Combining Model Learning and Model Checking to Analyze TCP Implementations

Abstract. We combine model learning and model checking in a challenging case study involving Linux, Windows and FreeBSD implementations of TCP. We use model learning to infer models of different software components and then apply model checking to fully explore what may happen when these components (e.g. a Linux client and a Windows server) interact. Our analysis reveals several instances in which TCP implementations do not conform to their RFC specifications.

1 Introduction

Our society has become completely dependent on network and security protocols such as TCP/IP, SSH, TLS, BlueTooth, and EMV. Protocol specification or implementation errors may lead to security breaches or even complete network failures, and hence many studies have applied model checking to these protocols in order to find such errors. Since exhaustive model checking of protocol implementations is usually not feasible [23], two alternative approaches have been pursued in the literature. This article proposes a third approach.

A first approach, followed in many studies, is to use model checking for analysis of models that have been handcrafted starting from protocol standards. Through this approach many bugs have been detected, see e.g. [11,25,41,20,8,27]. However, as observed in [10], the relationships between a handcrafted model of a protocol and the corresponding standard are typically obscure, undermining the reliability and relevance of the obtained verification results. In addition, implementations of protocols frequently do not conform to their specification. Bugs specific to an implementation can never be captured using this way of model checking. For instance, [17] shows that both the Windows 8 and Ubuntu 13.10 implementations of TCP violate the standard. In [37], new security flaws were found in three of the TLS implementations that were analyzed, all due to violations of the standard. In [13] and [42] it was shown that implementations of a protocol for Internet banking and of SSH, respectively, violate their specification.

A second approach has been pioneered by Musuvathi and Engler [30]. Using the CMC model checker [31], they model checked the "hardest thing [they] could think of", the Linux kernel's implementation of TCP. Their idea was to run the *entire* Linux kernel as a CMC process. Transitions in the model checker correspond to events like calls from the upper layer, and sending and receiving packets. Each state of the resulting CMC model is around 250 kilobytes. Since CMC cannot exhaustively explore the state space, it focuses on exploring states that are the most different from previously explored states using various

heuristics and by exploiting symmetries. Through their analysis, Musuvathi and Engler found four bugs in the Linux TCP implementation. One could argue that, according to textbook definitions of model checking [16,7], what Musuvathi and Engler do is not model checking but rather a smart form of testing.

The approach we explore in this paper uses model learning. Model learning, or active automata learning [40,6,1], is emerging as a highly effective technique to obtain models of protocol implementations. In fact, all the standard violations reported in [17,37,13,42] have been discovered (or reconfirmed) with the help of model learning. The goal of model learning is to obtain a state model of a system by providing inputs to and observing outputs from a black-box system. This approach makes it possible to obtain models that fully correspond to the observed behavior of the implementation. Since the models are derived from a finite number of observations, we can (without additional assumptions) never be sure that they are correct: even when a model is consistent with all the observations up until today, we cannot exclude the possibility that there will be a discrepancy tomorrow. Nevertheless, through application of conformance testing algorithms [26], we may increase confidence in the correctness of the learned models. In many recent studies, state-of-the-art tools such as LearnLib [40] routinely succeeded to learn correct models efficiently. In the absence of a tractable white-box model of a protocol implementation, a learned model is often an excellent alternative that may be obtained at relatively low cost.

The main contribution of this paper is the combined application of model checking, model learning and abstraction techniques in a challenging case study involving Linux, Windows and FreeBSD implementations of TCP. Using model learning and abstraction we infer models of different software components and then apply model checking to fully explore what may happen when these components (e.g. a Linux client and a Windows server) interact.

The idea to combine model checking and model learning was pioneered in [34], under the name of *black box checking*. In [29], a similar methodology was introduced to use learning and model checking to obtain a strong model-based testing approach. Following [34,29], model checkers are commonly used to analyze models obtained via automata learning. However, these applications only consider specifications of a single system component, and do not analyze networks of learned models. Our results considerably extend previous work on learning fragments of TCP by [17] since we have (1) added inputs corresponding to calls from the upper layer, (2) added transmission of data, (3) inferred models of TCP clients in addition to servers, and (4) learned models for FreeBSD in addition to Windows and Linux. In order to obtain tractable models we use the theory of abstractions from [2], which in turn is inspired by earlier work on predicate abstraction [15,28]. Whereas in previous studies on model learning the abstractions were implemented by ad-hoc Java programs, we define them in a more systematic manner. We provide a language for defining abstractions, and from this definition we automatically generate mapper components for learning and model checking.

The main advantage of our method compared to approaches in which models are handcrafted based on specifications is that we analyze the "real thing" and may find "bugs" in implementations. In fact, our analysis revealed several instances in which TCP implementations do not conform to the standard. Compared to the white-box approach of Musuvathi and Engler [30], our black-box method has two advantages: (1) we obtain explicit component models that can be fully explored using model checking, and (2) our method appears to be easier to apply and is more flexible. For instance, once we had learned models of the Linux implementation of TCP, it was easy for us to extend our results to the setting of Windows and FreeBSD. A similar extension would be a major effort in the approach of [30] and even impossible without direct access to the code.

2 Background on model learning

Mealy machines During model learning, we represent protocol entities as Mealy machines. A *Mealy machine* is a tuple $\mathcal{M} = \langle I, O, Q, q^0, \rightarrow \rangle$, where I, O, and Q are finite sets of *input actions*, *output actions*, 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*. We write $q \xrightarrow{i/o} q'$ if $(q, i, o, q') \in \rightarrow$. We assume $\mathcal{M}$ to be *input enabled* (or *completely specified*) in the sense that, for each state q and input i, there is a transition $q \xrightarrow{i/o} q'$, for some o and q'. We call $\mathcal{M}$ *deterministic* if for each state q and input i there is exactly one output o and one state q' such that $q \xrightarrow{i/o} q'$. We call $\mathcal{M}$ *weakly deterministic* if for each state q, input i and output o there is exactly one state q' with $q \xrightarrow{i/o} q'$.

Let $\sigma = i_1 \cdots i_n \in I^*$ and $\rho = o_1 \cdots o_n \in O^*$. Then ρ is an *observation* triggered by σ in $\mathcal{M}$, notation $\rho \in A_{\mathcal{M}}(\sigma)$, if there are $q_0 \cdots q_n \in Q^*$ such that $q_0 = q^0$ and $q_{j-1} \xrightarrow{i_j/o_j} q_j$, for all j with $0 \leq j < n$. If $\mathcal{M}$ and $\mathcal{M}'$ are Mealy machines with the same inputs I and outputs O, then we write $\mathcal{M} \leq \mathcal{M}'$ if, for each $\sigma \in I^*$, $A_{\mathcal{M}}(\sigma) \subseteq A_{\mathcal{M}'}(\sigma)$. We say that $\mathcal{M}$ and $\mathcal{M}'$ are *(behaviorally) equivalent*, notation $\mathcal{M} \approx \mathcal{M}'$, if both $\mathcal{M} \leq \mathcal{M}'$ and $\mathcal{M}' \leq \mathcal{M}$.

If $\mathcal{M}$ is deterministic, then $A_{\mathcal{M}}(\sigma)$ is a singleton set for each input sequence σ. In this case, $\mathcal{M}$ can equivalently be represented as a structure $\langle I, O, Q, q^0, \delta, \lambda \rangle$, with $\delta : Q \times I \rightarrow Q$, $\lambda : Q \times I \rightarrow O$, and $q \xrightarrow{i/o} q' \Rightarrow \delta(q, i) = q' \wedge \lambda(q, i) = o$.

MAT framework The most efficient algorithms for model learning all follow the pattern of a *minimally adequate teacher (MAT)* as proposed by Angluin [6]. In the MAT framework, learning is viewed as a game in which a learner has to infer an unknown automaton by asking queries to a teacher. The teacher knows the automaton, which in our setting is a deterministic Mealy machine $\mathcal{M}$. Initially, the learner only knows the inputs I and outputs O of $\mathcal{M}$. The task of the learner is to learn $\mathcal{M}$ through two types of queries:

- With a *membership query*, the learner asks what the response is to an input sequence $\sigma \in I^*$. The teacher answers with the output sequence in $A_{\mathcal{M}}(\sigma)$.
- With an *equivalence query*, the learner asks whether a hypothesized Mealy machine $\mathcal{H}$ is correct, that is, whether $\mathcal{H} \approx \mathcal{M}$. The teacher answers *yes* if this is the case. Otherwise it answers *no* and supplies a *counterexample*, which is a sequence $\sigma \in I^*$ that triggers a different output sequence for both Mealy machines, that is, $A_{\mathcal{H}}(\sigma) \neq A_{\mathcal{M}}(\sigma)$.

Starting from Angluin's seminal L^* algorithm [6], many algorithms have been proposed for learning finite, deterministic Mealy machines via a finite number of queries. We refer to [22] for recent overview. In applications in which one wants to learn a model of a black-box reactive system, the teacher typically consists of a System Under Learning (SUL) that answers the membership queries, and a conformance testing tool [26] that approximates the equivalence queries using a set of *test queries*. A test query consists of asking to the SUL for the response to an input sequence $\sigma \in I^*$, similar to a membership query.

Abstraction We recall relevant parts of the theory of abstractions from [2]. Existing model learning algorithms are only effective when applied to Mealy machines with small sets of inputs, e.g. fewer than 100 elements. Practical systems like TCP, however, typically have huge alphabets, since inputs and outputs carry parameters of type integer or string. In order to learn an over-approximation of a "large" Mealy machine $\mathcal{M}$, we place a transducer in between the teacher and the learner, which translates concrete inputs in I to abstract inputs in X, concrete outputs in O to abstract outputs in Y, and vice versa. This allows us to abstract a Mealy machine with concrete symbols in I and O to a Mealy machine with abstract symbols in X and Y, reducing the task of the learner to inferring a "small" abstract Mealy machine.

Formally, a *mapper* for inputs I and outputs O is a deterministic Mealy machine $\mathcal{A} = \langle I \cup O, X \cup Y, R, r_0, \delta, \lambda \rangle$, where I and O are disjoint sets of *concrete input and output symbols*, X and Y are disjoint sets of *abstract input and output symbols*, and $\lambda : R \times (I \cup O) \to (X \cup Y)$, the *abstraction function*, respects inputs and outputs, that is, for all $a \in I \cup O$ and $r \in R$, $a \in I \Leftrightarrow \lambda(r, a) \in X$.

Basically, the *abstraction* of Mealy machine $\mathcal{M}$ via mapper $\mathcal{A}$ is the Cartesian product of the underlying transition systems. Formally, let $\mathcal{M} = \langle I, O, Q, q_0, \to \rangle$ be a Mealy machine and let $\mathcal{A} = \langle I \cup O, X \cup Y, R, r_0, \delta, \lambda \rangle$ be a mapper. Then $\alpha_{\mathcal{A}}(\mathcal{M})$, the *abstraction of* $\mathcal{M}$ *via* $\mathcal{A}$, is the Mealy machine $\langle X, Y \cup \{\bot\}, Q \times R, (q_0, r_0), \to \rangle$, where $\bot \notin Y$ is a fresh output and $\to$ is given by the rules

$$\frac{q \xrightarrow{i/o} q',\ r \xrightarrow{i/x} r' \xrightarrow{o/y} r''}{(q,r) \xrightarrow{x/y} (q', r'')} \qquad \frac{\nexists i \in I : r \xrightarrow{i/x}}{(q,r) \xrightarrow{x/\bot} (q,r)}$$

To understand how the mapper is utilized during learning, we follow the execution of a single input of a query. The learner produces an abstract input x,

which it sends to the mapper. By inversely following abstraction function λ, the mapper converts this to a concrete input i and updates its state via transition $r \xrightarrow{i/x} r'$. The concrete input x is passed on to the teacher, which responds with a concrete output o. This triggers the transition $r' \xrightarrow{o/y} r''$ in which the mapper generates the corresponding abstract output y and updates its state again. The abstract output is then returned to the learner.

We notice that the abstraction function is utilized invertedly when translating inputs. More precisely, the abstract input that the learner provides is an output for the mapper. The translation from abstract to concrete involves picking an arbitrary concrete value that corresponds with the given abstract value. It could be that multiple concrete values can be picked, in which case all values should lead to the same abstract behavior in order to learn a deterministic abstract model. It can also be that no values correspond to the input abstraction, in which case, by the second rule, $\perp$ is returned to the learner, without consulting the teacher. We define the *abstraction component* implementing $\alpha_{\mathcal{A}}$ as the transducer which follows from the mapper $\mathcal{A}$, but inverts the abstraction of inputs.

From the perspective of a learner, a teacher for $\mathcal{M}$ and abstraction component implementing $\alpha_{\mathcal{A}}$ together behave exactly like a teacher for $\alpha_{\mathcal{A}}(\mathcal{M})$. If $\alpha_{\mathcal{A}}(\mathcal{M})$ is deterministic, then the learner will eventually succeed in learning a deterministic machine $\mathcal{H}$ satisfying $\alpha_{\mathcal{A}}(\mathcal{M}) \approx \mathcal{H}$. In [2], also a *concretization* operator $\gamma_{\mathcal{A}}$ is defined. This operator is the adjoint of the abstraction operator: it turns any abstract machine $\mathcal{H}$ with symbols in X and Y into a concrete machine with symbols in I and O. If $\mathcal{H}$ is deterministic then $\gamma_{\mathcal{A}}(\mathcal{H})$ is weakly deterministic.

As shown in [2], $\alpha_{\mathcal{A}}(\mathcal{M}) \leq \mathcal{H}$ implies $\mathcal{M} \leq \gamma_{\mathcal{A}}(\mathcal{H})$. This tells us that when we apply mapper $\mathcal{A}$ during learning of some "large" Mealy machine $\mathcal{M}$, even though we may not be able to learn the behavior of $\mathcal{M}$ exactly, the concretization $\gamma_{\mathcal{A}}(\mathcal{H})$ of the learned abstract model $\mathcal{H}$ is an over-approximation of $\mathcal{M}$, that is, $\mathcal{M} \leq \gamma_{\mathcal{A}}(\mathcal{H})$. Similarly to the abstraction component, a *concretization component* for mapper $\mathcal{A}$ implements $\gamma_{\mathcal{A}}$. This component is again fully defined by a mapper, but handles abstraction of outputs invertedly. During model checking, the composition of the abstract model $\mathcal{H}$ and the concretization component for $\mathcal{A}$ provides us with an over-approximation of $\mathcal{M}$.

Framework for mapper definition In order to apply our abstraction approach, we need an abstraction and a concretization component for a given mapper $\mathcal{A}$. We could implement these components seperately in an arbitrary programming language, but then they would have to remain consistent with $\mathcal{A}$. Moreover, ensuring that translation in one component inverts the corresponding translation in the other is non-trivial, and difficult to maintain, as changes in one would have to be applied invertedly in the other.

We used an alternative approach, in which we first define a mapper and then derive the abstraction and concretization components automatically. To this

end, we built a language for defining a mapper in terms of (finite) registers, and functions to encode transitions and outputs. Our language supports case distinctions with programming-style if-else-statements, and requires that every branch leads to exactly one output and updates registers exactly once, such that the translations are complete. Listing 1 shows the example of a mapper for a simple login system. The mapper stores the first password received, and compares subsequent passwords to it. The abstract passwords used by the learner are $\{true, false\}$, denoting a correct or incorrect password, respectively. At the first attempt, *true* invertedly maps to any concrete password, and *false* maps to $\bot$. Later on *true* invertedly maps to the value picked the first time, while *false* maps to any other value. For TCP, we define multiple abstraction functions for inputs and outputs, in terms of multiple parameters per input or output.

Listing 1 A simple example mapper for a login system, in a simplified syntax

```
integer stored := −1;
map ENTER(integer password → boolean correct)
    if (stored = −1 ∧ password ≥ 0) ∨ stored = password then
        correct := true
    else
        correct := false
    end if
end map
update
    if stored = −1 ∧ password ≥ 0 then
        stored := password
    else
        stored := stored                    ▷ Every path explicitly assigns a value
    end if
end update
```

To derive the components, we need to implement the inverse of the abstraction function, for both inputs and outputs. This can be achieved using a constraint solver or by just trying out random concrete values and checking if they are translated to the sought after abstraction. In addition to finding a corresponding concrete value, another purpose of executing abstractions invertedly is to test the abstraction: different possible values should lead to the same abstract behavior, as the learner cannot handle non-determinism. A constraint solver usually picks values in a very structured and deterministic way, which does not test the abstraction well. Picking concrete values manually allows more control over obtaining a good test coverage, but is in general less scalable.

3 Learning setup

3.1 TCP as a system under learning

In TCP there are two interacting entities, the *server* and the *client*, which communicate over a network through packets, comprising a header and application data. On both sides there is an application, initiating and using the connection through *socket calls*. Each entity is learned seperately and is a SUL in the learning context. This SUL thus takes packets or socket calls as inputs. It can output packets or *timeout*, in case the system does not respond with any packet. RFC 793 [35] and its extensions, most notably [9,32], specify the protocol.

Packets are defined as tuples, comprising sequence and acknowledgement numbers, a payload and flags. By means of abstraction, we reduce the size of sequence and acknowledgement number spaces. Each socket call also defines an abstract and concrete input. Whereas packet configurations are the same for both client and server, socket calls differ. The server can *listen* for connections and *accept* them, whereas the client can actively *connect*. Both parties can *send* and *receive* data, or *close* an established connection (specifically, a half-duplex close [9, p. 88]). The server can additionally *close* its listening socket. Values returned by socket calls are not in the output alphabet to reduce setup complexity.

Figure 1 displays the learning setup used. The *learner* generates abstract inputs, representing packets or socket calls. The abstraction component concretizes each input by translating abstract parameters to concrete, and then updates its state. The concrete inputs are then passed on to the *network adapter*, which in turn transforms each input into a packet, sending it directly to the SUL, or into a socket call, which it issues to the SUL *adapter*. The SUL adapter runs on the same environment as the SUL and its sole role is to perform socket calls on the SUL. Each reponse packet generated by the SUL is received by the network adapter, which retrieves the concrete output from the packet or produces a *timeout* output, in case no packet was received within a predefined time interval. The output is then sent to the abstraction component, which computes the abstraction, updates its state again, and sends the abstract output to the learner.

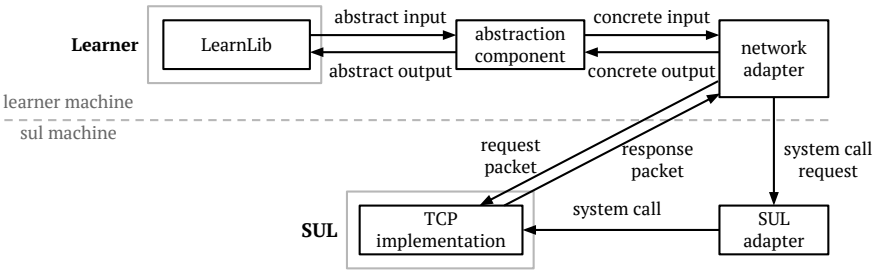

Fig. 1. Overview of the learning setup.

The learner is based on LearnLib [36], a Java library implementing L^* based algorithms for learning Mealy machines. The abstraction component is also written in Java, and interprets and inverts a mapper. The network adapter is a Python program based on Scapy [38], Pcapy [33], and Impacket [21]. It uses Scapy to craft TCP packets, and Scapy together with a Pcapy and Impacket based sniffer to intercept responses. The network adapter is connected to the SUL adapter via a standard TCP connection. This connection is used for communicating socket calls to be made on the SUL. Finally, the SUL adapter is a program which performs socket calls on command, written in C to have low level access to socket options. where this option is applicable as it helps unearth more behavior than the corresponding non-blocking calls.

3.2 Viewing the SUL as a Mealy machine

TCP implementations cannot be fully captured by Mealy Machines. To learn a model, we therefore need to apply some restrictions. As mentioned, the number of possible values for the sequence and acknowledgement numbers is reduced by means of abstractions. Furthermore, payload is limited to either 0 or 1 byte. Consequently, 1 byte of data is sent upon a *send*-call. Flags are limited to only the most interesting combinations, and we also abstract away from all other fields from the TCP layer or lower layers, allowing Scapy to pick default values.

TCP is also time-dependent. The SUL may, for instance, retransmit packets if they are not acknowledged within a specified time. The SUL may also reset if it does not receive the acknowledgement after a number of such retransmissions, or if it remains in certain states for too long. The former we handled by having the network adapter ignore all retransmissions. For the latter, we verified that the learning queries learning were short enough so as not to cause these resets.

TCP is inherently concurrent, as a server can handle multiple connections at any time. This property is difficult to capture in Mealy Machines. Overcoming this, the SUL adapter ensures that at most one connection is accepted at any time by using a set of variables for locking and unlocking the *accept* and *connect*-calls. Only one blocking socket call can be pending at any time, but non-blocking socket calls can always be called.

Furthermore, the backlog size parameter defines the number of connections to be queued up for an eventual *accept*-call by the server SUL. The model grows linearly with the this parameter, while only exposing repetitive behavior. For this reason we set the backlog to the value 1.

3.3 Technical challenges

We overcame several technical challenges in order to learn models. Resetting the SUL and setting a proper timeout value are solved similarly to [17].

Our tooling for sniffing packets sometimes missed packets generated by the SUL, reporting erroneous *timeout* outputs. This induced non-deterministic behavior, as a packet may or may not be caught, depending on timing. Each observation is therefore repeated three times to ensure consistency. Consistent outputs are cached to speed up learning, and to check consistency with new observations. It also allows to restart learning with reuse of previous observations.

In order to remove time-dependent behavior we use several TCP settings. Most notably, we disable slow acknowledgements and enable quick acknowledgements where possible (on Linux and FreeBSD). The intuition is that we want the SUL to send acknowledgements whenever they can be issued, instead of delaying them. We also had to disable syn cookies in FreeBSD, as this option caused generation of the initial sequence number in a seemingly time dependent way, instead of using fresh values. For Linux, packets generated by a *send*-call were occasionaly merged with previous unacknowledged packets, so we could only learn a model by omitting *send*.

3.4 Mapper definition

The mapper is based on the work of Aarts et al. [5], and on the RFCs. Socket calls contain no parameters and do not need abstraction, so they are mapped simply with the identity relation. TCP packets are mapped by mapping their parameters individually. Flags are again retained by an identity relation. The sequence and acknowledgement numbers are mapped differently for inputs and outputs; input numbers are mapped to $\{valid, invalid\}$, and outputs are mapped to $\{current, next, zero, fresh\}$. After a connection is set up, the mapper keeps track of the sequence number which should be sent by the SUL and learner. Valid inputs are picked according to this, whereas *current* and *next* represent repeated or incremented numbers, respectively. The abstract output *zero* simply denotes the concrete number zero, whereas *fresh* is used for all other numbers. If no connection is established, any sequence number is valid (as the RFCs then allow a fresh value), and the only valid acknowledgement number is zero.

Note that all concrete inputs with the same abstract value, should lead to an equivalent abstract behavior. Valid inputs are defined according to the RFC's. However, this is very hard for invalid inputs, as they may be accepted, ignored, they may lead to error recovery, or even undefined behavior. To learn the behavior for these inputs, abstractions should be defined precisely according to these behaviors, which is unfeasible to do by hand. As a result, we have excluded invalid inputs from the learning alphabet. To translate valid inputs, we first used a constraint solver which finds solutions for the transition relation. This is done by taking the disjunction of all path constraints, similar to symbolic execution techniques [24]. However, this did not test the abstraction well, as the constraint solver always picks zero if possible, for example. We therefore manually picked random concrete values to test with, instead. Values were picked with a higher probability if they were picked or observed previously during the same run, as

well as the successors of those values. This approach sufficed to translate all values for our experiments.

4 Model learning results

Using the abstractions defined in Section 2, we learned models of the TCP client and server for Windows 8, Ubuntu 14.04 and FreeBSD 10.2. For testing we used the conformance testing algorithm described in [39] to generate efficient test suites which are parameterized by a middle section of length k. Generated exhaustively, these ensure learned model correctness, unless the respective implementation corresponds to a model with at least k more states. For each model, we first executed a random test suite with k of 4, up to 40000 tests for servers, and 20000 tests for clients. We then ran an exhaustive test suite with k of 2 for servers, respectively 3 for clients.

Table 1 describes the setting of each of these experiments together with statistics on learning and testing: (1) the number of states in the final model, (2) the number of hypotheses found, (3) the total number of membership queries, (4) the total number of unique test queries run on the SUL before the last hypothesis, (5) the number of unique test queries run to validate the last hypothesis.

SUL		States	Hyp.	Memb. Queries	Tests to last Hyp.	Tests on last Hyp.
Client	Windows 8	13	2	1576	1322	50243
Server	Windows 8	38	10	11428	9549	65040
Client	Ubuntu 14.04	15	2	1974	15268	56174
Server	Ubuntu 14.04	57	14	17879	15681	66523
Client	FreeBSD 10.2	12	2	1456	1964	47387
Server	FreeBSD 10.2	55	18	22287	12084	75894

Table 1. Statistics for learning experiments

Figure 2 shows the model learned for the Windows 8 client. This model covers standard client behavior, namely connection setup, sending and receiving data and connection termination. Based on predefined access sequences, we identify each state with its analogous state in the RFC state diagram [35, p. 23], if such a state exists. Transitions taken during simulated communication between a Windows client and a server are colored green. These transitions were identified during model checking, on which we expand in Section 5.

Table 1 shows that the models for the Linux and FreeBSD servers have more states than for Windows, and all models have more states than described in the specification. We attribute this to several factors. We have already mentioned that model sizes grow linearly with the value of the backlog-parameter. While we set it to 1, the setting is overriden by operating system imposed minimum value of 2 for FreeBSD and Linux. Moreover, SUL behavior depends on blocking system calls and on whether the receive buffer is empty or not. Although specified, this

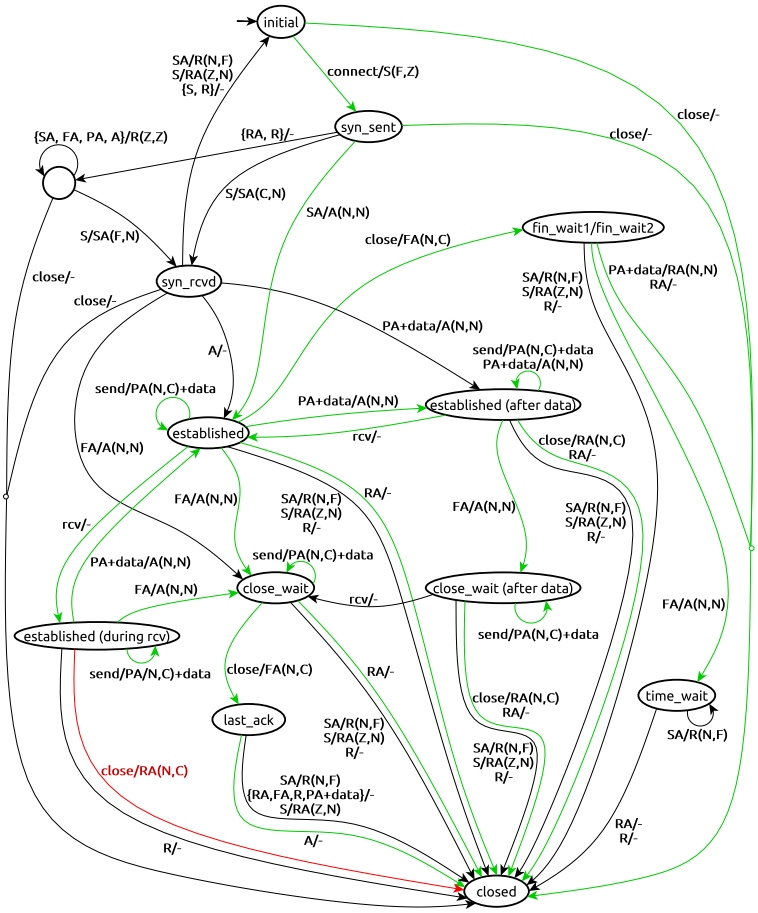

Fig. 2. Learned model for Windows 8 TCP Client. To reduce the size of the diagram, we eliminate all self loops with *timeout* outputs. We replace flags and abstractions by their capitalized initial letter, hence use *s* for *syn*, *a* for *ack*, *n* for *next*, etc. We omit input parameter abstractions, since they are the same for all packets, namely *valid* for both sequence and acknoweldgement numbers. Finally, we group inputs that trigger a transition to the same state with the same output. Timeouts are denoted by '-'.

is not modelled explicitly in the specification state diagram. As an example, the ESTABLISHED and CLOSE WAIT states from the standard each have multiple corresponding states in the model in Figure 2.

Non-conformance of implementations Inspection of learned models revealed several cases of non-conformance to RFC's in the corresponding implementations.

A first non-conformance involves terminating an established connection with a CLOSE. The resulting output should contain a FIN if all data up to that point has been received. If there is data not yet received, the output should contain a RST, which would signal to the other side an aborted termination [9, p. 88]. Windows does not conform to this, as a CLOSE can generate a RST instead of a FIN even in cases where there is no data to be received, namely, in states where a RCV call is pending. Figure 2 marks this behavior in red. FreeBSD implementations are also non-compliant, as they always generate FIN packets on a CLOSE, regardless if all data has been received. This would arguably fall under the list of common bugs [32], namely "Failure to send a RST after Half Duplex Close". The learned Linux models fully comply to these specifications.

A second non-conformance has to do with the processing of SYN packets. On receiving a SYN packet in a synchronized state, if the sequence number is in "the window" (as it always is, in our case), the connection should be reset (via a corresponding RST packet) [35, p.71]. Linux implementations conform for SYN packets but not for SYN+ACK packets, to which they respond by generating an acknowledgement with no change of state. Both Windows and FreeBSD respect this specification.

We note a final non-conformance in Windows implementations. In case the connection does not exist (CLOSED), a reset should be sent in response to any incoming packet except for another reset [35, p. 36], but Windows 8 sends nothing. FreeBSD can be configured to respond in a similar way to Windows, by changing the *blackhole setting*.[1] This behavior is claimed to provide "some degree of protention against stealth scans", and is thus intentional.

5 Model checking results

5.1 Model checking the learned behaviour

We analyzed the learned models of TCP implementations using the model checker NuSMV [14]. We composed pairs of learned client and server models with a handmade model of a non-lossy network, which simply delivers output from one entity as input for the other entity. Since the abstract input and output domains are different, the abstract models cannot communicate directly, and so we had to encode the concretized models within NuSMV code. We wrote a script that translated the abstract Mealy machine models from LearnLib to NuSMV modules, and another script that translated the corresponding mappers to NuSMV

[1] `https://www.freebsd.org/cgi/man.cgi?query=blackhole`

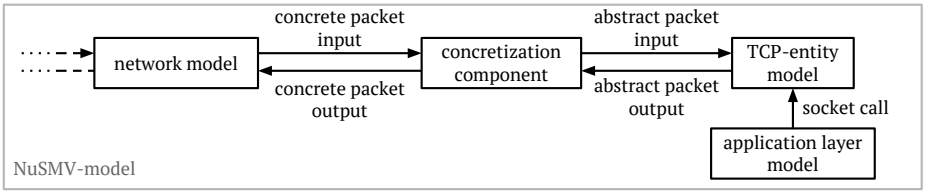

Fig. 3. Schematic overview of NuSMV-model. Only half of the setup is shown in detail, as the model is symmetric and another TCP-entity model is connected to the network.

modules. TCP entities produce abstract outputs, which are translated to concrete. The network module then passes along such concrete messages. Before being delivered to the other entity, these messages are again transformed into abstract inputs. By encoding mapper functions as relations, NuSMV is able to compute both the abstraction function and its inverse, i.e., act as a concretization component. The global structure of the model is displayed in Figure 3.

In Mealy machines, transitions are labeled by an input/output pair. In NuSMV transitions carry no labels, and we also had to split the Mealy machine transitions into a separate input and output part in order to enable synchronization with the network. Thus, a single transition $q \xrightarrow{i/o} q'$ from a (concrete) Mealy machine is translated to a pair of transitions in NuSMV:

$$(loc = q, in = .., out = ..) \rightarrow (loc = q, in = i, out = ..) \rightarrow (loc = q', in = i, out = o).$$

Sequence and acknowledgement numbers in the implementations are 32-bit numbers, but were restricted to 3-bit numbers to reduce the state space. Whereas concrete messages are exchanged from one entity to the other, socket call inputs from the application are simulated by allowing system-calls to occur nondeterministically. A simplification we make is that we do not allow parallel actions: an action and all resulting packets have to be fully processed until another action can be generated. Consequently, there can be at most one packet in the composite model at any time. For example, once a three way handshake is initiated between a client and a listening server via a *connect*-call, no more system-calls can be performed until the handshake is finalized.

5.2 Checking specifications

After a model is composed, the interaction between TCP entities can be analyzed using the NuSMV model checker. However, it is important to realize that, since we used abstractions, the learned models of TCP servers and clients are over-approximations of the actual behaviors of these components. If Mealy machine $\mathcal{M}$ models the actual behavior of a component, $\mathcal{A}$ is the mapper used, and

$\mathcal{H}$ is the abstract model that we learned then, as explained in Section 2, correctness of $\mathcal{H}$ implies $\mathcal{M} \leq \gamma_{\mathcal{A}}(\mathcal{H})$. Since $\gamma_{\mathcal{A}}(\mathcal{H})$ is weakly deterministic, in this case there exists a forward simulation relation from $\mathcal{M}$ to $\gamma_{\mathcal{A}}(\mathcal{H})$. This forward simulation is preserved by the translation from Mealy machines to NuSMV. Results from Grumberg and Long [19] then imply that, for any $\forall$CTL*-formula (which includes all LTL-formulas) we can transfer model checking results for $\gamma_{\mathcal{A}}(\mathcal{H})$ to the (unknown) model $\mathcal{M}$. Since simulations are preserved by composition, this result even holds when $\gamma_{\mathcal{A}}(\mathcal{H})$ is used as a component in a larger model.

Another essential point is that only a subset of the abstract inputs is used for learning. Hence invalid inputs (i.e. inputs with invalid parameters) are not included in our models. Traces with these inputs can therefore not be checked. Hence, the first property that we must check is a global invariant that asserts that invalid inputs will never occur. In case they do, NuSMV will provide a counterexample, which is used to find the cause of invalidity. During our initial experiments, NuSMV found several counterexamples showing that invalid inputs may occur. Based on analysis of these counterexamples we either refined/corrected the definition of one of the mappers, or we discovered a counterexample for the correctness of one of the abstract models. After a number of these iterations, we obtained a model in which invalid inputs can no longer occur.

With only valid inputs, composite model may be checked for arbitrary $\forall$CTL* formulas. Within these formulas, we may refer to input and output packets and their constituents (sequence numbers, acknowledgements, flags,..). This yields a powerful language for stating properties, illustrated by a few examples below. These formulas are directy based on the RFC's.

Many properties that are stated informally in the RFC's refer to control states of the protocol. These control states, however, cannot be directly observed in our black-box setting. Nevertheless, we can identify states, e.g. based on inputs and outputs leading to and from it. For example, we base the proposition *established* on RFC 793, which states that: "The connection becomes 'established' when sequence numbers have been synchronized in both directions" [35, p. 11], and that only a CLOSE or ABORT socket call or incoming packets with a RST or FIN can make an entity leave the ESTABLISHED state [35, section 3.9].

We first show a simple safety formula checking desynchonization: if one entity is in the ESTABLISHED state, the other cannot be in SYN_SENT and TIME_WAIT:

$$\mathsf{G}\neg(\mathit{tcp1\text{-}state} = \mathit{established} \wedge (\mathit{tcp2\text{-}state} = \mathit{syn\_sent} \vee \mathit{tcp2\text{-}state} = \mathit{time\_wait}))$$

The next specification considers terminating an established connection with a CLOSE-input. The output should contain a FIN, except if there is unread data (in which case it should contain a RST). This corresponds to the first non-conformance case explained in Section 4. The specification is captured by the following formula, in which T is the triggered-operator as defined in NuSMV.

$$\mathsf{G}(\mathit{state} = \mathit{established} \rightarrow ((\mathit{input} = \mathit{rcv}\ \mathsf{T}\ \mathit{input} \neq \mathit{packet\ with\ data}) \wedge \mathit{input} = \mathit{close}) \rightarrow (\mathsf{F}\ \mathit{output} = \mathit{packet\ with}\ \textsc{fin})))$$

We have formalized and checked, in a similar way, specifications for all other non-conforming cases as well as many other specifications.

We have also checked which transitions in the abstract models are reachable in the composed system. For every transition, we take its input and starting state, and check whether they can occur together. In this way we can find the reachable parts of model. This proves useful when analysing models, as the reachable parts likely harbor bugs with the most impact. Similarly, comparing reachable parts helps reveal the most relevant differences between implementations. The first and third non-conformances in Section 4 occur in the reachable parts of the respective models. Figure 2 marks these parts in green.

6 Conclusions and future work

We combined automata learning, model checking and abstraction techniques to obtain and analyze models of Windows, Linux and FreeBSD TCP server and client implementations. Composing these models together with the model of a network allowed us to perform model checking over the composite setup and verify that any valid number generated by one TCP entity is seen as valid number by the other TCP entity. We have also identified breaches of the RFC's in all operating systems. Our work suggests several directions for future work.

Based on our understanding of TCP, we manually defined abstractions (mappers) that made it possible to learn models of TCP implementations. Getting the mapper definitions right turned out to be tricky. In fact, we had to restrict our learning experiments to *valid* abstractions of the sequence and acknowledgement numbers. This proved limiting when searching for interesting rules to model check, like for example those that would expose known implementation bugs. Such rules often concern invalid parameters, which do not appear in the models we learned. Learning algorithms that construct the abstractions automatically could potentially solve this problem. We hope that extensions of the learning algorithms for register automata as implemented in the Tomte [4] and RALib [12] tools will be able to construct abstractions for TCP fully automatically.

Our work was severely restricted by the lack of expressivity of Mealy machines. In order to squeeze the TCP implementation into a Mealy machine, we had to eliminate timing based behavior as well as re-transmissions. Other frameworks for modeling state machines might facilitate modelling these aspects. Obviously, we would also need learning algorithms capable of generating such state machines. There has been some preliminary work on extening learning algorithms to I/O transition systems [3,44] and to timed automata [18,43]. Extensions of this work could eliminate some of the restrictions that we encountered.

References

1. F. Aarts. *Tomte: Bridging the Gap between Active Learning and Real-World Systems.* PhD thesis, Radboud University Nijmegen, October 2014.
2. F. Aarts, B. Jonsson, J. Uijen, and F.W. Vaandrager. Generating models of infinite-state communication protocols using regular inference with abstraction. *Formal Methods in System Design*, 46(1):1–41, 2015.
3. F. Aarts and F.W. Vaandrager. Learning I/O automata. In P. Gastin and F. Laroussinie, editors, *21st International Conference on Concurrency Theory (CONCUR), Paris, France, August 31st - September 3rd, 2010, Proceedings*, volume 6269 of *Lecture Notes in Computer Science*, pages 71–85. Springer, 2010.
4. Fides Aarts, Paul Fiterău-Broştean, Harco Kuppens, and Frits W. Vaandrager. Learning register automata with fresh value generation. In Martin Leucker, Camilo Rueda, and Frank D. Valencia, editors, *Theoretical Aspects of Computing - ICTAC 2015 - 12th International Colloquium Cali, Colombia, October 29-31, 2015, Proceedings*, volume 9399 of *Lecture Notes in Computer Science*, pages 165–183. Springer, 2015.
5. Fides Aarts, Bengt Jonsson, Johan Uijen, and Frits W. Vaandrager. Generating models of infinite-state communication protocols using regular inference with abstraction. *Formal Methods in System Design*, 46(1):1–41, 2015.
6. D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.
7. C. Baier and J.-P. Katoen. *Principles of Model Checking.* MIT Press, Cambridge, Massachusetts, 2008.
8. J. Berendsen, B. Gebremichael, F.W. Vaandrager, and M. Zhang. Formal specification and analysis of Zeroconf using Uppaal. *ACM Transactions on Embedded Computing Systems*, 10(3), April 2011.
9. R. Braden. *RFC 1122 Requirements for Internet Hosts - Communication Layers.* Internet Engineering Task Force, October 1989.
10. E. Brinksma and A. Mader. On verification modelling of embedded systems. Technical Report TR-CTIT-04-03, Centre for Telematics and Information Technology, Univ. of Twente, The Netherlands, January 2004.
11. G. Bruns and M.G. Staskauskas. Applying formal methods to a protocol standard and its implementations. In *Proceedings International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE 1998),* 20-21 April, 1998, Kyoto, Japan, pages 198–205. IEEE Computer Society, 1998.
12. S. Cassel. *Learning Component Behavior from Tests: Theory and Algorithms for Automata with Data.* PhD thesis, University of Uppsala, 2015.
13. G. Chalupar, S. Peherstorfer, E. Poll, and J. de Ruiter. Automated reverse engineering using Lego. In *Proceedings 8th USENIX Workshop on Offensive Technologies (WOOT'14),* San Diego, California, Los Alamitos, CA, USA, August 2014. IEEE Computer Society.
14. Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *Computer Aided Verification*, pages 359–364. Springer, 2002.
15. E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.
16. E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, Cambridge, Massachusetts, 1999.

17. P. Fiterău-Broştean, R. Janssen, and F.W. Vaandrager. Learning fragments of the TCP network protocol. In Frédéric Lang and Francesco Flammini, editors, *Proceedings 19th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'14),* Florence, Italy, volume 8718 of *Lecture Notes in Computer Science*, pages 78–93. Springer, September 2014.
18. Olga Grinchtein, Bengt Jonsson, and Martin Leucker. Learning of event-recording automata. *Theor. Comput. Sci.*, 411(47):4029–4054, 2010.
19. O. Grumberg and D.E. Long. Model checking and modular verification. *ACM Trans. Program. Lang. Syst.*, 16(3):843–871, 1994.
20. G.J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison Wesley, 2004.
21. Impacket. `http://www.coresecurity.com/corelabs-research/open-source-tools/impacket`. Accessed: 2016-01-28.
22. Malte Isberner. *Foundations of Active Automata Learning: An Algorithmic Perspective*. PhD thesis, Technical University of Dortmund, 2015.
23. Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Comput. Surv.*, 41(4):21:1–21:54, October 2009.
24. James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.
25. I. van Langevelde, J.M.T. Romijn, and N. Goga. Founding FireWire bridges through Promela prototyping. In 8^{th} *International Workshop on Formal Methods for Parallel Programming: Theory and Applications (FMPPTA)*. IEEE Computer Society Press, April 2003.
26. D. Lee and M. Yannakakis. Principles and methods of testing finite state machines — a survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.
27. Lars Lockefeer, David M. Williams, and Wan J. Fokkink. Formal specification and verification of TCP extended with the window scale option. In Frédéric Lang and Francesco Flammini, editors, *Formal Methods for Industrial Critical Systems - 19th International Conference, FMICS 2014, Florence, Italy, September 11-12, 2014. Proceedings*, volume 8718 of *Lecture Notes in Computer Science*, pages 63–77. Springer, 2014.
28. C. Loiseaux, S. Graf, J. Sifakis, A. Boujjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6(1):11–44, 1995.
29. Karl Meinke and Muddassar A Sindhu. Incremental learning-based testing for reactive systems. In M. Gogolla and B. Wolff, editors, *Tests and Proofs*, pages 134–151. Springer, 2011.
30. Madanlal Musuvathi and Dawson R. Engler. Model checking large network protocol implementations. In Robert Morris and Stefan Savage, editors, *1st Symposium on Networked Systems Design and Implementation (NSDI 2004), March 29-31, 2004, San Francisco, California, USA, Proceedings*, pages 155–168. USENIX, 2004.
31. Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: A pragmatic approach to model checking real code. In David E. Culler and Peter Druschel, editors, *5th Symposium on Operating System Design and Implementation (OSDI 2002), Boston, Massachusetts, USA, December 9-11, 2002*. USENIX Association, 2002.
32. V. Paxson, M. Allman, S. Dawson, W. Fenner, J. Griner, I. Heavens, K. Lahey, J. Semke, and B. Volz. Known TCP Implementation Problems. RFC 2525 (Informational), March 1999.
33. Pcapy. `http://www.coresecurity.com/corelabs-research/open-source-tools/pcapy`. Accessed: 2016-01-28.

34. D. Peled, M.Y. Vardi, and M. Yannakakis. Black Box Checking. *Journal of Automata, Languages, and Combinatorics*, 7(2):225–246, 2002.
35. J. Postel. Transmission control protocol. RFC 793 (Standard), sep 1981. Updated by RFCs 1122, 3168.
36. H. Raffelt, B. Steffen, T. Berg, and T. Margaria. LearnLib: a framework for extrapolating behavioral models. *STTT*, 11(5):393–407, 2009.
37. Joeri de Ruiter and Erik Poll. Protocol state fuzzing of tls implementations. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 193–206, Washington, D.C., August 2015. USENIX Association.
38. Scapy. `http://www.secdev.org/projects/scapy/`. Accessed: 2016-01-28.
39. W. Smeenk, J. Moerman, D.N. Jansen, and F.W. Vaandrager. Applying automata learning to embedded control software. In M. Butler, S. Conchon, and F. Zaidi, editors, *Proceedings 17th International Conference on Formal Engineering Methods (ICFEM 2015), Paris, 3-6 November 2015*, volume 9407 of *Lecture Notes in Computer Science*, pages 1–17. Springer-Verlag, 2015.
40. B. Steffen, F. Howar, and M. Merten. Introduction to active automata learning from a practical perspective. In M. Bernardo and V. Issarny, editors, *Formal Methods for Eternal Networked Software Systems - 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM 2011, Bertinoro, Italy, June 13-18, 2011. Advanced Lectures*, volume 6659 of *Lecture Notes in Computer Science*, pages 256–296. Springer, 2011.
41. M. Stoelinga. Fun with FireWire: A comparative study of formal verification methods applied to the IEEE 1394 root contention protocol. *Formal Aspects of Computing Journal*, 14(3):328–337, 2003.
42. M. Tijssen. *Automatic Modeling of SSH Implementations with State Machine Learning Algorithms*. Bachelor thesis, ICIS, Radboud University Nijmegen, June 2014.
43. S. Verwer. *Efficient Identification of Timed Automata — Theory and Practice*. PhD thesis, Delft University of Technology, March 2010.
44. Michele Volpato and Jan Tretmans. Active learning of nondeterministic systems from an ioco perspective. In *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, pages 220–235. Springer, 2014.