from this state imply that new sequence numbers have to be generated for both *client* and *server*. The server and client sequence number variables (*clientSN* and *serverSN*) must be updated accordingly. For this purpose, we define an oracle which responds whether the system is in the initial state. We found that the oracle for Windows 8 can be implemented by a function over the state variables stored in the mapper and the output parameters. For Ubuntu 13.10, definition of such a function was made difficult by the fact that, depending on the system's current state, the abstract input *Request*(ACK+RST, *valid*, *invalid*) either resets the system or is ignored. If the oracle is defined via a function, definition of this function depends on the operating system for which TCP is learned. Below we show definition of *INIT* for Windows 8.

```
function IsInitial
    if $IsResponse$ then
        $INIT \leftarrow RST \in Flags \wedge SeqAbs = valid \wedge SYN \in lastFlagSent$
    else if $IsTimeout$ then
        $INIT \leftarrow INIT \vee (lastAbsSeq = valid \wedge RST \in lastFlagSent)$
    end if
end function
```

6 Complications encountered

To learn the system, several issues had to be addressed. Firstly, we had to implement resetting mechanisms that would prompt the TCP connection to return to the start state. We did this by either opening a new connection to the *server* on a different port each time we started a new query or by resetting the connection. In the first case, we were hit by thresholds on the number of connections allowed on a given state. More specifically, for Windows 8, once the *server* reaches a certain number of connections left in the CLOSE_WAIT-state, the *server* proceeds to send FIN+ACK-segment to close all connections. The second approach implies sending a RST-segment with a *valid* sequence number, which was possible with the mapper previously described. Consequently, we opted for this approach for our final experiments. Another possible approach would have been resetting the server application. We believe this approach, ignoring the mechanisms involved, would also prove problematic since there are instances when a connection on the *server* is blocked in a certain state for a period of time. During that period the server cannot terminate. We found this situation with the SYN_RCVD-state on Windows 8.

We also had to manage the handling of SYN+ACK retransmits. When the *server* is in the SYN_RCVD-state, it expects a corresponding ACK-segment to the SYN+ACK-segment it sent, thus ending the 3 way handshake. In this situation, the TCP protocol specifies that, if the *server* does not receive the expected acknowledgement within a time frame, it re-sends the SYN+ACK-segment a number of times after which it closes the connection. This behavior is not accounted for because it would require timer adjustments to fit with the retransmission time frame. We disabled SYN+ACK retransmission for Ubuntu by setting the *tcp_synack_retries* to 0. No such settings were needed in Windows 8 since the whole query can be executed within one the retransmission time frame.

We also encountered difficulties with packet receival. By analyzing packet communication we found that Scapy sometimes misses fast *server* responses, that is, responses sent after a short time span from their corresponding requests. We believe this could be caused by Scapy's slow performance in intercepting responses quick enough. To circumvent this problem, we crafted a network tracking tool based on Impacket [8] and Pcapy [9] which augments Scapy's receival capabilities. In case Scapy does not receive any responses back, the tracking tool either confirms that no response was intercepted or returns the response that Scapy missed.

Our experiments were also affected by the operating system on which the *learner* setup was deployed. The Ubuntu operating systems the *learner* was run on are unaware of what network packets are sent by Scapy, and therefore cannot recognize the response packets sent by the *server*. More specifically, as a TCP-connection is set up by the learner, the operating system notices a connection that it has not set up itself. Consequently, it responds with a RST-segment to shut down that connection. The problem was solved by blocking outgoing segments with a firewall, so that the RST-segments do not close the connection. This can be done in Ubuntu using the *iptables* command.

We encountered many stumbling blocks in our attempt to learn a model for Ubuntu 13.10. The TCP implementation for Ubuntu is very time sensitive and many factors interfere that can invalidate learning. We tried to eliminate as many of the time related factors as possible, since they cannot be learned with current learning tools.

First of all, with SYN+ACK retransmission disabled on Ubuntu 13.10, the *server* drops the connection quickly if it does not get a proper ACK response (and transitions to LISTENING-state). Consequently, when running long sequence of inputs, we were often faced with non-determinism caused by this reset. Moreover, we could not find a parameter that increases the time spent in SYN_RCVD. To go around this problem we used short queries (6-8 length) when learning the TCP implementation for Ubuntu 13.10.

Next, we faced occasional delayed ACK-segments on Ubuntu. Whenever an ACK-segment was delayed, the *adapter* was detecting *timeout* instead of the delayed acknowledgements. To improve on this, we set higher bounds for timeouts, but low enough so that the *server* doesn't transition from SYN_RCVD to LISTENING.

For Ubuntu 13.10 we also discovered that, whenever in the CLOSE_WAIT-state, the *server* behaves non deterministically when the client *client* sends ACK and FIN+ACK-segments with *valid* sequence number and *invalid* acknowledgement number. On receiving these packets, the *server* either retransmits the the ACK-segment it sent when acknowledging the FIN+ACK-segment that lead to the CLOSE_WAIT-state or it sends a timeout. At the moment, we cannot track this behaviour down to a timing parameter. Because of this issue, we eliminated these inputs from the alphabet. We also eliminated the abstract input *Request*(ACK+RST, *valid*, *invalid*) from the alphabet since it reset the system only in certain states. This made definition of the initialization function impossible without overly complicating the mapper.

To sum up, we faced many challenges learning Ubuntu 13.10's implementation of TCP and we do not have yet full insight on their association with Ubuntu's TCP parameters. Investigation of this behaviour is a subject for future work. In spite this, under the circumstances described, we were able to learn a model unaffected by many of Ubuntu's TCP timing triggers.

7 Experimental Results

We learned models for Windows 8 and Ubuntu 13.10 LTS. As mentioned previously, the *client* and *server* reside in different operating systems. Because the *adapter* can only function under Linux, we ran the learner setup (or *client*) on Ubuntu systems.

For Windows 8, the *client* was deployed on a guest virtual machine using the Ubuntu 12.04 LTS operating system, while the *server* resided on the Windows 8 host. We also experimented with the *client* residing on a separate computer running Ubuntu 13.10 that communicated with the same Windows 8 server and obtained similar results. Similarly for Ubuntu 13.10, the *server* and *client* were deployed both on one computer, each in its own Ubuntu virtual machine within the same Windows 8 host, and on separate machines.

In order to reduce the size of the diagram, we eliminate all self loops that have *timeout* outputs. Moreover, we use the initial flag letters as shorthands: *s* for *syn*, *a* for *ack*, *f* for *fin* and *r* for *rst*. We condense $Request/Response(flags, seq, ack)$ to $flags(seq, ack)$ and we group inputs that have the same abstract output and resulting state. *valid* and *invalid* abstract parameters are shorthanded to *v* and *inv* respectively. Finally, inputs that have the same effect regardless of the *valid* or *invalid* value of a parameter are merged and the parameter is replaced with _.

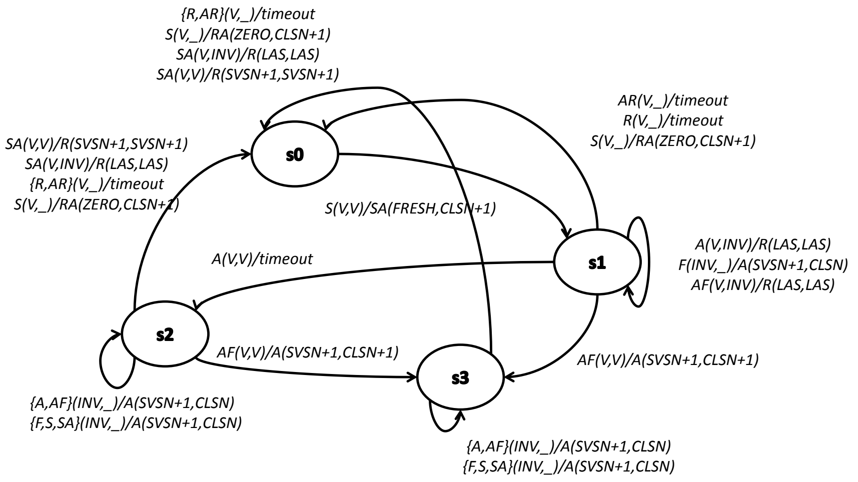


Fig. 6. Learned model for Windows 8 TCP

Figures 6 and 7 show the state models learned for the two operating systems. Both models depict 4 states of the reference model. We can identify handshake and termination on the two diagrams by following the sequence of inputs: S(v,v), A(v,v), AF(v,v). We see that for each input in the sequence the same output is generated. There are, however, notable differences, like for example the verbosity of the listening state in case of Ubuntu 13.10. RST-segment responses also differ. Whereas in Ubuntu 13.10, a RST-segment response always caries a 0 acknowledgment number, for Windows 8, similar to its joining sequence number, it takes the value of the last acknowledgment number sent resulting in RST(*las*,*las*) outputs.

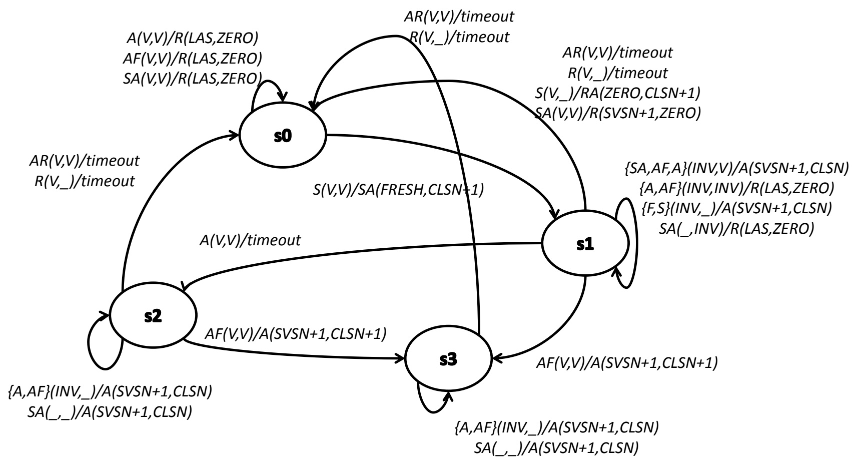


Fig. 7. Learned model for Ubuntu 13.10 TCP

8 Concluding Remarks and Future Work

We defined and implemented a learning setup for the inference using abstraction of the TCP network protocol. We then used this setup to learn fragments of different implementations of the TCP protocol, more specifically, the Windows 8 and Ubuntu 13.10 implementations. We learned these implementations on the basis of flags, acknowledgement and sequence numbers.

For our experiments, we built initial state predicting mappers tailored to the operating system's TCP implementation. These mappers reduced the number of concrete inputs and outputs to a small set of abstract inputs and outputs. We ran our setup for each mapper and respective operating system and, in each case, covered 4 states of the TCP protocol.

Comparing the models obtained for each protocol, we found a slight variation between the Windows and Ubuntu implementations of the TCP protocol. While in normal scenarios behavior turned out to be similar, some abnormal scenarios revealed differences in the values of sequence and acknowledgement numbers, as well as the flags found in response packets. This variation results in differing transitions between the state machines inferred for each OS. The difference in behavior is already leveraged by tools such as nmap(see [10]), as a means of operating system fingerprinting.

In the future we also plan to further investigate the behavior of the Ubuntu implementation. Moreover, we want to extend the TCP alphabet so that we also account for data transfer. We also aim to learn fragments of protocols built over TCP, for instance SSH and FTP. In order to facilitate future experiments, we also aim to automate the learning process. We believe the mapper can be constructed automatically using tools that automatically infer invariants over sets o values. These tools can then be harnessed by algorithms that automatically construct the mapper. Such an algorithm already exists and is implemented in Tomte, however, the only invariant it supports is equality. We believe we can extend this constraint to simple linear invariants. Such an extension would enable the automatic inference of mappers for more complex systems, including the TCP protocol.

References

1. Aarts, F.: Tomte. `http://www.italia.cs.ru.nl/tomte/`
2. Aarts, F., de Ruiter, J., Poll, E.: Formal models of bank cards for free. In: 4th International Workshop on Security Testing, Luxembourg, March 22, Proceedings, SECTEST 2013 (2013)
3. Aarts, F., Heidarian, F., Kuppens, H., Olsen, P., Vaandrager, F.: Automata learning through counterexample guided abstraction refinement. In Giannakopoulou, D., Mry, D., eds.: FM 2012: Formal Methods. Volume 7436 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2012) 10–27
4. Aarts, F., Jonsson, B., Uijen, J.: Generating models of infinite-state communication protocols using regular inference with abstraction. In Petrenko, A., Simo, A., Maldonado, J., eds.: Testing Software and Systems. Volume 6435 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2010) 188–204 Full version avalable at `https://pms.cs.ru.nl/iris-diglib/src/getContent.php?id=2013-Aarts-InferenceRegular`.
5. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. **75**(2) (November 1987) 87–106
6. Bchler, M., Hossen, K., Mihancea, P.F., Minea, M., Groz, R., Oriat, C.: Model inference and security testing in the spacios project. In: Proc. CSMR-WCRE, IEEE (2014) 411–414
7. Cho, C.Y., Babi ć, D., Shin, E.C.R., Song, D.: 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 '10, New York, NY, USA, ACM (2010) 426–439
8. Corelabs: Impacket. `http://corelabs.coresecurity.com/index.php?module=Wiki&action=view&type=tool&name=Impacket`
9. Corelabs: Pcapy. `http://corelabs.coresecurity.com/index.php?module=Wiki&action=view&type=tool&name=Pcapy`
10. Fyodor: Nmap. `http://nmap.org/book/osdetect.html`
11. Hagerer, A., Hungar, H., Niese, O., Steffen, B.: Model generation by moderated regular extrapolation. In Kutsche, R.D., Weber, H., eds.: FASE02. Volume 2306 of LNCS., SV (2002) 80–95
12. Howar, F., Steffen, B., Jonsson, B., Cassel, S.: Inferring canonical register automata. In Kuncak, V., Rybalchenko, A., eds.: Verification, Model Checking,

and Abstract Interpretation. Volume 7148 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2012) 251–266
13. Merten, M., Steffen, B., Howar, F., Margaria, T.: Next generation LearnLib. In Abdulla, P., Leino, K., eds.: TACAS. Volume 6605 of Lecture Notes in Computer Science., Springer (2011) 220–223
14. Postel (editor), J.: Transmission Control Protocol - DARPA Internet Program Protocol Specification (RFC 3261) (September 1981) Available via `http://www.ietf.org/rfc/rfc793.txt`.
15. Raffelt, H., Steffen, B., Berg, T., Margaria, T.: LearnLib: a framework for extrapolating behavioral models. STTT **11**(5) (2009) 393–407
16. Raffelt, H., Steffen, B., Berg, T., Margaria, T.: Learnlib: a framework for extrapolating behavioral models. International Journal on Software Tools for Technology Transfer **11**(5) (2009) 393–407
17. Secdev: Scapy. `http://www.secdev.org/projects/scapy/`
18. Secdev: Scapy troubleshooting. `www.secdev.org/projects/scapy/doc/troubleshooting.html`
19. Shahbaz, M., Groz, R.: Inferring mealy machines. In Cavalcanti, A., Dams, D., eds.: FM 2009: Formal Methods. Volume 5850 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2009) 207–222
20. Smeenk, W., Vaandrager, F., Janssen, D.: Applying automata learning to embedded control software. (2013)
21. SPaCIoS: Deliverable 2.2.1: Method for assessing and retrieving models (2013)