Learning-Based Testing the Sliding Window Behavior of TCP Implementations

Paul Fiterău-Broştean*[1] and Falk Howar[2]

[1] Institute for Computing and Information Sciences, Radboud University, Nijmegen, the Netherlands
[2] Institute for Applied Software Systems Engineering, Clausthal University of Technology, Clausthal-Zellerfeld, Germany

Abstract. We develop a learning-based testing framework for register automaton models that can express the windowing behavior of TCP, thereby presenting the first significant application of register automata learning to realistic software for a class of automata with Boolean-arithmetic constraints over data values. We have applied our framework to TCP implementations belonging to different operating systems and have found a violation of the TCP specification in Linux and Windows. The violation has been confirmed by Linux developers.

1 Introduction

Automata provide both formal and intuitive means of specifying the behavior for a wide range of applications, in particular network protocols. Unfortunately, protocol specifications often are textual and rarely include state machine models. Without such models, it is difficult to test if an application behaves as expected. Manual construction of models is a laborious and error-prone process and models become outdated as soon as the specification changes. Learning-based testing, as sketched in Figure 1, alleviates this problem by generating models while testing a system. These models cannot serve as specifications but can be used to check desired properties, which are usually easier to formalize and maintain than complete behavioral models.

Integrating model learning, model-based testing, and model checking allows a tester to automatically obtain a model for a system under test. For a set of test inputs, model learning runs a series of tests on the system until, eventually, it will produce a conjectured model of the system's behavior. This model is used as the basis for model-based testing. Testing can discover counterexamples, which indicate incorectness of the model. In such case, model learning is restarted, being provided with the counterexample. Once no counterexample is found, the model can be used for checking properties. The output of learning-based testing is threefold: model learning produces a conformance test suite for the model [4], checking of properties can produce examples that document the violation of a

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

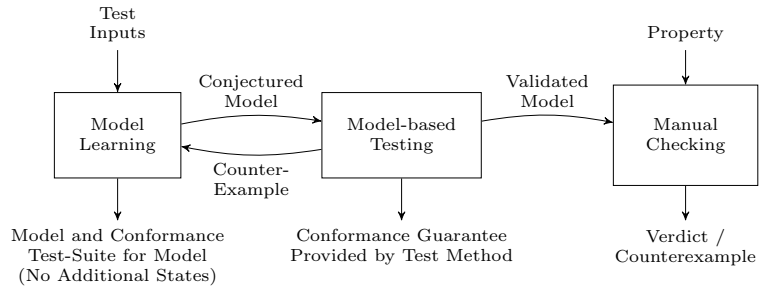

Fig. 1: Learning-based Testing with Additional Checking of Properties.

specification, and in case no violation is found, testing can yields a conformance guarantee.

In order to instantiate learning-based testing for a certain class of models, one needs a learning algorithm and a testing algorithm for this class of models. In this paper, we present a learning-based testing framework for a class of register automata that can express the windowing behavior of TCP. Our framework utilizes the SL^* learning algorithm for register automata [6] and a random walk testing algorithm for such register automaton models. The testing algorithm ensures approximate correctness of models with a high confidence. We manually inspect models and find a violation of the TCP specification in Linux and Windows implementations.

Our work is the first significant application of register automata learning to realistic software for a class of automata with Boolean-arithmetic constraints over data values. Our results show that, on the one hand, learning more expressive models can ease the burden of manually constructed sophisticated test harnesses. On the other hand, experiments show that model learning for more expressive models is very expensive. Future work will focus on scaling learning-based testing to industrial applications as well as on integrating automated model checking into our approach.

Related Work. Learning-based testing in the form that we present here is based on the observation that model learning and model-based testing are merely two sides of the same coin [16]. The term has been introduced in [12] for a combination of model learning, model checking, and random testing. In contrast to our work, the approach is based on finite state models. On the other hand, model checking is automated and feed the model learning algorithm with counterexamples, leading to higher degree of automation.

Learning-based techniques have been steadily gaining traction for more than a decade, after pioneering work on learning and testing CTI systems [10] and learning and checking systems [13]. Previous applications of learning-based testing or checking have lead to the discovery of flaws in TLS implementations [15] and of various forms of specification non-compliance in TCP [7, 8] and SSH [9] implementations. What all these case studies have in common, is the difficulty of manually constructing a sophisticated test harness for the system. This is in large part caused by the need to abstract away from system functionality, so that the functionality seen by the learner fits within the less expressive formal-

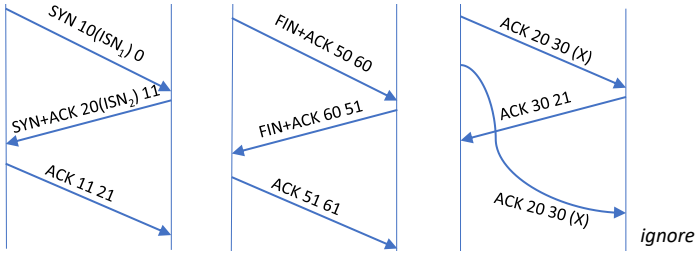

Fig. 2: TCP handshake, connection closure, and data transfer with retransmission. Labels show flags, sequence and acknoweldgement numbers. 1 byte of payload marked by (X). Initial Sequence Numbers marked by (ISN).

ism the learner can infer, typically mealy machines or DFAs. Our learning setup can infer more expressive register automata, and requires no form of abstraction other than a general one for handling fresh values.

Outline. We provide a brief introduction to TCP in the next section before presenting our learning-based testing framework in Section 3. We discuss application of our framework on real TCP implementations in Section 4, before concluding in Section 5.

2 The Sliding Window Behavior of TCP

The Transport Control Protocol (TCP) is a widely used transport layer protocol of the TCP/IP stack, with implementations provided by all operating systems. TCP ensures reliable data transfer between parties. In order to communicate, a TCP client and server application must first establish a TCP connection, which is done by way of a handshake. They can then exchange data over the established connection until one of the parties decides to terminate the connection. A closure procedure ensues, which ultimately removes the connection. In all stages of the protocol, interaction is done by exchanging *TCP segments*. These segments are often the result of calls on the socket interface, which is available to each side and provides access to TCP services. Moreover, each side keeps track of the state of the connection. TCP uses sequence numbers and a sliding receive window to keep track of which segments have been received and acknowledged by the other party. This helps compensate for a potentially lossy communication channel in which reordering of segments can occur (e.g., due to changing routing of segments).

For the sake of exposition, let us assume a setting in which all segments are 1 byte in size. As sequence numbers encode the relative position of a segment in a byte stream, this assumption allows us to confuse the relative position segment in a sequence of segments with its position in a byte stream.

Sequence Numbers. To achieve reliable data transfer, TCP uses sequence and acknowledgement numbers, and flags which are included in the header of all TCP segments. In a stream of segments from a sender to a receiver, the *sequence number* encodes the relative number of a segment in such a stream. The receiver

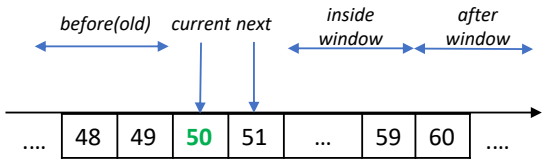

Fig. 3: Relevant Relations of Sequence Numbers in TCP.

acknowledges a received segment by responding with a segment including as *acknowledgement number* the next expected sequence number. Sequence numbers are generated relative to an Initial Sequence Number (ISN), so the first segment has sequence number ISN, the second ISN+1... As data is sent, the sequence number increases, as does the acknowledgement number in responses.

Receive Window. Segments received with a sequence number greater than the one expected fall in two categories: those whose sequence number falls within a *receive window* of that expected and those whose sequence number falls outside of the receive window. The former should be processed by the receiver, the latter should be treated as invalid. As a concrete example, only reset segments (segments with the RST flag enabled) with the sequence number within the receive window are processed, and may reset the connection, those whose number lies outside should be ignored. The receive window is included in the TCP header and its value is communicated in each TCP segment a side sends.

Sliding Windows. Once a received segment is successfully processed, the receive window can be moved forward: if a sequence number of a received segment is equal to the sequence number expected, the expected sequence number is increased. If not equal, the expected sequence number is left unchanged. Acknowledgement numbers are also checked. Those equal to the last sequence number sent acknowledge all segments up to this last one. Those greater are unacceptable as they acknowledge segments not yet sent. Those smaller than the last sequence number sent are old acknowledgements. Segments with unacceptable or old acknowledgement numbers are generally discarded.

As stated above, sequence numbers and receive windows are used, among other things, to deal with reordering of routed segments and to prevent the processing of (bytes in) old segments, which are segments carrying already seen data with sequence numbers smaller than the those expected. Old segments are often the result of re-transmissions, which happen when a timeout for receiving an acknowledgement has expired. TCP is full duplex, which means communicating sides maintain two byte streams, one for each direction. Each side keeps track of the next sequence number to be sent, as well as the sequence number expected from the other side. To open (via handshake), maintain and close the two byte streams, TCP uses control flags. The SYN flag, for example, marks the beginning of a byte stream, whereas the FIN flag marks the end. Figure 2 gives sequence diagrams for typical TCP scenarios.

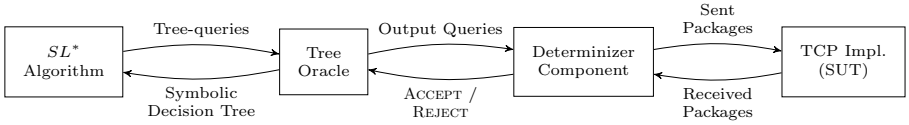

Fig. 4: Learning Register Automaton Models from Tests.

The description so far assumed that all segments were 1 byte in size. In actuality, the size of a segment is the size of the payload carried, plus 1 if either SYN or FIN flags are enabled, or 0 otherwise. We restrict the learning setting to one where segments carry no payload (thus segments are either of size 0 or 1).

Figure 3 depicts the relevant relations sequence numbers may have relative to a current sequence number, in line with our earlier description. These relations are equality and inequality over the current sequence number, and over its summation to one (for segments including either FIN or SYN), and to the receive window size.

3 Instantiating Learning-based Testing for TCP

In order to apply learning-based testing to the windowing behavior of TCP, we instantiate the components of the framework that were sketched in Section 1. We use the SL^* active learning algorithm for learning register automaton models [6]. Active learning algorithms rely on the existence of a minimally adequate teacher (cf. [3]) that answers two kinds of queries for the learning algorithm: *output queries* (i.e., execution of tests) and *equivalence queries*. The learning algorithm submits a conjectured model to an equivalence oracle and expects a counterexample to the model (if one exists). In our scenario, we implement this oracle by performing model-based testing on the model.

The SL^* algorithm additionally assumes the existence of a *tree oracle*. A tree oracle produces register automata fragments that encode the relevant data relations for a sequence of actions on a SUT. The resulting setup is shown in Figure 4. In order to infer symbolic transitions, e.g., for input $ACK(p_1, p_2)$ with two data parameters $p1$ and p_2 from a state that is reached in the protocol by sending a message $SYN(10, 0)$ and receiving message $SYN + ACK(20, 11)$, the SL^* algorithm will perform a tree query for prefix $SYN(10, 0)$ and suffix ACK. The tree oracle will generate output queries for all relevant concrete instances of ACK messages capturing possible relations between values of p_1, p_2 and data values in the prefix (e.g., equality, being a sequence number, or being in a window). The determinizer component will test if output queries are valid traces of a TCP implementation by exchanging actual TCP packages with a system under testing (SUT). The tree oracle encodes the observed behavior and relevant relations as a symbolic decision tree.

In the remainder of this section, we present register automata for the windowing behavior of TCP, tree queries that capture all relevant data relations, and use the presented ideas as a basis for instantiating model-based testing in our framework.

3.1 Register Automata

We assume a set Σ of *actions*, each with an arity that determines how many values from $\mathbb{N}$ it takes as parameters (e.g., ACK takes two data values). To simplify presentation, we assume that all actions have arity 1, but it is straightforward to extend to the case where actions have arbitrary arity. A *data symbol* is a term of form $\alpha(d)$, where α is an action and $d \in \mathbb{N}$ is a data value. A *data word* is a sequence of data symbols. The concatenation of two data words w and w' is denoted ww'. In this context, we often refer to w as a *prefix* and w' as a *suffix*. For a data word $w = \alpha_1(d_1)\ldots\alpha_n(d_n)$, let $Acts(w)$ denote its sequence of actions $\alpha_1\ldots\alpha_n$, and $Vals(w)$ its sequence of data values $d_1\ldots d_n$. Let $|w|$ denote the number of symbols in w.

While there are infinitely many data words for every sequence of actions with data parameters, many of these data words are equivalent when considering only relations between data values (e.g., equality, being a sequence number, or being in a window). For a set of relations $\mathcal{R}$, data words $w = \alpha_1(d_1)\ldots\alpha_n(d_n)$ and $w' = \alpha_1(d'_1)\ldots\alpha_n(d'_n)$ are $\mathcal{R}$-*indistinguishable*, denoted $w \approx_{\mathcal{R}} w'$, if $R(d_{i_1},\ldots,d_{i_j})$ iff $R(d'_{i_1},\ldots,d'_{i_j})$ whenever R is a relation in $\mathcal{R}$ and $i_1,\cdots,i_j$ are indices among $1\ldots n$. We use $[w]_{\mathcal{R}}$ to denote the set of words that are $\mathcal{R}$-indistinguishable from w. A *data language* $\mathcal{L}$ is a set of data words that respects $\mathcal{R}$ in the sense that $w \approx_{\mathcal{R}} w'$ implies $w \in \mathcal{L} \leftrightarrow w' \in \mathcal{L}$.

In order to capture the windowing behavior of TCP, we define the set of relations $\mathcal{R} = \{R_{\otimes,c} \ : \ \otimes \in \{<,\leq,=,\geq,>\} \ \wedge \ c \in \{0,1,100\}\}$, and relation $R_{\otimes,c} \subset \mathbb{N}\times\mathbb{N}$ such that $xR_{\otimes,c}y$ iff $x + c \otimes y$. Relations $R_{\otimes,0}$ encode equality and an order on the sets of sequence numbers. Relations in $R_{\otimes,1}$ encode the successor relation between sequence numbers and $R_{\otimes,100}$ describes windows (of size 100).

We assume a set of *registers* $x_1, x_2, \ldots$ that can store data values of data words. A *parameterized symbol* is a term of form $\alpha(p)$, where α is an action and p a formal parameter. An *atomic guard* g over p is a logic formula of form $(x_i + c \otimes p)$ with $\otimes \in \{<,\leq,=,\geq,>\}$ and $c \in \{0,1,100\}$. We allow for aggregation of atomic guards into *intervals* of form $(g_1 \wedge g_2)$, where atomic guards g_1 and g_2 specify a lower and an upper bound on p, respectively. A valuation $\nu : \{p, x_1, x_2, \ldots\} \mapsto \mathbb{N}$ satisfies a guard g if $g[\nu] \ = \ g[\nu(p)/p][\nu(x_1)/x_1][\ldots]$ is true and we write $\nu \models g$ in this case.

An *assignment* is a simple parallel update of registers with values from registers or the formal parameter p. We represent an assignment which updates the registers $x_{i_1},\ldots,x_{i_m}$ with values from the registers $x_{j_1},\ldots,x_{j_n}$ or p as a mapping π from $\{x_{i_1},\ldots,x_{i_m}\}$ to $\{x_{j_1},\ldots,x_{j_n}\} \cup \{p\}$, meaning that the value of the register or parameter $\pi(x_{i_k})$ is assigned to the register x_{i_k}, for $k = 1,\ldots,m$.

Definition 1 (Register automaton). *A* register automaton *(RA) is a tuple* $\mathcal{A} = (L, l_0, \mathcal{X}, \Gamma, \lambda)$*, where*

- L *is a finite set of* locations*, with* $l_0 \in L$ *as the* initial location*,*
- $\mathcal{X}$ *maps each location* $l \in L$ *to a finite set* $\mathcal{X}(l)$ *of registers, and*
- Γ *is a finite set of* transitions*, each of form* $\langle l, \alpha(p), g, \pi, l' \rangle$*, where*
  - $l \in L$ *is a source location,*

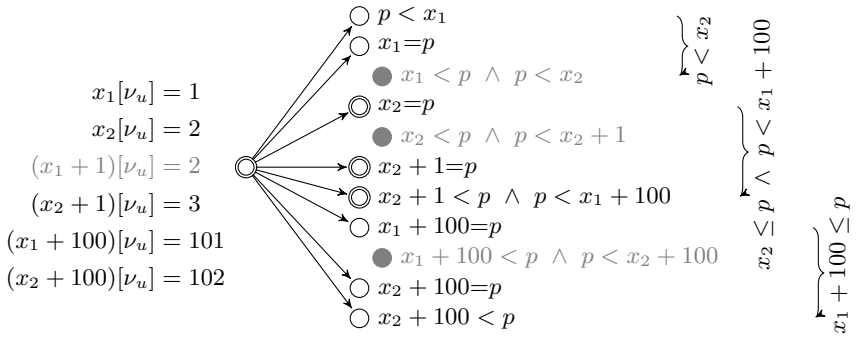

Fig. 5: Potential (left), maximally refined $(u, \hat{v})$-tree (center), and canonic guards (right) for u with $\nu_u = \{x_1 \mapsto 1, x_2 \mapsto 2\}$ and $\hat{v}$ with $|\hat{v}| = 1$. Actions omitted.

- $l' \in L$ *is a target location,*
- $\alpha(p)$ *is a parameterized symbol,*
- g *is a guard over* p *and* $\mathcal{X}(l)$*, and*
- π *(the* assignment*) is a mapping from* $\mathcal{X}(l')$ *to* $\mathcal{X}(l) \cup \{p\}$*, and*

– λ *maps each* $l \in L$ *to* $\{+, -\}$. □

We require register automata to have no initial registers (i.e., $\mathcal{X}(l_0) = \emptyset$) and to be *completely specified* in the sense that for each location $l \in L$ and action α, the disjunction of the guards on the α-transitions from l is equivalent to *true*.

RA Semantics Let us formalize the semantics of RAs. A *state* of an RA $\mathcal{A} = (L, l_0, \mathcal{X}, \Gamma, \lambda)$ is a pair $\langle l, \nu \rangle$ where $l \in L$ and ν is a valuation over $\mathcal{X}(l)$, i.e., a mapping from $\mathcal{X}(l)$ to $\mathcal{D}$. A *step* of $\mathcal{A}$, denoted $\langle l, \nu \rangle \xrightarrow{\alpha(d)} \langle l', \nu' \rangle$, transfers $\mathcal{A}$ from $\langle l, \nu \rangle$ to $\langle l', \nu' \rangle$ on input of the data symbol $\alpha(d)$ if there is a transition $\langle l, \alpha(p), g, \pi, l' \rangle \in \Gamma$ with

– $\nu \models g[d/p]$, i.e., d satisfies the guard g under the valuation ν, and
– ν' is the updated valuation with $\nu'(x_i) = \nu(x_j)$ if $\pi(x_i) = x_j$, otherwise $\nu'(x_i) = d$ if $\pi(x_i) = p$.

A *run* of $\mathcal{A}$ over a data word $w = \alpha(d_1) \dots \alpha(d_n)$ is a sequence of steps of $\mathcal{A}$

$$\langle l_0, \nu_0 \rangle \xrightarrow{\alpha_1(d_1)} \langle l_1, \nu_1 \rangle \quad \dots \quad \langle l_{n-1}, \nu_{n-1} \rangle \xrightarrow{\alpha_n(d_n)} \langle l_n, \nu_n \rangle$$

for some initial valuation ν_0. The run is *accepting* if $\lambda(l_n) = +$ and *rejecting* if $\lambda(l_n) = -$. The word w is *accepted (rejected) by* $\mathcal{A}$ *under* ν_0 if $\mathcal{A}$ has an accepting (rejecting) run over w which starts in $\langle l_0, \nu_0 \rangle$. An RA is *determinate* if there is no data word over which it has both accepting and rejecting runs. In this case we interpret an RA $\mathcal{A}$ as a mapping from the set of data words to $\{+, -\}$, where $+$ stands for ACCEPT and $-$ for REJECT. When using register automata as models for reactive system, we refine the set of actions into inputs and outputs (cf. [5]).

3.2 Tree Queries

For a data language $\mathcal{L}$, a data word u with $Vals(u) = d_1, \ldots, d_k$, and a set V of sequences of actions (so-called abstract suffixes), a (u, V)-*tree* is a decision tree (a tree-shaped RA) $\mathcal{T} = (L, l_0, \mathcal{X}, \Gamma, \lambda)$ with root l_0 and $\mathcal{X}(l_0) \subseteq \{x_1, \ldots, x_k\}$ that (1) has runs over exactly all data words v with $Acts(v) \in V$ and that (2) accepts a data word v from $\langle l_0, \nu_u \rangle$ iff $uv \in \mathcal{L}$. Please note, that we do not require $\mathcal{X}(l_0)$ to be empty for decision trees and let ν_u such that $\nu_u(x_i) = d_i$ for $x_i \in \mathcal{X}(l_0)$ and d_i the i-th data value of u.

A tree oracle for $\mathcal{L}$ is a function $\mathcal{O}$ that for any prefix u and set of abstract suffixes V constructs a (u, V)-tree $\mathcal{O}(u, V)$. The SL^* algorithm combines multiple symbolic decision trees (SDTs) into a conjectured model. We can implement a tree oracle by starting with a maximally refined symbolic decision tree that has one unique sequence of transitions for every $\mathcal{R}$-indistinguishable class of words $[uv]_\mathcal{R}$ with $Acts(v) \in V$ and then compute a more concise tree by iteratively merging equivalent subtrees.

Maximally refined SDTs. For simplicity, we describe the generation of a maximally refined symbolic decision tree for a prefix u and a single abstract suffix $\hat{v}$. This allows us to omit actions from the presentation. For $|Vals(u)| = k$, the *potential of* u is the set of terms $(x_i + c)$ with $1 \leq i \leq k$ and $c \in \{0, 1, 100\}$that can appear in guards after u. The valuation ν_u (with $\nu_u(x_i) = d_i$ for $d_i \in Vals(u)$) induces an order on the terms in the potential. An example of this order is shown on the left of Figure 5 for a word u with two data values.

Omitting the trivial case of the empty sequence, let $|\hat{v}| = 1$ for the moment. We generate guards for cases p smaller than the smallest term in the potential of u, p equal to one of the terms, p in the interval between two successive terms, and p greater than any term in the potential of u. These guards are maximally refined: each (satisfiable) guard describes one class $[uv]_\mathcal{R}$ of $\mathcal{R}$-indistinguishable words. We instantiate each guard with the help of a constraint solver and use an output query to determine if $uv \in \mathcal{L}$. Figure 5 (middle) exemplifies the construction. As indicated by gray lines on the left of the figure, some terms in the potential are equal. For these cases we pick one of the equal terms as the basis for guards. Gray colored guards cannot be instantiated and are omitted.

In the general case of $|\hat{v}| > 1$, we apply the above technique iteratively, generating sequences of guards and transitions for the parameters of $\hat{v}$. We maintain data values of the suffix symbolically during sequence generation and only instantiate complete sequences of guards. The approach scales to sets of suffix sequences as we construct maximally refined paths: paths of suffixes with common prefixes will have common guards for those prefixes and can be expressed as trees.

Maximally abstract SDTs and Monotonicity. In order to guarantee convergence of learning on a canonical automaton, the SL^* makes some monotonicity requirements on tree oracles [6]. For growing sets of abstract suffixes V, $V', \ldots$ with $V \subset V'$, it has to be shown that $\mathcal{O}(u, V')$ refines $\mathcal{O}(u, V)$ by only adding registers to $\mathcal{X}(l_0)$, and only refining guards of transitions. Additionally, if de-

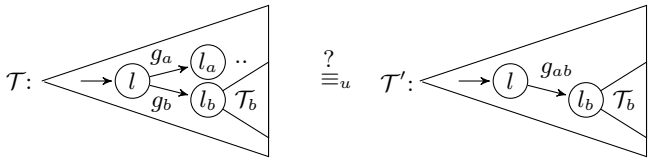

Fig. 6: Merging Sub-Trees of an SDT.

cision trees $\mathcal{O}(u, V)$ and $\mathcal{O}(u', V)$ cannot be made equal under some renaming of registers from $\mathcal{X}(l_0)$ in one tree, trees $\mathcal{O}(u, V')$ and $\mathcal{O}(u', V')$ cannot become equal either by such a renaming. These conditions trivially hold on maximally refined SDTs. Unfortunately, however, maximally refined SDTs do not lead to finite models during learning as the shape of a tree depends on the length of the prefix. We transform maximally refined SDTs into more abstract trees by merging transitions and equivalent sub-trees (akin to BDD minimization), thereby hiding irrelevant structural differences between trees.

The essential idea is that two (u, V)-trees $\mathcal{T}$ and $\mathcal{T}'$ are semantically equivalent after u, denoted by $\mathcal{T} \equiv_u \mathcal{T}'$, if both trees accept the same set of suffixes under initial valuation ν_u with $\nu_u(x_i) = d_i$ for $d_i \in Vals(u)$. We can check semantic equivalence with finitely many test runs (i.e., one for each path in a maximally refined SDT for V). Let now l be a location in $\mathcal{T}$ with outgoing transitions to l_a and l_b, guarded by g_a and g_b, respectively, as sketched in Figure 6. For some new guard g_{ab}, equivalent to $(g_a \vee g_b)$, we construct $\mathcal{T}'$ from $\mathcal{T}$ w.l.o.g. by removing the transition from l to l_a and the sub-tree rooted at l_a. On the transition from l to l_b, we replace g_b by g_{ab} (cf. right part of the figure). We abstract g_a and g_b into g_{ab} if $\mathcal{T} \equiv_u \mathcal{T}'$.

In order to arrive at a canonical representation, we perform merging in a fixed order: we always merge guards for the smallest possible terms with respect to the order on the potential (cf. maximally refined trees). This ensures that merging always results in intervals. An example is shown on the right of Figure 5. Merged guards are obtained from top (smaller terms) to bottom (greater terms).

Our semantic merging process satisfies all three requirements: Adding more suffixes (and hence paths) cannot lead to merging subtrees that could not be merged before. Guards are refined into finer intervals. Since the original boundaries will be maintained, monotonic growth of registers follows. Finally, since abstract trees are semantically equivalent to maximally refined trees, differences between trees are preserved when adding suffixes.

Output queries observe the behavior of the SUT on a sequence of test inputs. In learning-based testing, these queries are computed by executing tests on the actual system under test.

Testing has to be done in an adaptive fashion, synchronizing data values that are used in test inputs by the learning algorithm and those used in actual tests as the SUT may introduce new sequence numbers during tests. As an example, the learning algorithm may assume to receive a message $SYN + ACK$ with (new) sequence number 1. Then, in the actual communication the SUT sends a random new sequence number.

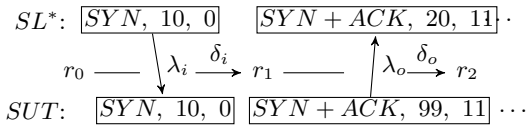

Fig. 7: Translation between Neat Trace and SUT Trace.

To tackle this problem, the work [2] introduces a determinizer component, placed between the *learner* and the SUT. This component provides the learner with a deterministic, or 'neat' view of the SUT, by constructing and applying a 1 to 1 mapping from regular values to *neat values*. This mapping transforms all relation equivalent traces (input/output sequences) encountered to a single neat trace. The learner then infers the SUT only in terms of its neat traces.

Output Queries.

We extend the determinizer concept to a setting with inequalities and sums. Our definition focuses on data values and ignores actions, which are invariant under mapping. The determinizer is the mapper $\mathcal{D} = \langle R, r_0, \delta_i, \delta_o, \lambda_i, \lambda_o \rangle$ over states $R = \{r \subseteq \mathbb{N} \times \mathbb{N} \mid r \text{ finite and one-to-one}\}$ with initial state $r_0 = \emptyset$. Value transformations (λ) and mapper updates (δ) are defined for $c \in 0, 1, 100$ and $x, y, n, m \in \mathbb{N}$ as follows.

$$\lambda_i(r,n) = \begin{cases} x + c & \text{if } m + c = n \text{ for some } (x,m) \in r \\ \mathsf{smaller}(\mathsf{dom}(r)) & \text{if } m + c > n \text{ for all } (\cdot, m) \in r \\ \mathsf{fresh}(\mathsf{dom}(r)) & \text{if } m + c < n \text{ for all } (\cdot, m) \in r \\ (x+y)/2 & \text{else; for } (x - c_1, m_l - c_1), (y - c_2, m_u - c_2) \in r \\ & \text{s.t. } (m_l < n < m_u) \text{ and } (m_u - m_l) \text{ minimal} \end{cases}$$

$$\lambda_o(r,x) = \begin{cases} n + c & \text{if } y + c = x \text{ for some } (y,n) \in r \\ \mathsf{fresh}(\mathsf{ran}(r)) & \text{otherwise} \end{cases}$$

$$\delta_i(r,n) = \begin{cases} r & \text{if } (\cdot, n) \in r \\ r \cup \{\ (\lambda_i(r,n),\ n)\ \} & \text{otherwise} \end{cases}$$

$$\delta_o(r,x) = \begin{cases} r & \text{if } (x, \cdot) \in r \\ r \cup \{\ (x,\ \lambda_o(r,x))\ \} & \text{otherwise} \end{cases}$$

There, dom and ran denote domain and image of a function. Functions $\mathsf{fresh} : \mathbb{N}^* \to \mathbb{N}$ and $\mathsf{smaller} : \mathbb{N}^* \to \mathbb{N}$ generate fresh values and smaller values. For $X \subset \mathbb{N}$ we use the concrete functions $\mathsf{fresh}(X) := (\lfloor max(X)) \div s_u \rfloor + 1) \times s_u$ and $\mathsf{smaller}(X) := (\lfloor \mathsf{min}(X) \div s_l \rfloor - 1) \times s_l$. Step sizes s_u and s_l are fixed big enough to avoid collisions (accidental relations between data values) during experiments.

Figure 7 shows an example application of the mapper, producing a neat trace from Figure 2. Whenever the system generates an output, the determinizer processes it by replacing the output values with neat values before delivering the output to the learner. Conversely, on generating a concrete input, the learner passes it to the determinizer which replaces neat input values with regular values, and sends the resulting input to the SUT. Every time it processes a value, the determinizer updates its state.

3.3 Model-based Testing

We instantiate the testing part of our framework with a relative simple adaptation of a random algorithm to the scenario of register automaton models. For a register automaton model $\mathcal{A}$, each test run begins by traversing the model to a randomly selected location of $\mathcal{A}$ and is continued by a random sequence of inputs until either a discrepancy is discovered between model and system under test, or until the run terminates and a new run starts.

Our extension consists in selecting data values for inputs. For a run with current prefix w and next input α, we use the machinery introduced above (the potential of a word, and symbolic guards that describe classes $[w\alpha(d)]_{\mathcal{R}}$ of data words) as a basis for computing a pool of data values for α. The pool contains one data value d for each $\mathcal{R}$-indistinguishable class $[w\alpha(d)]_{\mathcal{R}}$ of data words. We add a bias to the selection of data values, so that values in or related to those stored in registers in $\mathcal{A}$ after running over w are more likely to be picked.

We can easily obtain a PAC-inspired conformance guarantee (cf. [17]) with this testing method for the probability distribution on the set of data words induced by a model $\mathcal{A}$ and the above strategy for selecting tests. With respect to this distribution, $\mathcal{A}$ is an ϵ-approximation of SUT if $\sum_{w \in S} Pr(w) \leq \epsilon$ for the symmetric difference S of sets of words accepted by $\mathcal{A}$ and SUT. The probability of $\mathcal{A}$ not being an ϵ-approximation of the SUT after performing k independent test runs is at most $(1-\epsilon)^k$. For some confidence value δ, we simply choose k such that $(1-\epsilon)^k < \delta$ (i.e., such that $k > ln(\delta)/ln(1-\epsilon)$.

4 Testing TCP Implementations

We have implemented the theories introduced earlier into RaLib [5]. We then set up an experimental setup through which we could connect RaLib to various TCP clients. RaLib inferred models, which we checked manually for conformance with the specification.

4.1 Experimental Setup

The experimental setup used to learn TCP is similar to the setup used in [7] and [8]. As in those works, the alphabet used to learn TCP defines two types of inputs. The first type is *packet inputs*, used to describe TCP segments sent to the system. These inputs are parameterized by TCP flag combinations, sequence and acknowledgement numbers. The second type of inputs is *socket inputs* such as `connect` and `close`, referring to the methods defined by the socket interface. Outputs defined are *packet outputs*, which bear the same structure as packet inputs and describe TCP segments generated by the system, and *timeouts*, which suggest that no output was generated by the system. For model learning, we use the SL^* algorithm with the theory and optimizations discussed earlier. Additionally, we used techniques for reducing the size of counterexamples as shorter counterexamples tend to lead to shorter suffixes, which greatly decreases the

Table 1: Learning Statistics. `BASE` stands for Baseline. `[T]` marks Use of Typing.

SUL	Alpha.	Term.	Inp. Num. Loc.	Hyp.	Learning Inputs	Resets	Testing Inputs	Resets
Linux 3.19	`[T]BASE`	yes	6	15	4,311	947	113,921	11,720
	`BASE`	yes	6	15	9,930	2,168	116,479	12,339
	`[T]BASE+ACK`	yes	8	21	77,922	13,414	119,768	12,289
FreeBSD 11.0	`[T]BASE`	yes	6	16	4,239	933	113,953	11,708
	`BASE`	yes	6	16	9,958	2,152	116,446	12,333
	`[T]BASE+ACK`	no	8	21	418,977	80,200	81,024	8,367
Windows 10	`BASE-CLOSE`	no	6	14	193,712	24,848	119,768	12,289

number of inputs needed to run. For sample techniques and a corresponding discussion we refer to [11]. Finally, to speed up learning, we used multiple systems under learning in parallel. Model-based testing was done using the algorithm described in the previous section.

4.2 Experiments and Results

We attempted to learn TCP client implementations of Linux, FreeBSD and Windows. We chose clients, since they are simpler to learn and contain less redundancy compared to servers (cf. [8]). In terms of the configurations used, we disabled adaptive receive windows (or window scale), so that receive windows remain fixed over the course of each test. Moreover, in the segments sent to the SUT we advertise the same receive window as that of the SUT. Doing so we avoid having to include an additional sum constant for our own receive window.

Our baseline alphabet consists of the `connect`, `SYN+ACK`, `ACK+RST`, `RST` and `close` inputs. This alphabet covers several states in the specification. The alphabet should also reveal how SUTs in these states react to RST segments. These segments are generated in cases where one side abruptly terminates a connection and should be processed only if their sequence numbers are in window of the expected. We have also extended the alphabet with the `ACK` input if learning with the baseline was successful. To obtain models in an adequate time, we do not explore data relations between all formal parameters in some experiments. This optimization has been introduced as *typing* of symbolic parameters in [5].

Once a hypothesis was constructed, we tested it using the algorithm presented earlier. We have set the size of the random sequence to 10 (sufficient for exploring the behavior we are interested in) and ran $15,000$ tests on the final hypothesis. Using the confidence metric from the previous section, this yields a confidence of more than $99,9\%$ that a model is an 0.05%-approximation of the SUT for data words up to a length of 10 — relative to the probability distribution our randomized testing algorithm generates over the set of data words.

Table 1 reports the setting, termination status and learning statistics for all experiments done. The setting indicates the concrete SUT, the alphabet relative to the baseline and whether typing was used. Successful experiments took at most two days to complete, the determining factors being the size in parameters of the suffixes and the 0.3 seconds wait time used for each response before concluding a

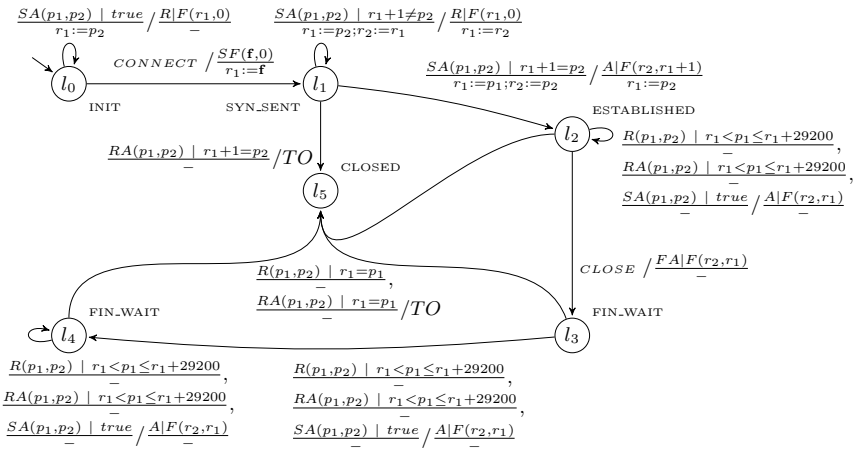

Fig. 8: Model of Linux Client. Flags are replaced by their starting characters (i.e. FIN by F, SYN by S). We group inputs with guards soliciting the same output and assignment over registers and use input/output notation. Inputs have guards over parameters. In outputs, parameters are instantiated.

timeout. We automatically terminated experiments still unresolved after $500,000$ inputs. For these experiments, we still display the last hypothesis and learning numbers at the point of termination. Results are available on RaLib's website.[3]

Using both un-typed and typed baseline alphabets we inferred models for Linux and FreeBSD. We inferred a model for Linux using the `ACK`-extended typed alphabet, but not for BSD. Learning FreeBSD for this setting followed a similar course to learning Linux, leading to a similar hypothesis. Testing generated a counterexample, whose processing resulted in a long new suffix. The suffix proved too expensive for tree queries to terminate within the input bounds set.

We couldn't learn Windows models even after removing the `CLOSE` input. Analysis of the last conjectured model and the generated tests revealed behavior inconsistent with the specification: Windows accepts sequence numbers up to and including window size plus one in the `ESTABLISHED` state for `RST` inputs. This helps demonstrate a limitation of our approach: relevant data relations $\mathcal{R}$ are an input to learning and convergence is guaranteed only for systems that respect $\mathcal{R}$ (cf. Section 3.1).

4.3 Analysis of Conformance to RFC

Figure 8 presents the model learned for Linux using the baseline alphabet. The models learned for FreeBSD and Linux are near identical with one exception. Linux defines an in-window sequence number as a value up to and including $rcv.nxt+win$ (for a next expected sequence number $rcv.nxt$). FreeBSD excludes the higher bound. Windows, on the other hand, even seems to include $rcv.nxt+$

[3] See: `https://goo.gl/23VNfv`

```
#define after(seq2, seq1)  before(seq1, seq2)
static inline bool before(__u32 seq1, __u32 seq2) {
        return (__s32)(seq1-seq2) < 0;
}
static inline bool tcp_sequence(
    const struct tcp_sock *tp, u32 seq, u32 end_seq) {
  return !before(end_seq, tp->rcv_wup) &&
    !after(seq, tp->rcv_nxt + tcp_receive_window(tp));
}
```

Listing 9: Relevant Code of TCP Implementation in Linux Kernel.

win+1. The RFC 793[14, page 26] specifies a closed upper bound. Thus, FreeBSD conforms to the upper bound requirement whereas Linux and Windows do not. For Linux, we trace this violation to code in the most recent kernel, v4.11.[4] Listing 9 shows the relevant code snippets. To check whether a sequence number is not after the window, they use the $!(seq > rcv.nxt + win)$ conjunct, allowing $rcv.nxt + win$ to be within the window. We inquired Linux developers about this issue and they confirmed it and said they would issue a fix for it. During our experiments, we have uncovered a different, unrelated, bug relating to faulty re-transmissions for which a fix has been issued.

Aside from that, reset processing seems to be implemented as stated in the RFC with the remark that both systems implement the 'Blind Reset Attack Using RST Bit' safe guard introduced in RFC 5961[1, page 7], by which only RST segments with the sequence number equal to the expected sequence number cause the termination of a connection. RST segments whose sequence number is in window but not equal to the expected sequence number prompt a 'challenge ACK response'. We can verify that this is the case by analyzing the Linux model's responses to RST segments in the `ESTABLISHED` and `FIN_WAIT1` states. As a note, RFC 5961 might have been the cause of the inconsistency remarked previously. As of this writing, RFC 5961 gives a wrong description of the within/outside window conditions of RFC 793. The error had been reported in 2016 and is included in the RFC errata[5].

5 Conclusion

Our work introduces the first application of register automata learning to real networked systems, in the form of TCP clients. To that end, we have developed the theories needed to learn TCP into the learning framework of [6]. We implemented heuristics that improve scalability of learning and developed a component that deals with non-determinism in fresh data values. The application of our learning-based testing setup resulted in models for TCP client implementations of Linux and FreeBSD. Our setup helped reveal violation of the RFC 793 standard [14] in Linux and Windows. In Linux we identified the root cause for the violation in the Kernel code.

[4] See: `https://goo.gl/9A8ZYM`
[5] See: `https://www.rfc-editor.org/errata/rfc5961`

In a next step, we plan to produce models for extended sets of inputs and models of TCP servers. Despite the optimizations used, we eventually faced combinatorial blow up in the number of required tests. Combining learning with static or symbolic analysis methods may help reducing this blow up by identifying more precisely the relations one should test for. This will also address the limitation of fixed relations that prevented us from learning a model for Windows.

References

1. R. Stewart A. Ramaiah and M. Dalal. Improving TCP's Robustness to Blind In-Window Attacks. RFC 5961, August 2010.
2. F. Aarts, P. Fiterău-Broştean, H. Kuppens, and F. W. Vaandrager. Learning register automata with fresh value generation. In *ICTAC 2015*, volume 9399 of *LNCS*, pages 165–183. Springer, 2015.
3. D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, November 1987.
4. T. Berg, O. Grinchtein, B. Jonsson, M. Leucker, H. Raffelt, and B. Steffen. On the correspondence between conformance testing and regular inference. In *FASE 2005*, pages 175–189. Springer, 2005.
5. S. Cassel, F. Howar, and B. Jonsson. RALib: A LearnLib extension for inferring EFSMs. In *DIFTS 2015*, 2015.
6. S. Cassel, F. Howar, B. Jonsson, and B. Steffen. Active learning for extended finite state machines. *Formal Aspects of Computing*, 28(2):233–263, 2016.
7. P. Fiterău-Broştean, R. Janssen, and F.W. Vaandrager. Learning fragments of the TCP network protocol. In F. Lang and F. Flammini, editors, *FMICS 2014*, volume 8718 of *LNCS*, pages 78–93. Springer, 2014.
8. P. Fiterău-Broştean, R. Janssen, and F.W. Vaandrager. Combining model learning and model checking to analyze TCP implementations. In *CAV 2016*, volume 9780 of *LNCS*, pages 454–471. Springer, 2016.
9. P. Fiterău-Broştean, T. Lenaerts, J. de Ruiter, E. Poll, F.W. Vaandrager, and P. Verleg. Model learning and model checking of ssh implementations. *SPIN Symposium*, page to appear in, 2017.
10. A. Hagerer, H. Hungar, O. Niese, and B. Steffen. Model Generation by Moderated Regular Extrapolation. In *FASE 2002*, volume 2306 of *LNCS*, pages 80–95. Springer, 2002.
11. P. Koopman, P. Achten, and R. Plasmeijer. *Model-Based Shrinking for State-Based Testing*, volume 8322 of *LNCS*, pages 107–124. Springer, 2014.
12. K. Meinke and M. A. Sindhu. Lbtest: A learning-based testing tool for reactive systems. In *ICST 2013*, pages 447–454. IEEE Computer Society, 2013.
13. D. Peled, M. Y. Vardi, and M. Yannakakis. Black Box Checking. *J. Autom. Lang. Comb.*, 7(2):225–246, November 2001.
14. J. Postel. Transmission Control Protocol. RFC 793, September 1981.
15. J. de Ruiter and E. Poll. Protocol state fuzzing of TLS implementations. In *USENIX Security*, pages 193–206, Washington, D.C., 2015. USENIX Association.
16. J. Tretmans. Model-based testing and some steps towards test-based modelling. In *SFM 2011*, LNCS, pages 297–326. Springer, 2011.
17. L. G. Valiant. A theory of the learnable. *Commun. ACM*, 27(11):1134–1142, November 1984.