Grey-Box Learning of Register Automata

Bharat Garhewal[1]*, Frits Vaandrager[1], Falk Howar[2],
Timo Schrijvers[1], Toon Lenaerts[1], and Rob Smits[1]

[1] Radboud University, Nijmegen, The Netherlands
{bharat.garhewal, frits.vaandrager}@ru.nl
[2] Dortmund University of Technology

Abstract. Model learning (a.k.a. active automata learning) is a highly effective technique for obtaining black-box finite state models of software components. Thus far, generalization to infinite state systems with inputs and outputs that carry data parameters has been challenging. Existing model learning tools for infinite state systems face scalability problems and can only be applied to restricted classes of systems (register automata with equality/inequality). In this article, we show how we can boost the performance of model learning techniques by extracting the constraints on input and output parameters from a run, and making this grey-box information available to the learner. More specifically, we provide new implementations of the tree oracle and equivalence oracle from RALib, which use the derived constraints. We extract the constraints from runs of Python programs using an existing tainting library for Python, and compare our grey-box version of RALib with the existing black-box version on several benchmarks, including some data structures from Python's standard library. Our proof-of-principle implementation results in almost two orders of magnitude improvement in terms of numbers of inputs sent to the software system. Our approach, which can be generalized to richer model classes, also enables RALib to learn models that are out of reach of black-box techniques, such as combination locks.

Keywords: Model learning · Active Automata Learning · Register Automata · RALib · Grey-box · Tainting

1 Introduction

Model learning, also known as active automata learning, is a black-box technique for constructing state machine models of software and hardware components from information obtained through testing (i.e., providing inputs and observing the resulting outputs). Model learning has been successfully used in numerous applications, for instance for generating conformance test suites of software components [14], finding mistakes in implementations of security-critical protocols [9–11], learning interfaces of classes in software libraries [15], and checking that

* Supported by NWO TOP project 612.001.852 "Grey-box learning of Interfaces for Refactoring Legacy Software (GIRLS)".

a legacy component and a refactored implementation have the same behaviour [20]. We refer to [18, 21] for surveys and further references.

In many applications it is crucial for models to describe *control flow*, i.e., states of a component, *data flow*, i.e., constraints on data parameters that are passed when the component interacts with its environment, as well as the mutual influence between control flow and data flow. Such models often take the form of *extended finite state machines* (EFSMs). Recently, various techniques have been employed to extend automata learning to a specific class of EFSMs called *register automata*, which combine control flow with guards and assignments to data variables [1, 2, 5].

While these works demonstrate that it is theoretically possible to infer such richer models, the presented approaches do not scale well and they are not yet satisfactorily developed for richer classes of models (c.f. [17]): Existing techniques either rely on manually constructed mappers that abstract the data aspects of input and output symbols into a finite alphabet, or otherwise infer guards and assignments from black-box observations of test outputs. The latter can be costly, especially for models where control flow depends on test on data parameters in input: in this case, learning an exact guard that separates two control flow branches may require a large number of queries.

One promising strategy for addressing the challenge of identifying data-flow constraints is to augment learning algorithms with white-box information extraction methods, which are able to obtain information about the SUL at lower cost than black-box techniques. Several researchers have explored this idea. Giannakopoulou et al. [12] develop an active learning algorithm that infers safe interfaces of software components with guarded actions. In their model, the teacher is implemented using concolic execution for the identification of guards. Cho et al. [8] present MACE an approach for concolic exploration of protocol behaviour. The approach uses active automata learning for discovering so-called deep states in the protocol behaviour. From these states, concolic execution is employed in order to discover vulnerabilities. Similarly, Botinčan and Babić [4] present a learning algorithm for inferring models of stream transducers that integrates active automata learning with symbolic execution and counterexample-guided abstraction refinement. They show how the models can be used to verify properties of input sanitizers in Web applications. Finally, Howar et al. [16] extend the work of [12] and integrate knowledge obtained through static code analysis about the potential effects of component method invocations on a component's state to improve the performance during symbolic queries. So far, however, white-box techniques have never been integrated with learning algorithms for register automata.

In this article, we show how dynamic taint analysis can be used to efficiently extract constraints on input and output parameters from a test, and improves the performance of the SL^* algorithm of Cassel et al. [7]. The SL^* algorithm generalises the classical L^* algorithm of Angluin [3] and has been used successfully to learn register automaton models, for instance of Linux and Windows imple-

mentations of TCP [10]. We have implemented the presented method on top of RALib [6], a library that provides an implementation of the SL^* algorithm.

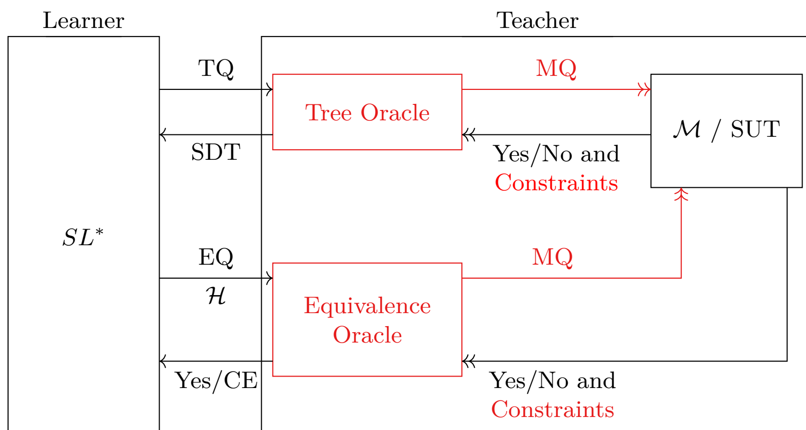


Fig. 1: MAT Framework (Our addition — tainting — in red): Double arrows indicate possible multiple instances of a query made by an oracle for a single query by the learner.

The integration of the two techniques (dynamic taint analysis and learning of register automata models) can be explained most easily with reference to the architecture of RALib, shown in Figure 1, which is a variation of the *Minimally Adequate Teacher* (MAT) framework of [3]: In the MAT framework, learning is viewed as a game in which a *learner* has to infer the behaviour of an unknown register automaton $\mathcal{M}$ by asking queries to a *teacher*. We postulate $\mathcal{M}$ models the behaviour of a *System Under Test (SUT)*. In the learning phase, the learner (i.e., SL^*) is allowed to ask questions to the teacher in the form of *tree queries* (TQs) and the teacher responds with *symbolic decision trees* (SDTs). In order to construct these SDTs, the teacher uses a *tree oracle*, which queries the SUT with *membership queries* (MQs) and receives a yes/no reply to each. Typically, the tree oracle asks many membership queries to answer a single tree query in order to infer causal impact and flow of data values. Based on the answers on a number of tree queries, the learner constructs an *hypothesis* in the form of a register automaton $\mathcal{H}$. The learner submits $\mathcal{H}$ as an *equivalence query (EQ)* to the teacher, asking whether $\mathcal{H}$ is equivalent to the SUT model $\mathcal{M}$. The teacher uses an *equivalence oracle* to answer equivalence queries. Typically, the equivalence oracle asks many membership queries to answer a single equivalence query. If, for all membership queries, the output produced by the SUT is consistent with hypothesis $\mathcal{H}$, the answer to the equivalence query is 'Yes' (indicating learning is complete). Otherwise, the answer 'No' is provided, together with a counterexample (CE) that indicates a difference between $\mathcal{H}$ and $\mathcal{M}$. Based on this counterexample, learning continues. In this extended MAT framework, we

have constructed new implementations of the tree oracle and equivalence oracle that leverage the constraints on input and output parameters that are imposed by a program run: dynamic tainting is used to extract all constraints on parameters that are performed during a run of a program. Our implementation learns models of Python programs, using an existing tainting library for Python [13].

We compare our grey-box tree oracle and equivalence oracle with the existing black-box versions of these oracles on several benchmarks, including Python's `queue` and `set` modules. Our proof-of-concept implementation[3] results in almost two orders of magnitude improvement in terms of numbers of inputs sent to the software system. Our approach, which generalises to richer model classes, also enables RALib to learn models that are completely out of reach for black-box techniques, such as combination locks.

Outline: Section 2 contains preliminaries, Section 3 discusses tainting in our Python SUTs, Section 4 contains the algorithms we use to answer TQs using tainting and the definition for the tainted equivalence oracle needed to learn combination lock automata, Section 5 contains the experimental evaluation of our technique, and Section 6 concludes.

2 Preliminary definitions and constructions

This section contains the definitions and constructions necessary to understand active automata learning for models with dataflow. We first define the concept of a *structure*, followed by *guards*, *data languages*, *register automata*, and finally *symbolic decision trees.*

Definition 1 (Structure). *A structure $\mathcal{S} = \langle R, \mathcal{D}, \mathcal{R} \rangle$ is a triple where R is a set of relation symbols, each equipped with an arity, $\mathcal{D}$ is an infinite domain of data values, and $\mathcal{R}$ contains a distinguished n-ary relation $r^{\mathcal{R}} \subseteq \mathcal{D}^n$ for each n-ary relation symbol $r \in R$.*

In the remainder of this section, we fix a structure $\mathcal{S} = \langle R, \mathcal{D}, \mathcal{R} \rangle$, where R contains a binary relation symbol $=$ and unary relation symbols $= c$, for each c contained in a finite set C of constant symbols, $\mathcal{D}$ equals the set $\mathbb{N}$ of natural numbers, $=^{\mathcal{R}}$ is interpreted as the equality predicate on $\mathbb{N}$, and to each symbol $c \in C$ a natural number n_c is associated such that $(= c)^{\mathcal{R}} = \{n_c\}$.

Guards are a restricted type of Boolean formulas that may contain relation symbols from R.

Definition 2 (Guards). *We postulate a countably infinite set $\mathcal{V} = \{v_1, v_2, \ldots\}$ of* variables. *In addition, there is a variable $p \notin \mathcal{V}$ that will play a special role as formal parameter of input symbols; we write $\mathcal{V}^+ = \mathcal{V} \cup \{p\}$. A* guard *is a conjunction of relation symbols and negated relation symbols over variables. Formally, the set of* guards *is inductively defined as follows:*

[3] Available at `https://bitbucket.org/toonlenaerts/taintralib/src/basic/`.

– *If* $r \in R$ *is an* n*-ary relation symbol and* $x_1, \ldots, x_n$ *are variables from* $\mathcal{V}^+$*, then* $r(x_1, \ldots, x_n)$ *and* $\neg r(x_1, \ldots, x_n)$ *are guards.*
– *If* g_1 *and* g_2 *are guards then* $g_1 \wedge g_2$ *is a guard.*

Let $X \subset \mathcal{V}^+$*. We say that* g *is a guard* over X *if all variables that occur in* g *are contained in* X*. A* variable renaming *is a function* $\sigma : X \to \mathcal{V}+$*. If* g *is a guard over* X *then* $g[\sigma]$ *is the guard obtained by replacing each variable* x *in* g *by* $\sigma(x)$*.*

Next, we define the notion of a *data language*. For this, we fix a finite set of *actions* Σ. A *data symbol* $\alpha(d)$ is a pair consisting of an action $\alpha \in \Sigma$ and a data value $d \in \mathcal{D}$. While relations may have arbitrary arity, we assume that all actions have an arity of one to ease notation and simplify the text. A *data word* is a finite sequence of data symbols, and a *data language* is a set of data words. We denote concatenation of data words w and w' by $w \cdot w'$, where w is the *prefix* and w' is the *suffix*. $Acts(w)$ denotes the sequence of actions $\alpha_1\alpha_2 \ldots \alpha_n$ in w, and $Vals(w)$ denotes the sequence of data values $d_1 d_2 \ldots d_n$ in w. We refer to a sequence of actions in Σ^* as a *symbolic suffix*. If w is a symbolic suffix then we write $[\![w]\!]$ for the set of data words u with $Acts(u) = w$.

Data languages may be represented by *register automaton*, defined below.

Definition 3 (Register Automaton). *A Register Automaton (RA) is a tuple* $\mathscr{M} = (L, l_0, \mathcal{X}, \Gamma, \lambda)$ *where*

– L *is a finite set of locations, with* l_0 *as the initial location;*
– $\mathcal{X}$ *maps each location* $l \in L$ *to a finite set of registers* $\mathcal{X}(l)$*;*
– Γ *is a finite set of transitions, each of the form* $\langle l, \alpha(p), g, \pi, l' \rangle$*, where*
  - l, l' *are source and target locations respectively,*
  - $\alpha(p)$ *is a parametrised action,*
  - g *is a* guard *over* $\mathcal{X}(l) \cup \{p\}$*, and*
  - π *is an assignment mapping from* $\mathcal{X}(l')$ *to* $\mathcal{X}(l) \cup \{p\}$*; and*

– $\lambda : L \to \{+, -\}$.

We require that $\mathscr{M}$ *is* deterministic *in the sense that for each location* $q \in Q$ *and input symbol* $\alpha \in \Sigma$*, the conjunction of the guards of any pair of distinct* α*-transitions with source* q *is not satisfiable.* $\mathscr{M}$ *is* completely specified *if for all* α*-transitions out of a location, the disjunction of the guards of the* α*-transitions is a tautology.* $\mathscr{M}$ *is said to be* simple *if there are no registers in the initial location, i.e.,* $\mathcal{X}(l) = \varnothing$*. In this text, all RAs are assumed to be completely specified and simple, except when we explicitly say they are not.*

Example 1 (FIFO-buffer). The register automaton displayed in Figure 2 models a FIFO-buffer with capacity 2. It has three accepting locations l_0, l_1 and l_2; and one rejecting location l_3. Function $\mathcal{X}$ assigns the empty set of registers to locations l_0 and l_3, singleton set $\{x\}$ to location l_1, and set $\{x, y\}$ to l_2.

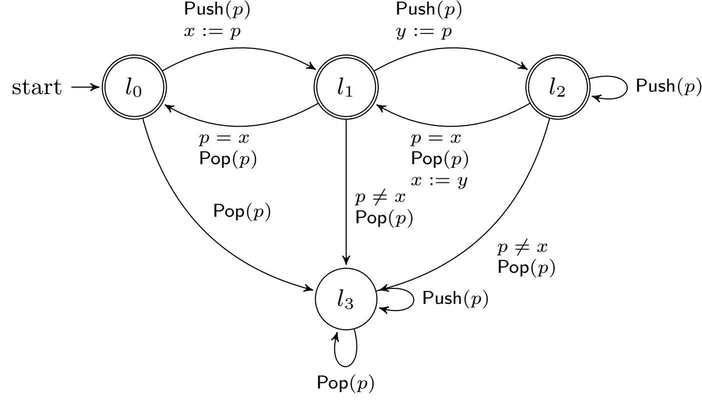


Fig. 2: FIFO-buffer with a capacity of 2 modelled as a register automaton

2.1 Semantics of a RA

We now formalise the semantics of an RA. A *valuation* of a set of variables X is a function $\nu : X \to \mathcal{D}$ that assigns data values to variables in X. If ν is a valuation of X and g is a guard with variables contained in X then $\nu \models g$ is defined inductively by:

- $\nu \models r(x_1, \ldots, x_n)$ iff $(\nu(x_1), \ldots, \nu(x_n)) \in r^{\mathcal{R}}$
- $\nu \models \neg r(x_1, \ldots, x_n)$ iff $(\nu(x_1), \ldots, \nu(x_n)) \notin r^{\mathcal{R}}$
- $\nu \models g_1 \wedge g_2$ iff $\nu \models g_1$ and $\nu \models g_2$

A *state* of a RA $\mathscr{M} = (L, l_0, \mathcal{X}, \Gamma, \lambda)$ is a pair $\langle l, \nu \rangle$, where $l \in L$ is a location and $\nu : \mathcal{X}(l) \to \mathcal{D}$ is a valuation of the set of registers at location l. A *run* of $\mathscr{M}$ over data word $w = \alpha_1(d_1) \ldots \alpha_n(d_n)$ is a sequence

$$\langle l_0, \nu_0 \rangle \xrightarrow{\alpha_1(d_1), g_1, \pi_1} \langle l_1, \nu_1 \rangle \ldots \langle l_{n-1}, \nu_{n-1} \rangle \xrightarrow{\alpha_n(d_n), g_n, \pi_n} \langle l_n, \nu_n \rangle,$$

where

- for each $0 \leq i \leq n$, $\langle l_i, \nu_i \rangle$ is a state (with l_0 the initial location),
- for each $0 < i \leq n$, $\langle l_{i-1}, \alpha_i(p), g_i, \pi_i, l_i \rangle \in \Gamma$ such that $\iota_i \vDash g_i$ and $\nu_i = \iota_i \circ \pi_i$, where $\iota_i = \nu_{i-1} \cup \{(p, d_i)\}$ extends ν_{i-1} by mapping p to d_i.

A run is *accepting* if $\lambda(l_n) = +$, else *rejecting*. The language of $\mathscr{M}$, notation $L(\mathscr{M})$, is the set of words w such that $\mathscr{M}$ has an accepting run over w. Word w is *accepted (rejected) under* valuation ν_0 if $\mathscr{M}$ has an accepting (rejecting) run that starts in state $\langle l_0, \nu_o \rangle$.

Example 2. Consider the FIFO-buffer example from Figure 2. This register automaton has a run

$$\begin{aligned}
\langle l_0, \nu_0 = []\rangle &\xrightarrow{\mathsf{Push}(7), g_1 \equiv \top, \pi_1 = [x \mapsto p]} \langle l_1, \nu_1 = [x \mapsto 7]\rangle \\
&\xrightarrow{\mathsf{Push}(7), g_2 \equiv \top, \pi_2 = [x \mapsto x, y \mapsto p]} \langle l_2, \nu_2 = [x \mapsto 7, y \mapsto 7]\rangle \\
&\xrightarrow{\mathsf{Pop}(7), g_3 \equiv p = x, \pi_3 = [x \mapsto y]} \langle l_1, \nu_3 = [x \mapsto 7]\rangle \\
&\xrightarrow{\mathsf{Push}(5), g_4 \equiv \top, \pi_4 = [x \mapsto x, y \mapsto p]} \langle l_2, \nu_4 = [x \mapsto 7, y \mapsto 5]\rangle \\
&\xrightarrow{\mathsf{Pop}(7), g_5 \equiv p = x, \pi_5 = [x \mapsto y]} \langle l_1, \nu_5 = [x \mapsto 5]\rangle \\
&\xrightarrow{\mathsf{Pop}(5), g_6 \equiv p = x, \pi_6 = []} \langle l_0, \nu_6 = []\rangle
\end{aligned}$$

and thus the trace is $\mathsf{Push}(7)$ $\mathsf{Push}(7)$ $\mathsf{Pop}(7)$ $\mathsf{Push}(5)$ $\mathsf{Pop}(7)$ $\mathsf{Pop}(5)$. ⌟

2.2 Symbolic Decision Tree

The SL^* algorithm uses *tree queries* in place of membership queries. The arguments of a tree query are a prefix data word u and a symbolic suffix w, i.e., a data word with uninstantiated data parameters. The response to a tree query is a so called *symbolic decision tree* (SDT), which has the form of tree-shaped register automaton that accepts/rejects suffixes obtained by instantiating data parameters in one of the symbolic suffixes. Let us illustrate this on the FIFO-

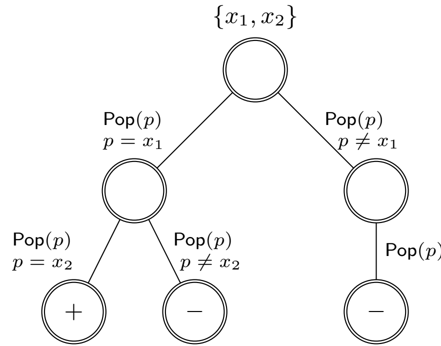


Fig. 3: A symbolic decision tree for prefix Push(5) Push(7) and (symbolic) suffix Pop Pop.

buffer example for the prefix $\mathsf{Push}(5)$ $\mathsf{Push}(7)$ and the symbolic suffix Pop Pop. The acceptance/rejection of suffixes obtained by instantiating data parameters after $\mathsf{Push}(5)$ $\mathsf{Push}(7)$ can be represented by the SDT in Figure 3. In the initial location, values 5 and 7 from the prefix are stored in registers x_1 and x_2, respectively. Thus, in general, SDTs will not be simple register automata. Moreover,

since the leaves of an SDT have no outgoing transitions, they are also not completely specified. We use the convention that register x_i stores the ith data value from the prefix. The SDT accepts suffixes of form $\mathsf{Pop}(d_1)$ $\mathsf{Pop}(d_2)$ iff d_1 equals the value stored in register x_1, and d_2 equals the data value stored in register x_2.

The formal definitions of an SDT and the notion of a tree oracle are presented in Appendix A. For a more detailed discussion we refer to [7].

3 Tainting

We postulate that the behaviour of the SUT (in our case: a Python program) can be modelled by a register automaton $\mathscr{M}$. In a black-box setting, observations on the SUT will then correspond to words from the data language of $\mathscr{M}$. In this section, we will describe the additional observations that a learner can make in a grey-box setting, where also the constraints on the data parameters that are imposed within a run become visible. In this setting, observations of the learner will correspond to what we call tainted words of $\mathscr{M}$. Tainting semantics is an extension of the standard semantics in which each input value is "tainted" with a unique marker from $\mathcal{V}$. In a data word $w = \alpha_1(d_1)\alpha_2(d_2)\ldots\alpha_n(d_n)$, the first data value d_1 is tainted with marker v_1, the second data value d_2 with v_2, etc. Whereas the same data value may occur repeatedly in a data word, all the markers are different.

3.1 Semantics of Tainting

A *tainted state* of a RA $\mathscr{M} = (L, l_0, \mathcal{X}, \Gamma, \lambda)$ is a triple $\langle l, \nu, \zeta \rangle$, where $l \in L$ is a location, $\nu : \mathcal{X}(l) \to \mathcal{D}$ is a valuation, and $\zeta : \mathcal{X}(l) \to \mathcal{V}$ is a function that assigns a marker to each register of l. A *tainted run* of $\mathscr{M}$ over data word $w = \alpha_1(d_1)\ldots\alpha_n(d_n)$ is a sequence

$$\tau = \langle l_0, \nu_0, \zeta_0 \rangle \xrightarrow{\alpha_1(d_1), g_1, \pi_1} \langle l_1, \nu_1, \zeta_1 \rangle \ldots \langle l_{n-1}, \nu_{n-1} \rangle \xrightarrow{\alpha_n(d_n), g_n, \pi_n} \langle l_n, \nu_n, \zeta_n \rangle,$$

where

- $\langle l_0, \nu_0 \rangle \xrightarrow{\alpha_1(d_1), g_1, \pi_1} \langle l_1, \nu_1 \rangle \ldots \langle l_{n-1}, \nu_{n-1} \rangle \xrightarrow{\alpha_n(d_n), g_n, \pi_n} \langle l_n, \nu_n \rangle$ is a run of $\mathscr{M}$,
- for each $0 \leq i \leq n$, $\langle l_i, \nu_i, \zeta_i \rangle$ is a tainted state,
- for each $0 < i \leq n$, $\zeta_i = \kappa_i \circ \pi_i$, where $\kappa_i = \zeta_{i-1} \cup \{(p, v_i)\}$.

The tainted word of τ is the sequence $w = \alpha_1(d_1)G_1\alpha_2(d_2)G_2 \cdots \alpha_n(d_n)G_n$, where $G_i = g_i[\kappa_i]$, for $0 < i \leq n$. We define *constraints*$_{\mathscr{M}}(\tau) = [G_1, \ldots, G_n]$.

Let $w = \alpha_1(d_1)\ldots\alpha_n(d_n)$ be a data word. Since register automata are deterministic, there is a unique tainted run τ over w. We define *constraints*$_{\mathscr{M}}(w) =$ *constraints*$_{\mathscr{M}}(\tau)$, that is, the constraints associated to a data word are the constraints of the unique tainted run that corresponds to it. In the untainted setting a membership query for data word w leads to a response "yes" if $w \in L(\mathscr{M})$, and a response "no" otherwise, but in a tainted setting the predicates *constraints*$_{\mathscr{M}}(w)$ are also included in the response, and provide additional information that the learner may use to accomplish its task.

Example 3. Consider the FIFO-buffer example from Figure 2. This register automaton has a tainted run

$$\begin{aligned}
\langle l_0, [], [] \rangle &\xrightarrow{\mathsf{Push}(7)} \langle l_1, [x \mapsto 7], [x \mapsto v_1] \rangle \\
&\xrightarrow{\mathsf{Push}(7)} \langle l_2, [x \mapsto 7, y \mapsto 7], [x \mapsto v_1, y \mapsto v_2] \rangle \\
&\xrightarrow{\mathsf{Pop}(7)} \langle l_1, [x \mapsto 7], [x \mapsto v_2] \rangle \\
&\xrightarrow{\mathsf{Push}(5)} \langle l_2, [x \mapsto 7, y \mapsto 5], [x \mapsto v_2, y \mapsto v_4] \rangle \\
&\xrightarrow{\mathsf{Pop}(7)} \langle l_1, [x \mapsto 5], [y \mapsto v_4] \rangle \\
&\xrightarrow{\mathsf{Pop}(5)} \langle l_0, [], [] \rangle
\end{aligned}$$

(For readability, guards g_i and assignments π_i have been left out.) The constraints in the corresponding tainted trace can be computed as follows:

$$\begin{aligned}
\kappa_1 &= [p \mapsto v_1] & G_1 &\equiv \top[\kappa_1] \equiv \top \\
\kappa_2 &= [x \mapsto v_1, p \mapsto v_2] & G_2 &\equiv \top[\kappa_2] \equiv \top \\
\kappa_3 &= [x \mapsto v_1, y \mapsto v_2, p \mapsto v_3] & G_3 &\equiv (p = x)[\kappa_3] \equiv v_3 = v_1 \\
\kappa_4 &= [x \mapsto v_2, p \mapsto v_4] & G_4 &\equiv \top[\kappa_4] \equiv \top \\
\kappa_5 &= [x \mapsto v_2, y \mapsto v_4, p \mapsto v_5] & G_5 &\equiv (p = x)[\kappa_5] \equiv v_5 = v_2 \\
\kappa_6 &= [x \mapsto v_4, p \mapsto v_6] & G_6 &\equiv (p = x)[\kappa_6] \equiv v_6 = v_4
\end{aligned}$$

and thus the tainted word is:

$$\mathsf{Push}(7)\ \top\ \mathsf{Push}(7)\ \top\ \mathsf{Pop}(7)\ v_3 = v_1\ \mathsf{Push}(5)\ \top\ \mathsf{Pop}(7)\ v_5 = v_2\ \mathsf{Pop}(5)\ v_6 = v_4,$$

and the corresponding list of constraints is $[\top, \top, v_3 = v_1, \top, v_5 = v_2, v_6 = v_4]$. ⌟

Various techniques can be used to observe tainted traces, for instance symbolic and concolic execution. In this work, we have used a Python library called "`taintedstr`" to achieve tainting in Python and make tainted traces available to the learner.

3.2 Tainting in Python

Tainting in Python is achieved by using a library called "`taintedstr`"[4], which implements a "`tstr`" (*tainted string*) class. We do not discuss the entire implementation in detail, but only introduce the portions relevant to our work. The "`tstr`" class works by *operator overloading*: each operator (method) in Python can be overloaded to record the invocation of the operator. The `tstr` class overloads the implementation of the "`__eq__`" (equality) method in Python's `str` class, amongst others. In this text, we only consider the equality method. A `tstr` object x can be considered as a triple $\langle o, t, cs \rangle$, where: o is the (base) string

[4] See [13] and `https://github.com/vrthra/taintedstr`.

object, t is the taint value associated with string o, and cs is a set of comparisons made by x with other objects, where each comparison $c \in cs$ is a triple $\langle m, a, b \rangle$ where

- m is the name of the binary method invoked on x,
- a is a copy of x, and
- b is the argument supplied to m.

Each a method m in the `tstr` class is an overloaded implementation of the relevant (base) method f as follows:

```
1 def m(self, other):
2     self.cs.add((m._name_, self, other))
3     return self.o.f(other) # 'o' is the base string
```

We present a short example of how such a method would work below:

Example 4 (`tstr` tainting). Consider two `tstr` objects: $x_1 = \langle \text{“1”}, 1, \emptyset \rangle$ and $x_2 = \langle \text{“1”}, 2, \emptyset \rangle$. Calling $x_1 == x_2$ returns **True** as “1 == 1”; however, as a side-effect, the set of comparisons cs in x_1 is updated with the triple $c = \langle \text{“__eq__”}, x_1, x_2 \rangle$. We may then confirm that x_1 is compared to x_2 by checking the taint values of the variables in comparison c: $x_1.t = 1$ and $x_2.t = 2$. Doing so may seem non-obvious in this example, but is necessary when two objects have the same string value, as we have overloaded the equality method. ⌟

We will only posses information about comparisons which are actually performed when running the code and only those which deal with methods called by a `tstr` object.

Example 5 (Complicated Comparison). Consider the following code snippet, where x_1, x_2, x_3 are `tstr` objects with some base values and $1, 2, 3$ as taint values respectively:

```
1   if not (x_1 == x_2 or (x_2 != x_3)):
2       # do something
```

Consider a case where the base values of x_1 and x_2 are equal: the Python interpreter will “short-circuit” the if-statement and the second condition, $x_2 \neq x_3$, will not be evaluated. Thus, the set of comparisons of x_1 will only contain the equality comparison with x_2 and the set of comparisons of x_2 will not record the comparison with x_3 as $x_2.__\text{ne}__(x_3)$ was never called. On the other hand, if the base values of x_1 and x_2 are not equal, the interpreter will not short-circuit, and both comparisons will be recorded as $\{x_2 = x_3, x_1 \neq x_2\}$. Although the comparisons are stored as a set, from the perspective of the tainted trace, the guard(s) is a single conjunction: $x_2 = x_3 \wedge x_1 \neq x_2$. However, note that the external negation (the `not` wrapping the two conditions) will not be recorded by any of the `tstr` objects: the negation was not performed on the `tstr` objects. ⌟

4 Learning Register Automata using Tainting

Given an SUT and a tree query, we generate an SDT in the following steps: *(i)* construct a *characteristic predicate* of the tree query (Algorithm 1) using membership and guard queries, *(ii)* transform the characteristic predicate into a non-minimal SDT using Algorithm 2, and *(iii)* minimize the obtained SDT using Algorithm 3.

4.1 Tainted Tree Oracle

Construction of Characteristic Predicate For $u = \alpha(d_1) \cdots \alpha_k(d_k)$ a data word, ν_u denotes the valuation of $\{x_1, \ldots, x_k\}$ with $\nu_u(x_i) = d_i$, for $1 \leq i \leq k$. Suppose $w = \alpha_{k+1} \cdots \alpha_{k+n}$ is a symbolic suffix. Then H is a *characteristic predicate* for u and w in $\mathscr{M}$ if, for each valuation ν of $\{x_1, \ldots, x_{k+n}\}$ that extends ν_u,

$$\nu \models H \quad \Leftrightarrow \quad \alpha_1(\nu(x_1)) \cdots \alpha_{k+n}(\nu(x_{k+n})) \in L(\mathscr{M}),$$

that is, H characterises the data words u' with $Acts(u') = w$ such that $u \cdot u'$ is accepted by $\mathscr{M}$. A characteristic predicate is computed by Algorithm 1, using tainted membership queries, that is, membership queries that do not only return a yes/no answer but also the constraints on the parameters that are imposed by a run. During the execution of the algorithm, predicate G describes the part of the parameter space that still needs to be explored, and H is the characteristic predicate for the part of the parameter space that has been covered. We note that if there exists no parameter space to be explored (i.e., w is empty), and $u \cdot w \in L(\mathscr{M})$, then $H := \top$.

Algorithm 1: `ComputeCharacteristicPredicate`

Data: A tree query consisting of a prefix $u = \alpha_1(d_1) \cdots \alpha_k(d_k)$ and symbolic suffix $w = \alpha_{k+1} \cdots \alpha_{k+n}$
Result: A characteristic predicate for u and w in $\mathscr{M}$

1 $G := \top$, $H := \bot$, $V := \{x_1, \ldots, x_{k+n}\}$
2 **do**
3     $\nu :=$ valuation for V that extends ν_u and satisfies $\nu \models G$
4     $z := \alpha_1(\nu(x_1)) \cdots \alpha_{k+n}(\nu(x_{k+n}))$ `// Construct query`
5     $I := \bigwedge_{i=k+1}^{k+n} constraints_{\mathscr{M}}(z)[i]$ `// Constraints resulting from query`
6   **if** $z \in L(\mathscr{M})$ **then** `// Result query ''yes'' or ''no''`
7         $H := H \vee I$
8     $G := G \wedge \neg I$
9 **while** $\exists$ *valuation* ν *for* V *that extends* ν_u *and satisfies* $\nu \models G$
10 **return** H

Construction of a non-minimal SDT Given a characteristic predicate H for a tree query (u, w), we convert H into a (possibly) non-minimal symbolic decision tree (i.e., an SDT which may contain redundancies) using Algorithm 2. We no longer need to query the SUT and may construct the SDT directly from

Algorithm 2: `SDTConstructor`

Data: Characteristic predicate H, Integer $idx = k + 1$

Result: Non-minimal SDT $\mathcal{T}$

1 **if** $idx = k + n$ **then**
2     $l_0 :=$ SDT node
3     $\lambda := -$ if $H = \bot$ else $+$ // Leaf node of the SDT
4   **return** $\langle \{l_0\}, l_0, \varnothing, \varnothing, \lambda \rangle$
5 **else**
6     $\mathcal{T} :=$ SDT node
7     $I_t := \{i \mid x_n \odot x_i \in H,\ n > i\}$ // $\odot \in \{=, \neq\}$ is symmetric
8   **if** I_t *is* $\varnothing$ **then**
9         $t :=$ `SDTConstructor`$(H, idx + 1)$ // No guards present
10   Add t with guard $\top$ to $\mathcal{T}$
11   **else**
12        $g := \bigwedge_{i \in I_t} x_n \neq x_i$ // Disequality guard case
13        $H' := \bigvee_{f \in H} f \wedge g$ if $f \wedge g$ is satisfiable else $\bot$ // f is a disjunct
14        $t' :=$ `SDTConstructor`$(H', idx + 1)$
15   Add t' with guard g to $\mathcal{T}$
16   **for** $i \in I_t$ **do**
17            $g := x_n = x_i$ // Equality guard case
18            $H' := \bigvee_{f \in H} f \wedge g$ if $f \wedge g$ is satisfiable else $\bot$
19            $t' :=$ `SDTConstructor`$(H, idx + 1)$
20   Add t' with guard g to $\mathcal{T}$
21   **end**
22   **return** $\mathcal{T}$

H. Algorithm 2 proceeds in the following manner: for a symbolic action $\alpha(x_n)$ with parameter x_n, construct the *potential set* I_t (lines 6 & 7), that is, the set of parameters to which x_n has been compared to in $\mathscr{M}$. Using I_t, we can construct the disequality and equality guards as follows:

- **Disequality guard**: The disequality guard will be $g := \bigwedge_{\{i \in I_t\}} x_n \neq x_i$. We can then check which guards in H are still satisfiable with the addition of g and constructs the predicate H' for the next call of Algorithm 2 (lines 13–16).
- **Equality guard (s)**: For each parameter x_i for $i \in I_t$, the equality guard will be $g := x_n = x_i$. We can then check which guards in H are still satisfiable with the addition of g and this becomes the predicate H' for the next call of Algorithm 2 (lines 18–21).

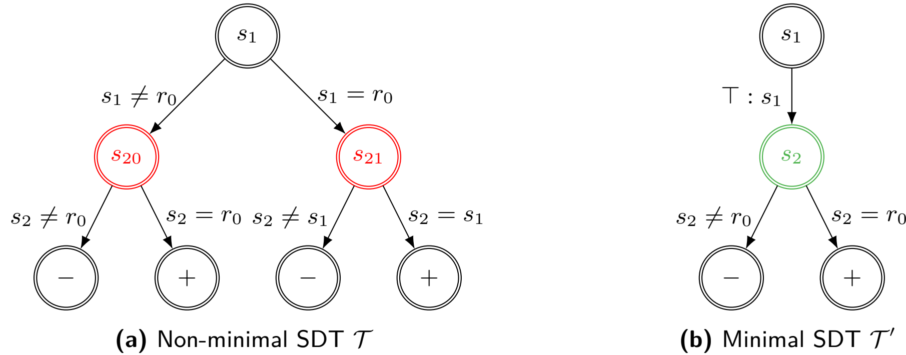
At the base case (lines $1-4$), there are no more parameters remaining and we return a non-accepting leaf if $H = \bot$, otherwise accepting. As mentioned, at each non-leaf location l of the SDT $\mathcal{T}$ returned by Algorithm 2, there exists a potential set I_t. For each parameter x_i, we know that there is a comparison between x_i and x_n in the SUT.

However, characteristic predicate H may contain redundant comparisons, making the SDT $\mathcal{T}$ bigger than necessary, creating a 'non-minimal' SDT, as shown in Figure 4a. We use Algorithm 3 to minimise each SDT $\mathcal{T}$ returned by Algorithm 2. Algorithm 3 transforms a non-minimal SDT $\mathcal{T}$ to a minimal SDT $\mathcal{T}'$ by removing all irrelevant parameters from each location of $\mathcal{T}$.

Algorithm 3: `MinimiseSDT`

Data: Non-minimal SDT $\mathcal{T}$
Result: Minimal SDT $\mathcal{T}'$
1. **if** $\mathcal{T}$ *is a leaf* **then** `// Base case`
2.   **return** $\mathcal{T}$
3. **else**
4. $\mathcal{T}' :=$ SDT node
       `// Minimise the lower levels`
5.   **for** *guard* g *with associated sub-tree* t *in* $\mathcal{T}$ **do**
6.   Add guard g with associated sub-tree `MinimiseSDT`(t) to $\mathcal{T}'$
7.   **end**
       `// Minimise the current level`
8. $I :=$ Potential set of root node of $\mathcal{T}$
9. $t' :=$ disequality sub-tree of $\mathcal{T}$ with guard $\bigwedge_{i \in I} x_n \neq x_i$
10. $I' := \varnothing$
11.   **for** $i \in I$ **do**
12. $t :=$ sub-tree of $\mathcal{T}$ with guard $x_n = x_i$
13.   **if** $t'\langle x_i, x_n\rangle \not\simeq t$ *or* $t'\langle x_i, x_n\rangle$ *is undefined* **then**
14. $I' := I' \cup \{i\}$
15.   Add guard $x_n = x_i$ with corresponding sub-tree t to $\mathcal{T}'$
16.   **end**
17.   Add guard $\bigwedge_{i \in I'} x_n \neq x_i$ with corresponding sub-tree t' to $\mathcal{T}'$
18.   **return** $\mathcal{T}'$

SDT Minimisation We present an example of the application of Algorithm 3, shown for the SDT of Figure 4a. Figure 4a visualises a non-minimal SDT $\mathcal{T}$, where s_{20} and s_{21} (in red) are essentially "duplicates" of each other: the sub-tree for node s_{20} is isomorphic to the sub-tree for node s_{21} under the relabelling "$s_1 = r_0$". We indicate this relabelling using the notation $\mathcal{T}[s_{20}]\langle r_0, s_1\rangle$ and the isomorphism relation under the relabelling as $\mathcal{T}[s_{20}]\langle r_0, s_1\rangle \simeq \mathcal{T}[s_{21}]$. Algorithm 3 accepts the non-minimal SDT of Figure 4a and produces an equivalent minimal SDT, shown in Figure 4b. Nodes s_{20} and s_{21} are merged into one node,



(a) Non-minimal SDT $\mathcal{T}$

(b) Minimal SDT $\mathcal{T}'$

Fig. 4: Minimisation algorithm in action: The non-minimal SDT on the left contains redundant decision points (red nodes), which are not present in the minimal SDT on the right (green nodes).

s_2, marked in green. We can observe that both SDTs still encode the same decision tree. SL^* requires such 'minimal' SDTs. With Algorithm 3, we have completed our tainted tree oracle, and can now proceed to the tainted equivalence oracle.

4.2 Tainted Equivalence Oracle

The tainted equivalence oracle, similar to its non-tainted counterpart, accepts a hypothesis $\mathcal{H}$ and verifies whether $\mathcal{H}$ is equivalent to register automaton $\mathscr{M}$ that models the SUT or not. If $\mathcal{H}$ and $\mathscr{M}$ are equivalent, the oracles returns a reply "yes", otherwise it returns "no" together with a counterexample.

Definition 4 (Tainted Equivalence Oracle). *For a given hypothesis $\mathcal{H}$ and an SUT $\mathcal{S}$, a tainted equivalence oracle is a function $\mathcal{O}_{\mathcal{E}}(\mathcal{H}, n, \mathcal{S})$ for all tainted traces w of $\mathcal{S}$ where $|w| \leq n$, $\mathcal{O}_{\mathcal{E}}(\mathcal{H}, n, \mathcal{S})$ returns w if $w \in \mathcal{L}(\mathcal{H}) \uplus \mathcal{L}(\mathcal{S})$, and 'Yes' otherwise.*

The RandomWalk Equivalence Oracle as implemented in RALib — like the name suggests — essentially walks randomly through an SUT in order to find a CE. Our Tainted Equivalence Oracle, in comparison, can compute *which* guards have been triggered in the SUT and can then 'explore' the SUT by triggering *other* guards. Example 6 presents a scenario of a combination lock automaton that can be learned (relatively easily) using a tainted equivalence oracle and cannot be handled by normal oracles.

Example 6 (Combination Lock RA). A combination lock RA is a type of RA which requires a *sequence* of specific inputs to perform a particular action.

Figure 5 presents an RA $\mathcal{C}$: '4-digit' combination lock, which requires a specific sequence of actions to unlock: the sequence $w = \alpha(c_0)\alpha(c_1)\alpha(c_2)\alpha(c_3)$ unlocks the automaton, where $\{c_0, c_1, c_2, c_3\}$ are constants. Consider a case where a

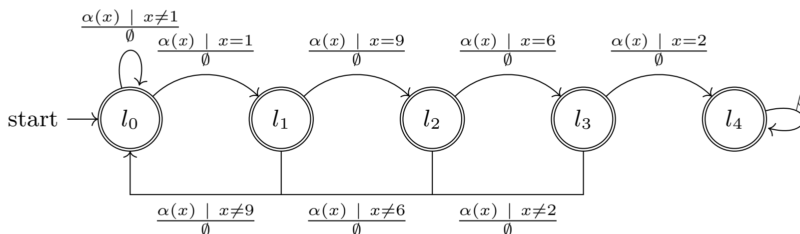


Fig. 5: Combination Lock $\mathcal{C}$: Sequence $\alpha(1)\alpha(9)\alpha(6)\alpha(2)$ *unlocks* the automaton. Error transitions (from l_3 – l_1 to l_0) are distinct, but have been 'merged' for conciseness. A sink state l_5 has not been drawn, as well as transitions to the sink state that are required to make the register automaton non-blocking.

hypothesis $\mathcal{H}$ is being checked for equivalence against the RA $\mathcal{C}$ with $w \notin \mathcal{L}(\mathcal{H})$. While would be difficult for an untainted equivalence oracle to generate the word w randomly; the tainted equivalence oracle will record at every location the comparison of input datavalue x with some constant c_i and explore all corresponding guards at the location, eventually constructing the word w.

If the 'depth' of the combination lock is not deep, then an untainted equivalence oracle may be able to find a CE, however, as the depth would increase, the probability of finding a CE would decrease as well.

⌟

5 Experimental Evaluation

We have used stubbed versions of the FIFO-Queue and Set modules[5] from Python for learning the FIFO and Set models. Additionally, the Combination Lock automata were constructed manually. Source code for all other models was obtained by translating existing benchmarks from [19] (see also `automata.cs.ru.nl`) to Python code. Furthermore, each experiment was repeated 30 times with different random seeds. Each experiment was bounded according to the following:

- Learning Phase: 10^9 inputs and 5×10^7 resets,
- Testing Phase: 10^9 inputs and 5×10^4 resets,
- Maximum depth of testing (i.e., length of the longest word during testing): 50, and
- Timeout (for the learner to respond): 10 minutes.

Figure 6 gives an overview of our experimental results. We use the notation 'TTO' to represent 'Tainted Tree Oracle' and 'TEO' to represent 'Tainted Equivalence Oracle' (with similar labels for the untainted oracles). In the figure, we

[5] From Python's `queue` module and standard library, respectively.

can see that as the size of the container increases, the difference between the fully tainted version (TTO+TEO, in blue) and the completely untainted version (NTO+NEO, in red) increases. In the case where only a tainted tree oracle is used (TTO+NEO, in green), we see that it is following the fully tainted version closely (for the FIFO models) and is slightly better in the case of the SET models.

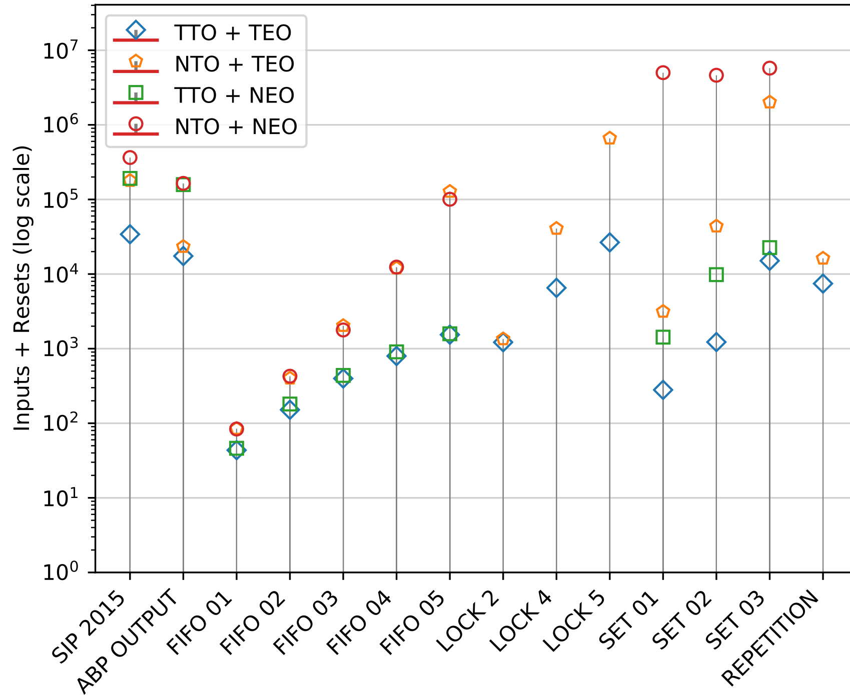


Fig. 6: Benchmark plots: Number of symbols used with tainted versions of the oracles (blue and green) are generally *lower* than with normal oracles (red and orange). Note that the y-axis is log-scaled. Additionally, normal oracles are unable to learn the Combination Lock and Repetition automata and are hence not plotted.

The addition of the tainted equivalence oracle by itself cannot be shown to have a conclusive advantage for the benchmarks we have considered. However, the addition of the tainted tree oracle by itself results in significantly fewer number of symbols, even without the tainted equivalence oracle (TTO v/s NTO, compare the green and red lines). The TTO+TEO combination does not provide vastly better results in comparison to the TTO+NEO results, however, it is still (slightly) better.

We note that — as expected — the NEO does not manage to provide CEs for the Repetition and Combination Lock automata. The TEO is therefore much more useful for finding CEs in SUTs which utilise constants. For complete details of the data used to produce the plots, please refer to Appendix B.

6 Conclusions and Future Work

In this article, we have presented an integration of dynamic taint analysis, a white-box technique for tracing data flow, and register automata learning, a black-box technique for inferring behavioural models of components. The combination of the two methods improves upon the state-of-the-art in terms of class of systems for which models can be generated and in terms of performance: Tainting makes it possible to infer data-flow constraints even in instances with a high essential complexity (e.g., in the case of so-called combination locks). Our implementation outperforms pure black-box learning by two orders of magnitude with a growing impact in the presence of multiple data parameters and registers. Both improvements are important steps towards the applicability of model learning in practice as they will help scaling to industrial use cases.

At the same time our evaluation shows the need for further improvements: Currently, the SL^* algorithm uses symbolic decision trees and tree queries globally, a well-understood weakness of learning algorithms that are based on observation tables. It also uses individual tree oracles each type of operation and relies on syntactic equivalence of decision trees. A more advanced learning algorithm for extended finite state machines will be able to consume fewer tree queries, leverage semantic equivalence of decision trees. Deeper integration with white-box techniques could enable the analysis of many (and more involved) operations on data values.

Acknowledgement We are grateful to Andreas Zeller for explaining the use of tainting for dynamic tracking of constraints, and to Rahul Gopinath for helping us using his library for tainting Python programs.

References

[1] Aarts, F., Heidarian, F., Kuppens, H., Olsen, P., Vaandrager, F.: Automata learning through counterexample-guided abstraction refinement. In: Giannakopoulou, D., Méry, D. (eds.) 18th International Symposium on Formal Methods (FM 2012), Paris, France, August 27-31, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7436, pp. 10–27. Springer (Aug 2012), `http://dx.doi.org/10.1007/978-3-642-32759-9_4`

[2] Aarts, F., Jonsson, B., Uijen, J., Vaandrager, F.: Generating models of infinite-state communication protocols using regular inference with abstraction. Formal Methods in System Design 46(1), 1–41 (2015), `http://dx.doi.org/10.1007/s10703-014-0216-x`

[3] Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. 75(2), 87–106 (Nov 1987), https://doi.org/10.1016/0890-5401(87)90052-6

[4] Botinčan, M., Babić, D.: Sigma*: Symbolic learning of input-output specifications. In: Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 443–456. POPL '13, ACM, New York, NY, USA (2013), http://doi.acm.org/10.1145/2429069.2429123

[5] Cassel, S., Howar, F., Jonsson, B., Steffen, B.: Active learning for extended finite state machines. Formal Asp. Comput. 28(2), 233–263 (2016), http://dx.doi.org/10.1007/s00165-016-0355-5

[6] Cassel, S., Howar, F., Jonsson, B., Steffen, B.: Learning extended finite state machines. In: Giannakopoulou, D., Salaün, G. (eds.) Software Engineering and Formal Methods. pp. 250–264. Springer International Publishing, Cham (2014)

[7] Cassel, S., Howar, F., Jonsson, B., Steffen, B.: Active learning for extended finite state machines. Formal Aspects of Computing 28(2), 233–263 (Apr 2016), https://doi.org/10.1007/s00165-016-0355-5

[8] Cho, C.Y., Babić, D., Poosankam, P., Chen, K.Z., Wu, E.X., Song, D.: Mace: Model-inference-assisted concolic exploration for protocol and vulnerability discovery. In: Proceedings of the 20th USENIX Conference on Security. pp. 10–10. SEC'11, USENIX Association, Berkeley, CA, USA (2011), http://dl.acm.org/citation.cfm?id=2028067.2028077

[9] Fiterău-Broştean, P., Janssen, R., Vaandrager, F.: Combining model learning and model checking to analyze TCP implementations. In: Chaudhuri, S., Farzan, A. (eds.) Proceedings 28th International Conference on Computer Aided Verification (CAV'16), Toronto, Ontario, Canada. Lecture Notes in Computer Science, vol. 9780, pp. 454–471. Springer (2016), http://www.sws.cs.ru.nl/publications/papers/fvaan/FJV16/

[10] Fiterău-Broştean, P., Howar, F.: Learning-based testing the sliding window behavior of TCP implementations. In: Petrucci, L., Seceleanu, C., Cavalcanti, A. (eds.) Critical Systems: Formal Methods and Automated Verification - Joint 22nd International Workshop on Formal Methods for Industrial Critical Systems - and - 17th International Workshop on Automated Verification of Critical Systems, FMICS-AVoCS 2017, Turin, Italy, September 18-20, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10471, pp. 185–200. Springer (2017)

[11] Fiterău-Broştean, P., Lenaerts, T., Poll, E., Ruiter, J.d., Vaandrager, F., Verleg, P.: Model learning and model checking of SSH implementations. In: Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software. pp. 142–151. SPIN 2017, ACM, New York, NY, USA (2017), http://doi.acm.org/10.1145/3092282.3092289

[12] Giannakopoulou, D., Rakamarić, Z., Raman, V.: Symbolic learning of component interfaces. In: Proceedings of the 19th International Conference on Static Analysis. pp. 248–264. SAS'12, Springer-Verlag, Berlin, Heidelberg (2012)

[13] Gopinath, R., Mathis, B., Höschele, M., Kampmann, A., Zeller, A.: Sample-free learning of input grammars for comprehensive software fuzzing. CoRR abs/1810.08289 (2018), http://arxiv.org/abs/1810.08289

[14] Hagerer, A., Margaria, T., Niese, O., Steffen, B., Brune, G., Ide, H.D.: Efficient regression testing of CTI-systems: Testing a complex call-center solution. Annual review of communication, Int.Engineering Consortium (IEC) 55, 1033–1040 (2001)

[15] Howar, F., Isberner, M., Steffen, B., Bauer, O., Jonsson, B.: Inferring semantic interfaces of data structures. In: ISoLA (1): Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 5th International Symposium, ISoLA 2012, Heraklion, Crete, Greece, October 15-18, 2012, Proceedings, Part I. Lecture Notes in Computer Science, vol. 7609, pp. 554–571. Springer (2012)

[16] Howar, F., Giannakopoulou, D., Rakamarić, Z.: Hybrid learning: Interface generation through static, dynamic, and symbolic analysis. In: Proceedings of the 2013 International Symposium on Software Testing and Analysis. pp. 268–279. ISSTA 2013, ACM, New York, NY, USA (2013), http://doi.acm.org/10.1145/2483760.2483783

[17] Howar, F., Jonsson, B., Vaandrager, F.W.: Combining black-box and white-box techniques for learning register automata. In: Steffen, B., Woeginger, G.J. (eds.) Computing and Software Science - State of the Art and Perspectives, Lecture Notes in Computer Science, vol. 10000, pp. 563–588. Springer (2019), https://doi.org/10.1007/978-3-319-91908-9_26

[18] Howar, F., Steffen, B.: Active automata learning in practice. In: Bennaceur, A., Hähnle, R., Meinke, K. (eds.) Machine Learning for Dynamic Software Analysis: Potentials and Limits: International Dagstuhl Seminar 16172, Dagstuhl Castle, Germany, April 24-27, 2016, Revised Papers. pp. 123–148. Springer International Publishing (2018)

[19] Neider, D., Smetsers, R., Vaandrager, F., Kuppens, H.: Benchmarks for Automata Learning and Conformance Testing, pp. 390–416. Springer International Publishing, Cham (2019), https://doi.org/10.1007/978-3-030-22348-9_23

[20] Schuts, M., Hooman, J., Vaandrager, F.: Refactoring of legacy software using model learning and equivalence checking: an industrial experience report. In: Ábrahám, E., Huisman, M. (eds.) Proceedings 12th International Conference on integrated Formal Methods (iFM), Reykjavik, Iceland, June 1-3. Lecture Notes in Computer Science, vol. 9681, pp. 311–325 (2016)

[21] Vaandrager, F.: Model learning. Commun. ACM 60(2), 86–95 (Feb 2017), http://doi.acm.org/10.1145/2967606

Appendix A Tree Oracle for Equalities

In this appendix, we prove that the tainted tree oracle generates symbolic decision trees (SDTs) which are isomorphic to the SDTs generated by the normal tree oracle as defined by Cassel et al. [7]. In order to do so, we first introduce the constructs used by Cassel et al. [7] for generating SDTs. We begin with some preliminaries:

For a word w with $\mathit{Vals}(w) = d_1 \ldots d_k$, we define a *potential* of w. The potential of w, written as $\mathit{pot}(w)$, is the set of indices $i \in \{1, \ldots, k\}$ for which there exists no $j \in \{1, \ldots, k\}$ such that $j > i$ and $d_i = d_j$. The concept of potential essentially allows unique access to a data value without duplication, abstracting from the concrete position of a data value in a word. Cassel et al. [7] use the concept of a *representative data value* during the construction of SDTs. For a word u with $\mathit{Vals}(u) = d_1, \ldots, d_k$, let μ_u be the valuation of $\{x_1, \ldots, x_k\}$ s.t. $\mu_u(x_i) = d_i$. A *u-guard* (referred to as 'guard' in the previous text) is a predicate g over $\{x_1, \ldots, x_k\} \cup \{p\}$. A (unique) data value d_u^g is a representative data value if $\mu_u \cup \{(p, d_u^g)\} \models g$, i.e., such that the u-guard g is satisfied under the valuation that extends μ_u by mapping p to d_u^g. Representative data values are used when constructing SDTs from (concrete) tests.

We may now define an SDT:

Definition 5 (Symbolic Decision Tree). *A Symbolic Decision Tree (SDT) is a register automaton $\mathcal{T} = (L, l_0, \mathcal{X}, \Gamma, \lambda)$ where L and Γ form a tree rooted at l_0.*

For location l of SDT $\mathcal{T}$, we write $\mathcal{T}[l]$ to denote the subtree of $\mathcal{T}$ rooted at l. An SDT that results from a tree query (u, w) (of a prefix word u and a symbolic suffix w), is required to satisfy some canconical form, captured by the following definition.

Definition 6 ((u, w)-tree). *For any data word u with k actions and any symbolic suffix w, a (u, w)-tree is an SDT $\mathcal{T}$ which has runs over all data words in $[\![w]\!]$, and which satisfies the following restriction: whenever $\langle l, \alpha(p), g, \pi, l' \rangle$ is the jth transition on some path from l_0, then for each $x_i \in \mathcal{X}(l')$ we have either* (i) *$i < k + j$ and $\pi(x_i) = x_i$, or* (ii) *$i = k + j$ and $\pi(x_i) = p$.*

If $u = \alpha(d_1) \cdots \alpha_k(d_k)$ is a data word then ν_u is the valuation of $\{x_1, \ldots, x_k\}$ satisfying $\nu_u(x_i) = d_i$, for $1 \leq i \leq k$. Using this definition, the notion of a *tree oracle*, which accepts tree queries and returns SDTs, can be described as follows.

Definition 7 (Tree Oracle). *A tree oracle for a structure $\mathcal{S}$ is a function $\mathcal{O}$, which for a data language $\mathcal{L}$, prefix word u and symbolic suffix w returns a (u, w)-tree $\mathcal{O}(\mathcal{L}, u, w)$ s.t. for any word $v \in [\![w]\!]$, the following holds: v is accepted by $\mathcal{O}(\mathcal{L}, u, w)$ under ν_u iff $u \cdot v \in \mathcal{L}$.*

Cassel et al. [7] require their (equality trees) SDTs to be *maximally abstract*, i.e., the SDTs must not contain any redundancies (such as Figure 4a): an element $i \in I$ iff i affects the behaviour of the current parameter. In order to convert

an SDT to a maximally abstract SDT, we require a procedure to minimise a (sub)-tree, called *specialisation of equality tree*:

Definition 8 (Specialisation of equality tree). *Let $\mathcal{T}$ be an equality tree for prefix u and set of symbolic suffixes V, and let $J \subseteq pot(u)$ be a set of indices. Then $\mathcal{T}\langle J \rangle$ denotes the equality tree for (u, V) obtained from $\mathcal{T}$ by performing the following transformations for each α:*

- *Whenever $\mathcal{T}$ has several initial α-transitions of form $\langle l_0, \alpha(p), (p = x_j), l_j \rangle$ with $j \in J$, then all subtrees of form $(\mathcal{T}[l_j])\langle J[(k+1) \mapsto j] \rangle$ for $j \in J$ must be defined and isomorphic, otherwise $\mathcal{T}\langle J \rangle$ is undefined. If all such subtrees are defined and isomorphic, then $\mathcal{T}\langle J \rangle$ is obtained from $\mathcal{T}$ by*
  1. *replacing all initial α-transitions of form $\langle l_0, \alpha(p), (p = x_j), l_j \rangle$ for $j \in J$ by the single transition $\langle l_0, \alpha(p), (p = x_m), l_m \rangle$ where $m = \max(J)$,*
  2. *replacing $\mathcal{T}[l_m]$ by $(\mathcal{T}[l_m])\langle J[(k+1) \mapsto m] \rangle$, and*
  3. *replacing all other subtrees $\mathcal{T}[l']$ reached by initial α-transitions (which have not been replaced in Step 1 by $(\mathcal{T}[l'])\langle J \rangle$.*

If, for some α, any of the subtrees generated in Step 2 or 3 are undefined, then $\mathcal{T}\langle J \rangle$ is also undefined, otherwise $\mathcal{T}\langle J \rangle$ is obtained after performing Steps $1 - 3$ for each α. Definition 8 is then used to define an equality tree $\mathcal{O}_{\mathcal{L}}(u, V)$.

Definition 9 (Necessary Potential set for Tree Oracle). *A necessary potential set I for the root location l_0 of an equality tree $\mathcal{O}(\mathcal{L}, u, V)$ is a subset of $pot(u)$ — the potential of the prefix u — such that for each index $i \in I$ the following holds:*

1. *$\mathcal{O}(\mathcal{L}, u\alpha(d_u^0), V_\alpha)\langle \{i, k+1\} \rangle$ is undefined, or*
2. *$\mathcal{O}(\mathcal{L}, u\alpha(d_u^0), V_\alpha)\langle \{i, k+1\} \rangle \not\simeq \mathcal{O}(\mathcal{L}, u\alpha(d_i), V_\alpha)$: i.e., the specialised sub-tree is not isomorphic to the original sub-tree.*

Intuitively, a necessary potential set contains indices of data values which influence future behaviour of the SUT. Consequently, indices of data values which do not influence the behaviour of the SUT are excluded from the necessary potential set. We use a special variant of a (u, V)-tree, called an *equality tree*:

Definition 10 (Equality Tree). *An* equality tree *for a tree query (u, V) is a (u, V)-tree $\mathcal{T}$ such that:*

- *for each action α, there is a potential set $I \subseteq pot(u)$ of indices such that the initial α-guards consist of the equalities of form $p = x_i$ for $i \in I$ and one disequality of form $\wedge_{i \in I} p \neq x_i$, and*
- *for each initial transition $\langle l_0, \alpha(p), g, l \rangle$ of $\mathcal{T}$, the tree $\mathcal{T}[l]$ is an equality tree for $(u\alpha(d_u^g), \alpha^{-1}V)$.*

Definition 11 (Tree oracle for equality). *For a language $\mathcal{L}$, a prefix u, and the set of symbolic suffixes V, the equality tree $\mathcal{O}(\mathcal{L}, u, V)$ is constructed as follows:*

– *If* $V = \{\epsilon\}$, *then* $\mathcal{O}(\mathcal{L}, u, \{\epsilon\})$ *is the trivial tree with one location* l_0 *and no registers. It is accepting if the word is accpeted, i.e.,* $\lambda(l_0) = +$ *if* $u \in \mathcal{L}$, *else* $\lambda(l_0) = -$. *To determine* $u \in \mathcal{L}$, *the tree oracle performs a membership query on* u.
– *If* $V \neq \{\epsilon\}$, *then for each* α *such that* $V_\alpha = \alpha^{-1}V$ *is non-empty,*
  - *let* I *be the necessary potential set (Definition 9),*
  - $\mathcal{O}(\mathcal{L}, u, V)$ *is constructed as* $\mathcal{O}(\mathcal{L}, u, V) = (L, l_0, \Gamma, \lambda)$, *where, letting* $\mathcal{O}(\mathcal{L}, u\alpha(d_i), V_\alpha)$ *be the tuple* $(L_i^\alpha, l_{0i}^\alpha, \Gamma_i^\alpha, \lambda_i^\alpha)$ *for* $i \in (I \cup \{0\})$,
    * L *is the disjoint union of all* L_i^α *plus an additional initial location* l_0,
    * Γ *is the union of all* Γ_i^α *for* $i \in (I \cup \{0\})$, *and in addition the transitions of form* $\langle l_0, \alpha(p), g_i, l_{0i}^\alpha \rangle$ *with* $i \in (I \cup \{0\})$, *where* g_i *is* $\bigwedge_{j \in I} p \neq x_j$ *for* $i = 0$, *and* g_i *is* $p = x_i$ *for* $i \neq 0$, *and*
    * λ *agrees with each* λ_i^α *on* L_i^α. *Moreover, if* $\epsilon \in V$, *then* $\lambda(l_0) = +$ *if* $u \in \mathcal{L}$, *otherwise* $\lambda(l_0) = -$. *Again, to determine whether* $u \in \mathcal{L}$, *the tree oracle performs a membership query for* u.

Intuitively, $\mathcal{O}(\mathcal{L}, u, V)$ is constructed by joining the trees $\mathcal{O}(\mathcal{L}, u\alpha(d_i), V_\alpha)$ with guard $p = x_i$ for $i \in I$, and the tree $\mathcal{O}(\mathcal{L}, u\alpha(d_u^0), V_\alpha)$ with guard $\bigwedge_{i \in I} p \neq x_i$, as children of a new root. Note, while V is a set of symbolic suffixes, RALib technically handles tree queries sequentially, i.e., as sequential tree queries of prefix u and symbolic suffix w. Consequently, we treat the set of symbolic suffixes V as a singleton, referred to as 'w'.

$\mathcal{O}(\mathcal{L}, u, w)$ is constructed bottom-up, recursively building new 'roots' at the top with larger and larger symbolic suffixes (and consequently, shorter and shorter prefixes). The choice of the necessary potential set I plays a crucial role: if I is larger than necessary, $\mathcal{O}(\mathcal{L}, u, w)$ contains redundant guards (and is hence a 'non-minimal' SDT).

We now have a clear goal for our proof: we must show that the SDT returned by Algorithm 3 is isomorphic to the SDT as defined in Definition 11 (under the assumption that the 'set' of symbolic suffixes V is a singleton). We can divide our proof into the following steps:

1. We show that Algorithm 1 produces a characteristic predicate for tree query (u, w), and contains all the information for constructing an equality tree,
2. Next, we show that Algorithm 2 guarantees that for potential set I_t of a location l_t of the tainted equality tree $\mathcal{T}_t$, the potential set I of equivalent location l of the normal equality tree $\mathcal{T}$ is a subset: $I \subseteq I_t$, and finally,
3. We can then reduce the make the tainted potential set equal to the normal potential set ($I_t = I$) and the resulting tainted equality tree will be isomorphic to the normal equality tree.

Each of the above steps correspond to one of our algorithms. We now begin with step 1: from Algorithm 1, we can state the following lemmas:

Lemma 1 (Characteristic Predicate). *For a tree query* (u, w), *Algorithm 1 always produces a characteristic predicate* H.

Proof. We recall that, under the test hypothesis, an SUT $\mathscr{M}$ is deterministic and has a finite number of logically disjoint branches to be followed from each state. Algorithm 1 initialises two variables $G := \top$ and $H := \bot$. For each word $z = u \cdot w$ under a valuation $\nu \vDash G$, we may perform a membership query on $\mathscr{M}$. Each query returns the guard $I = \wedge_{i=k+1}^{k+n} constraints_{\mathscr{M}}(z)[i]$ such that $\nu \vDash I$ and the acceptance of the word z in the language of $\mathscr{M}$, i.e., $z \in \mathscr{M}$.

For each iteration of the do-while loop, the variable G is updated with the negation of the previously satisfied guard I, i.e., $G := G \wedge \neg I$. This guarantees that any new valuation $\nu' \vDash G$ will not satisfy I, and hence, the next iteration of the do-while loop shall induce a different run of $\mathscr{M}$. Given that $\mathscr{M}$ only has a finite number of logical branches, Algorithm 1 terminates (and G contains a finite number of sub-terms).

We also know that for each tainted word z, we obtain the acceptance of $z \in L(\mathscr{M})$. If $z \in L(\mathscr{M})$, the variable H is updated to $H \vee I$. Therefore, the predicate H returned by Algorithm 1 is the characteristic predicate for the tree query (u, w). □

After constructing the characteristic predicate, we convert it to a non-minimal SDT using Algorithm 2, providing us with the following lemma:

Lemma 2 (Non-minimal SDT). *For any location l_t of a non-minimal SDT with an equivalent location l of a minimal SDT, the necessary potential set I_t of the non-minimal SDT is a superset of the necessary potential set I of the minimal SDT: $I \subseteq I_t \subseteq pot(u)$ where $pot(u)$ is the potential of the prefix u of locations l_t and l.*

Proof. We know that $I \subseteq pot(u)$ and $I_t \subseteq pot(u)$ by definition of the necessary potential set. For any word $w = u \cdot v$ where the prefix u leads to location l_t of the tainted non-minimal SDT, Algorithm 2 guarantees that the suffixes of u will be classified correctly. If the suffixes are classified correctly, we know that $I_t \supseteq I$ (otherwise the suffixes will not be classified correctly). Since $I_t \supseteq I$ and $I, I_t \subseteq pot(u)$, we conclude $I \subseteq I_t \subseteq pot(u)$. □

Following Lemma 2, if we wish to make $I = I_t$, we can simply remove all elements from I_t which do not satisfy the conditions outlined in Definition 9. Since we already know that $I \subseteq I_t$, we can confirm that after removal of all irrelevant parameters, $I = I_t$. Algorithm 3 accomplishes the same.

Cassel et al. [7] use the concept of representative data values for constructing the SDT, while we treat the values symbolically: a representative data value 'represents' the set of datavalues that satisfy a u-guard during construction of the SDT; in our case, we simply let `Z3` decide on all the values to use for our membership queries and obtain the guards about them using their taint markers as identifiers.

Theorem 1 (Isomorphism of tree oracles). *The SDTs generated by the tainted tree oracle and the untainted tree oracle for a tree query (u, w) are isomorphic.*

Proof. Lemma 1 guarantees that Algorithm 1 returns a characteristic predicate H for the tree query (u, w). Application of Algorithm 2 on H constructs a non-minimal SDT. Using Lemma 2 and Algorithm 3 on the non-minimal SDT, we may conclude that the root locations of the tainted tree oracle and normal tree oracle have the same necessary potential set. By inductive reasoning on the depth of the trees, the same holds for all sub-trees of both oracles, eventually reducing to the leaves, showing that the tainted tree oracle is isomorphic to tree oracle. □

Appendix B Detailed Benchmark results

Table 1 contains the full results of the values used to create the plots from Figure 6.

Table 1: Benchmarks

Model	Tree Oracle	EQ Oracle	Learn Symbols (Std. Dev)	Test Symbols (Std. Dev)	Total Symbols (Std. Dev)	Learned
Abp Output	Tainted	Normal	6.55E+02 (8.33E+01)	1.57E+05 (1.29E+05)	1.58E+05 (1.29E+05)	30/30
Abp Output	Tainted	Tainted	6.17E+02 (7.78E+01)	1.68E+04 (1.15E+04)	1.74E+04 (1.15E+04)	30/30
Abp Output	Normal	Normal	6.93E+03 (5.20E+03)	1.57E+05 (1.29E+05)	1.64E+05 (1.29E+05)	30/30
Abp Output	Normal	Tainted	6.51E+03 (3.97E+03)	1.68E+04 (1.15E+04)	2.33E+04 (1.29E+04)	30/30
Lock 2	Tainted	Normal	N-A (N-A)	N-A (N-A)	N-A (N-A)	0/30
Lock 2	Tainted	Tainted	7.10E+01 (0.00E+00)	1.15E+03 (6.76E+02)	1.22E+03 (6.76E+02)	30/30
Lock 2	Normal	Normal	N-A (N-A)	N-A (N-A)	N-A (N-A)	0/30
Lock 2	Normal	Tainted	2.00E+02 (0.00E+00)	1.15E+03 (6.76E+02)	1.35E+03 (6.76E+02)	30/30
Lock 4	Tainted	Normal	N-A (N-A)	N-A (N-A)	N-A (N-A)	0/30
Lock 4	Tainted	Tainted	2.41E+02 (0.00E+00)	6.29E+03 (5.52E+03)	6.53E+03 (5.52E+03)	30/30
Lock 4	Normal	Normal	N-A (N-A)	N-A (N-A)	N-A (N-A)	0/30
Lock 4	Normal	Tainted	3.45E+04 (0.00E+00)	6.29E+03 (5.52E+03)	4.08E+04 (5.52E+03)	30/30
Lock 5	Tainted	Normal	N-A (N-A)	N-A (N-A)	N-A (N-A)	0/30
Lock 5	Tainted	Tainted	3.80E+02 (0.00E+00)	2.62E+04 (1.45E+04)	2.66E+04 (1.45E+04)	30/30
Lock 5	Normal	Normal	N-A (N-A)	N-A (N-A)	N-A (N-A)	0/30
Lock 5	Normal	Tainted	6.35E+05 (0.00E+00)	2.62E+04 (1.45E+04)	6.61E+05 (1.45E+04)	30/30

Continued on next page

Table 1: Benchmarks

Model	Tree Oracle	EQ Oracle	Learn Symbols (Std. Dev)	Test Symbols (Std. Dev)	Total Symbols (Std. Dev)	Learned
Fifo 01	Tainted	Normal	2.90E+01 (4.08E+00)	1.71E+01 (6.12E+00)	4.62E+01 (6.73E+00)	30/30
Fifo 01	Tainted	Tainted	2.97E+01 (3.83E+00)	1.38E+01 (3.58E+00)	4.35E+01 (4.93E+00)	30/30
Fifo 01	Normal	Normal	6.65E+01 (1.84E+01)	1.71E+01 (6.12E+00)	8.37E+01 (1.80E+01)	30/30
Fifo 01	Normal	Tainted	7.07E+01 (1.74E+01)	1.38E+01 (3.58E+00)	8.46E+01 (1.68E+01)	30/30
Fifo 02	Tainted	Normal	1.16E+02 (3.26E+01)	6.47E+01 (2.77E+01)	1.81E+02 (4.28E+01)	30/30
Fifo 02	Tainted	Tainted	1.01E+02 (3.03E+01)	5.10E+01 (1.55E+01)	1.52E+02 (3.31E+01)	30/30
Fifo 02	Normal	Normal	3.62E+02 (1.29E+02)	6.47E+01 (2.77E+01)	4.27E+02 (1.33E+02)	30/30
Fifo 02	Normal	Tainted	3.50E+02 (1.48E+02)	5.10E+01 (1.55E+01)	4.01E+02 (1.49E+02)	30/30
Fifo 03	Tainted	Normal	3.03E+02 (8.53E+01)	1.34E+02 (5.84E+01)	4.38E+02 (9.39E+01)	30/30
Fifo 03	Tainted	Tainted	2.93E+02 (8.54E+01)	1.05E+02 (4.69E+01)	3.98E+02 (8.07E+01)	30/30
Fifo 03	Normal	Normal	1.64E+03 (9.00E+02)	1.34E+02 (5.84E+01)	1.78E+03 (8.82E+02)	30/30
Fifo 03	Normal	Tainted	1.93E+03 (1.34E+03)	1.05E+02 (4.69E+01)	2.03E+03 (1.31E+03)	30/30
Fifo 04	Tainted	Normal	6.87E+02 (1.51E+02)	2.20E+02 (1.11E+02)	9.06E+02 (2.14E+02)	30/30
Fifo 04	Tainted	Tainted	6.35E+02 (1.41E+02)	1.62E+02 (7.53E+01)	7.96E+02 (1.53E+02)	30/30
Fifo 04	Normal	Normal	1.22E+04 (1.22E+04)	2.20E+02 (1.11E+02)	1.24E+04 (1.22E+04)	30/30
Fifo 04	Normal	Tainted	1.19E+04 (1.21E+04)	1.62E+02 (7.53E+01)	1.20E+04 (1.21E+04)	30/30
Fifo 05	Tainted	Normal	1.23E+03 (3.35E+02)	3.53E+02 (2.13E+02)	1.58E+03 (4.49E+02)	30/30
Fifo 05	Tainted	Tainted	1.32E+03 (2.88E+02)	2.24E+02 (9.79E+01)	1.54E+03 (3.14E+02)	29/30
Fifo 05	Normal	Normal	1.00E+05	3.19E+02	1.01E+05	25/30

Continued on next page

Table 1: Benchmarks

Model	Tree Oracle	EQ Oracle	Learn Symbols (Std. Dev)	Test Symbols (Std. Dev)	Total Symbols (Std. Dev)	Learned
			(1.84E+05)	(1.67E+02)	(1.84E+05)	
Fifo 05	Normal	Tainted	1.28E+05	2.35E+02	1.28E+05	25/30
			(2.08E+05)	(8.76E+01)	(2.08E+05)	
Repetition	Tainted	Normal	N-A	N-A	N-A	0/30
			(N-A)	(N-A)	(N-A)	
Repetition	Tainted	Tainted	1.22E+02	7.33E+03	7.45E+03	30/30
			(0.00E+00)	(2.03E+03)	(2.03E+03)	
Repetition	Normal	Normal	N-A	N-A	N-A	0/30
			(N-A)	(N-A)	(N-A)	
Repetition	Normal	Tainted	8.90E+03	7.33E+03	1.62E+04	30/30
			(1.99E+03)	(2.03E+03)	(2.26E+03)	
Set 01	Tainted	Normal	1.45E+02	1.28E+03	1.43E+03	29/30
			(1.03E+02)	(1.52E+03)	(1.52E+03)	
Set 01	Tainted	Tainted	9.75E+01	1.83E+02	2.80E+02	30/30
			(3.56E+01)	(1.61E+02)	(1.56E+02)	
Set 01	Normal	Normal	5.00E+06	1.28E+03	5.01E+06	29/30
			(1.73E+07)	(1.52E+03)	(1.73E+07)	
Set 01	Normal	Tainted	2.96E+03	1.83E+02	3.15E+03	30/30
			(6.71E+03)	(1.61E+02)	(6.69E+03)	
Set 02	Tainted	Normal	1.61E+03	8.21E+03	9.82E+03	28/30
			(9.96E+02)	(1.26E+04)	(1.24E+04)	
Set 02	Tainted	Tainted	1.00E+03	2.21E+02	1.23E+03	29/30
			(3.26E+02)	(2.14E+02)	(3.68E+02)	
Set 02	Normal	Normal	4.61E+06	8.60E+03	4.62E+06	25/30
			(1.43E+07)	(1.31E+04)	(1.43E+07)	
Set 02	Normal	Tainted	4.35E+04	2.20E+02	4.37E+04	30/30
			(7.28E+04)	(2.10E+02)	(7.29E+04)	
Set 03	Tainted	Normal	1.76E+04	5.01E+03	2.26E+04	24/30
			(8.71E+03)	(9.51E+03)	(1.40E+04)	
Set 03	Tainted	Tainted	1.44E+04	6.91E+02	1.51E+04	30/30
			(5.05E+03)	(8.76E+02)	(4.95E+03)	
Set 03	Normal	Normal	5.76E+06	3.94E+03	5.76E+06	14/30
			(1.47E+07)	(6.48E+03)	(1.47E+07)	
Set 03	Normal	Tainted	2.01E+06	2.23E+02	2.01E+06	28/30
			(3.60E+06)	(2.06E+02)	(3.60E+06)	
Sip 2015	Tainted	Normal	2.14E+03	1.89E+05	1.92E+05	10/30
			(4.00E+02)	(2.60E+05)	(2.60E+05)	

Continued on next page

Table 1: Benchmarks

Model	Tree Oracle	EQ Oracle	Learn Symbols (Std. Dev)	Test Symbols (Std. Dev)	Total Symbols (Std. Dev)	Learned
Sip 2015	Tainted	Tainted	2.30E+03 (3.13E+02)	3.18E+04 (1.59E+04)	3.41E+04 (1.59E+04)	29/30
Sip 2015	Normal	Normal	1.57E+05 (4.42E+05)	2.07E+05 (2.69E+05)	3.65E+05 (4.81E+05)	9/30
Sip 2015	Normal	Tainted	1.47E+05 (2.80E+05)	3.18E+04 (1.59E+04)	1.79E+05 (2.78E+05)	29/30