Bachelor thesis
Computer Science

Radboud University

Learning register automata using Taint Analysis

Author:
Timo Schrijvers
s4596331

First supervisor/assessor:
prof. dr. F.W. Vaandrager
f.vaandrager@cs.ru.nl

Second assessor:
dr. N.H. Jansen
N.Jansen@cs.ru.nl

August 21, 2018

Abstract

This thesis shows how taint analysis can be integrated in model learning and try to decrease the number of learning queries that are needed to learn register automata. We describe a prototype of a tree oracle that uses tainting to extract information from a SUL. This prototype is integrated in RALib, a tool for learning Extended Finite State Machines. We compared the original version of RALib with the one that uses our prototype. We show that tainting could be promising for model learning, but the prototype still needs some improvements and further research to actually out preform RALib.

Contents

Chapter 1

Introduction

In our daily lives we make constant use of machines. We use a microwave to heat our food, an alarm clock to wake up and washing machine to clean our clothes. These machines have a particular thing in common. They all depend on a input from outside the machine. Without this input these machines will stay in the same state, which in most cases comes down to doing nothing. As humans we intuitively learn how these machines work, when pressing some buttons and observing the output. To learn how these machines work is rather easy, but describing the exact behaviour of a machine gets more difficult. To describe the exact behaviour we need a systematic way to learn from the inputs and outputs of the machine. Model learning (described in section 2.1) provides techniques to learn the behaviour of machines by only sending inputs and analyzing outputs. This way of analyzing machines can be quite inefficient as the number of inputs needed to learn the behaviour of a machine can grow to an enormous amount, when the machines become more complex.
If we take this to software, many different kind of software components also rely on input from an external source. Take for example a Java class. We can see an object as a machine, the methods of the class as inputs and the return statements from the methods as outputs. The difference between software components and machines like the microwave is that the inputs of the microwave are physical, while the inputs of the software components are also software. We cannot change anything to the press of a button, but we can alter software inputs. Instead of just sending inputs and analyzing outputs from software it is possible to alter the inputs such that they can extract information from inside the software component. With the extracted information, learning software components should become more efficient. A technique to extract information from software components trough infected inputs is taint analysis. In this thesis we try to integrate taint analysis into model learning and test if the extraction of information increases the efficiency of model learning.

Chapter 2

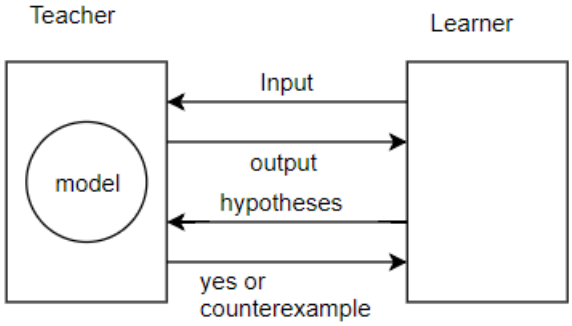
Preliminaries

2.1 Model Learning

Model learning is a technique to learn the behaviour of machines. Most machines have a predefined behaviour and react on input from outside the machine. While machines based on artificial intelligence may have an unpredictable behaviour, there is still a large set of machines that always show the same behaviour. Take for example a coffee machine. If you press certain buttons in a certain order, the machine will make a specific kind of coffee that belongs to that input. The coffee machine can be seen as a black box. You can only see what is happening on the outside. Whenever you press buttons, the coffee machine comes with a response, but what happens inside the machine will not be visible to the user. With model learning we can systematically construct models that represent the behaviour of machines. The techniques for model learning can be black box or white box. Black box techniques assume that everything what happens inside the machine is not visible, while white box techniques can extract information from within the machine. White box techniques are especially useful when modelling software components where, for example, part of the code is known.

When learning models, we assume a teacher and a learner from the Mini-

Figure 2.1: MAT framework



mally Adequate Teacher (MAT) framework [1]. The teacher directly interacts with the model and answers questions from the learner. The learner asks questions to the teacher and tries to learn the model with the given answers. There are two type of questions: membership queries and equivalence queries. A membership query is a sequence of inputs that can be fed to the model. The teacher responds on this sequence with a sequence of corresponding outputs, generated by the model. The learner stores all input output combinations in a table. When the learner has enough information it constructs a hypothetical model. The learner proposes the hypothetical model to the teacher. This is an equivalence query. The teacher compares the hypothetical model with the actual model and responds with yes or a counterexample. When the answer is yes the hypothetical model is equivalent to the original model, so the learner is done. If the answer is a counterexample, the hypothetical model is not correct yet, which means that the learner needs to ask more membership queries to construct a new hypothetical model.

2.2 Register automata

Model learning can be used to learn many different kinds of models, but we are going to focus on a specific set of models, namely the register automata. Register automata are an extension of finite automata. Finite automata, also called finite state machines, are machines that follow a predefined pattern based on a given input. They consist of a certain number of states and transitions. At any time, a finite automaton is in exactly one state. Starting in the initial state, the automaton can get into other states through transitions. Every state has zero or more transitions to other states, which are taken based on input actions. Register automata extend finite automata with registers in which data values can be stored. These values can be read from the registers for later usage. For our register automata we use definition 2.1 from [8]

Definition 2.1 (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, with $X(l_0) = \phi$, 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,
  - $l' \in L$ 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 $\{+, -\}$. □

In finite state machines without registers, states and locations are the same, but in register automata, a state is a defined by a pair $< l, i >$ where $l \in L$ and i is a valuation over $\mathcal{X}(l)$. For example, if $\mathcal{X}(l_1) = \{x_1, x_2\}$ then a state could be $< l_1, [x_1, x_2] \mapsto [3, 5] >$. The value of the registers can cause the same action to trigger different transitions, from a particular location. A parameterized symbol is an action with a data value from some infinite domain of data values, like the natural numbers or the set of strings. Actions can have an arbitrary number of parameters. We assume that Σ is a set of actions with an arity that determines how many parameters each action has. A guard is a condition that needs to hold for a transition to be taken. In a guard a parameter is compared with a register, that contains a previously stored data value. Guards can also consist of conjunctions, disjunctions or negations of multiple conditions. For each $l \in L$, the disjucntion of the guards from all transitions where l is the source location should be *true*. In the assignment the new registers from l' will be assigned a value from either the registers from l or one of the parameters.
Function λ indicates whether each location is an accepting location(+) or a rejecting location(-). Every register automaton accepts a particular data language. A data language consists of a set of data words, which again are a sequence of parameterized symbols drawn from Σ. A register automaton can either accept or reject a data word. After following all actions in a data word, the register automaton is situated in one of the locations. Only when this is an accepting location the data word is an element of the data language that is accepted by that register. We refer to [5] for more detailed information about register automata.

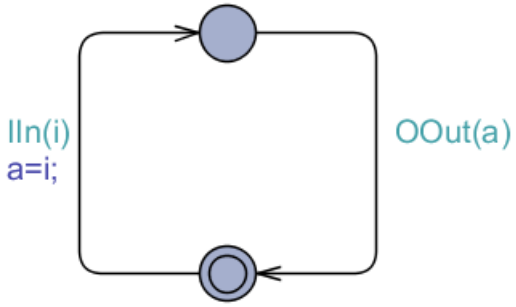
2.3 SUT

2.3.1 Uppaal

Model learning cannot be done without a SUL (System under Learning). Uppaal [2] is a tool that can model timed automata, extended with data-types. The graphical interface of Uppaal can be used to give a concrete representation of register automata.

Figure 2.2 shows a simple register automaton. This automaton receives an input IIn(i), then stores the parameter i in a and outputs the value that is stored in a with the output action OOut(a). The parameterized symbol can either be an input or an output. In our models, if a parameterized

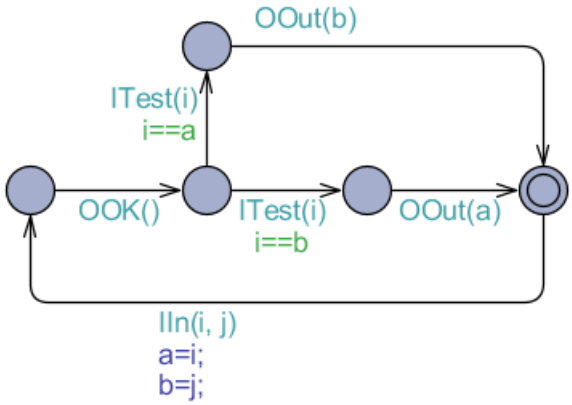
Figure 2.2: Register automaton example



symbol starts with an I, it is considered an input. If it starts with an O, it is considered an output. Input transitions always require an input from an environment, while output transitions always follow directly after an input, as an acknowledgement.

Figure 2.3 shows a register automaton with one more extension. In this

Figure 2.3: Register automaton example with guards



figure, a guard is added to the model. In this automaton the received input from IIn(i, j) is stored in registers a and b and the next input from ITest(i) will be compared to registers a and b. When i is equal to a, b will be the output parameter of OOut and when and i is equal to b, a will be the output parameter of OOut. Most of the time, non accepting transitions are not modelled. If i from ITest(i) is not equal to either a or b, then this automaton ends up in a non-accepting state.

2.3.2 SUT tool

The SUT tool is program that converts models created with Uppaal to a state machine in a programming language, which can actually handle input and output. We use the SUT tool to convert Uppaal models to a python script, but it can also convert models to java or other formats. The obtained python script can be used as a library for other python scripts or as a independent program that communicates via sockets. We use the python script as library. This library has a state machine object, that expects input symbols with a certain number of parameters one by one. If the symbol corresponds to any of the symbols that can be taken from the current state of the state machine and the number of arguments is correct, then the state transition is taken and the output symbol is returned with it's parameters. In any other case the script outputs that the input is wrong and stays in the same state.

Chapter 3

Taint analysis

3.1 Introduction

Taint analysis is a white-box technique to analyze software. Unlike black-box techniques, taint analysis keeps track of run-time information during the execution of a program. To illustrate how taint analysis works, it can be compared with an ant nest that lies underground. Only the entrances, the ants that enter and the ants that exit the nest are visible from the outside. Lets say that we color every ant that enters the nest with a different paint color. This paint spreads to anything that is touched by the colored ant. Now, if ants exit the nest and they have some colors from other ants, we know they have had contact with the colored ants. Also, when a purple ant enters the nest in one entrance and a purple ant exits the nest from another entrance, we know that those entrances are connected. If we compare this to taint analysis, the ant nest is a program, with the entering ants as inputs and the exiting ants as output. Input values can be given a mark. These values are then considered tainted values. While a program with tainted input values is running, values that are affected by those tainted values will also be tainted with the same mark. When the program outputs a value, the marks indicate which inputs affected this output.

3.2 Taint analysis tools

To apply taint analysis to our models, we need a suitable taint analysis tool. There are many tools that use taint analysis to find vulnerabilities in software. It is impossible to build a tool that can be used for every purpose. Therefore, all tools focus on specific vulnerabilities, like buffer overflows [10] or sql injections [7].
In most tools, taint analysis consists of three procedures: taint introduction, taint propagation and vulnerability detection. Every tool has its own implementation of these procedures, but most of them rely on the same basis.

During taint introduction it is decided which inputs are tainted and how they are tainted. It is possible to give a whole input file one taint mark, but you can also taint every input byte separately. Tainting smaller pieces of data results in better analysis, but also in worse performance. It is also important to choose the right size of a taint mark. Taint marks can consist of one bit that indicates whether the data value is tainted or not. It is also possible to give every piece of data an unique taint mark. More specific taint marks give more insight about the data flows in a program, but require more storage space and more difficult propagation.
After all inputs are properly tainted, the next step is taint propagation. During taint propagation, all data that is affected by tainted data will also be tainted. There are two ways in which tainted data can affect other data: through direct assignment and through control dependencies. For example, if x is tainted and the program executes the statement y=x+2, then y will gain the same taint mark as x, because the value of y is directly dependent on the value of x. Control dependencies are more difficult to propagate. For example, when a tainted value affects the guard of an if-statement, the assigned values within the body of that if-statement should also be tainted. When the execution of a program finishes or a vulnerability appears, then vulnerability detection starts. When the taint introduction and propagation steps are done correctly it is possible to track which values are dependent on which input. This way, the source of a value that triggered a vulnerability can be found.
Dynamic taint analysis tools are usually implemented on a dynamic binary analysis (DBA) tool like Valgrind [11]. DBA tools commonly use dynamic binary instrumentation (DBI). This means that analysis code is injected in the original code of a program during run-time.

3.3 Taintedstr

For our research, we do not want to directly discover vulnerabilities, but learn the behaviour of register automata. We've looked at multiple tools like [12] and [9] whether we could use the taint analysis part for our research. Most tools were not suited for learning register automata. Instead, we use a python library from the Pygmalion prototype [6]. Pygmalion is a software tool that, "given a program P without any input samples or models, learns an input grammar that represents the syntactically valid inputs for P—a grammar which can then be used for highly effective test generation for P."[6]. For this purpose Pygmalion makes use of taint analysis. The library taintedstr, that is used for taint analysis within the Pygmalion prototype, can also be used to taint state machines.
The taintedstr library mainly consists of the class tstr, which behaves like

a python string, but applies taint analysis when an instance of this class is used in a program. The tstr class is built to get a string as parameter when initialized. Every character of this input string is tainted with an integer value. Instances of tstr can be split, concatenated and even mingled with ordinary strings. If only string operations are applied on a given tstr object, every character keeps its own taint value.

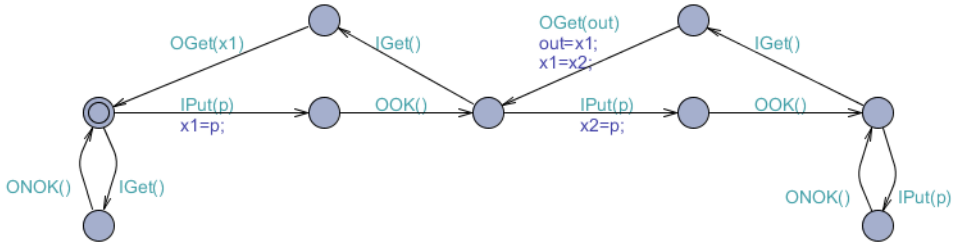
```
hello_str = tstr("Hello World!", [1, 1, 1, 1, 1, 0, 2, 2, 2, 2,
    2, 2])
split_str = hello_str.split(" ")
merged_str = split_str[0] + " World!"
print(merged_str,  merged_str._taint )
```

The code snippet above shows how tstr works. In the first line, a tstr object is initialized with the string "Hello World!". The first five characters are tainted with a value of 1, the following space with a value of 0 and the last six characters with a value of 2. If the list of taint values is not provided, the first character will get a taint value of 0, the second character a taint value of 1 and so on. After applying the split operation on the tstr object, split_str has two elements. The first is a tstr object that contains the string "Hello" where all characters are tainted with the value 1. The second is a tstr object that contains the string "World!" where all characters are tainted with the value 2. The third line concatenates " World!" to the first element of split_str. The fourth line will then print: "Hello World! [1, 1, 1, 1, 1, -1, -1, -1, -1, -1, -1, -1]". Calling _taint from a tstr object will return a list, representing the taint values of that tstr object. The five 1 values are the taint values of the original "Hello" from hello_str and the seven -1 values are the taint values of the new " World!" string. All characters that are concatenated to a tstr object are tainted with the value -1, which means that they are not tainted. The tstr class also has an attribute named 'comparisons'. This attribute contains a list of all comparisons that are invoked on a particular tstr object. Each element of the list contains the two values that are compared, the corresponding taint values and the comparison method.
Some software components can be modelled by a SUL. Before this is possible, a component needs to satisfy some requirements. First, values can only be directly compared with other value or copied from one register to another. This means that values cannot be changed. Secondly, the component must have a finite number of states. Because we apply taint analysis on components that satisfy these requirements, every input value can consist of exactly one tstr object with one taint value. For example, if you want an input to be the string "hello", then every character can be tainted with 1, which causes the taint value to be [1, 1, 1, 1, 1]. We use colors as taint values to make examples more readable and easier to understand. So for example, when the string "hello" has a taint value [1, 1, 1, 1, 1] we say that the taint value is red.

3.4 Tainting input and output

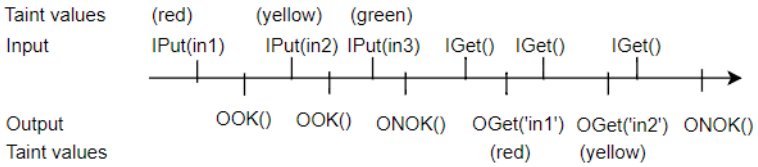
With the taintedstr library, taint analysis can be applied to a SUL created by the SUT tool described in section 2.3.2. The model in Figure 3.1 is a queue that can contain a maximum of 2 elements. This model has two input actions, namely IPut(p) and IGet(). IPut(p) tries to store parameter p into the queue and IGet() tries to get the front element from the queue.

Figure 3.1: Queue



First we created three tstr objects with the strings "in1", "in2" and "in3" with taint values red, yellow and green respectively. The three tstr objects are called in1, in2 and in3. Then, the flow of inputs and outputs from Figure 3.2 were sent to and received from the state machine in Figure 3.1. The

Figure 3.2: Stream of tainted input and output



taint values of the outputs 'in1' and 'in2' were red and yellow respectively. This means they belong to the input parameters in1 and in2. Although the tainted values are assigned to other variables within the program, the original taint values remain the same.

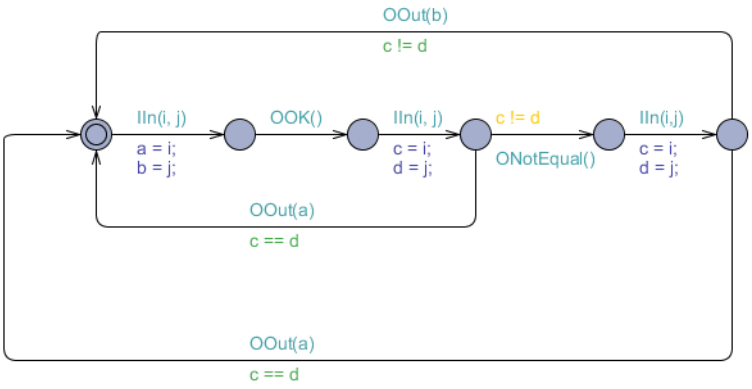
3.5 Tainting comparisons

Besides linking input and output values to each other, taint analysis can also find out which comparisons are done within the SUL. This makes it

possible to find out 'how' the output is received trough a series of inputs. Figure 3.3 shows a model that contains a few transitions with comparisons, which are only taken when the corresponding comparison yields true.

We initialized the six tstr objects in1, in2, in3, in4, in5, in6 with strings

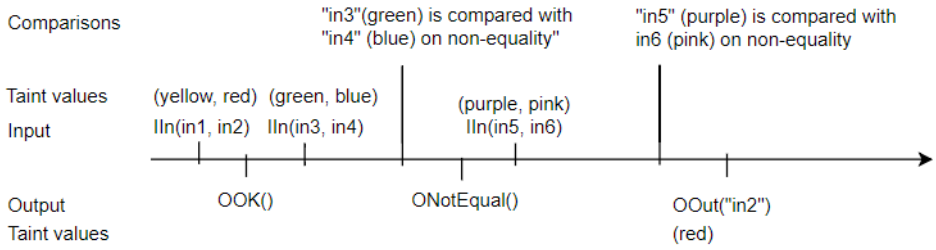
Figure 3.3: Model with comparisons



"in1", "in2", "in3", "in4", "in5", "in6" and taint values yellow, red, green, blue, purple, pink respectively. Figure 3.4 shows a flow of input, output and comparisons.

We start with the input IIn(in1, in2). This does not lead to any comparisons

Figure 3.4: Stream for model with comparisons

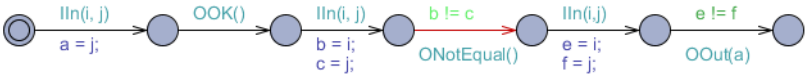


or interesting output, but due to the absence of comparisons we know that OOK() is the only output transition after the IIn input. After that, we feed the input IIn(in3, in4) to the model. Due to taint analysis we find that the comparison between "in3"(tainted green) and "in4" (tainted blue) is done after IIn(in3, in4). When we feed IIn(in5, in6), our final input, to the model, we receive the the comparison between "in5" (tainted purple) and "in6" (tainted pink) with output OOut("in2") where "in2" has a red taint value. From this information the following rule can be derived: "If

the parameters of the second IIn are not equal and the parameters from the third IIn are not equal, then the output is the second parameter of the first IIn(). With this rule and the rest of the information, we can construct the model in Figure 3.5. With only one run of our analysis, containing six input parameters a pretty large part of the model can be constructed. The next section describes how entire models can be constructed when multiple models like Figure 3.5 are combined systematically.

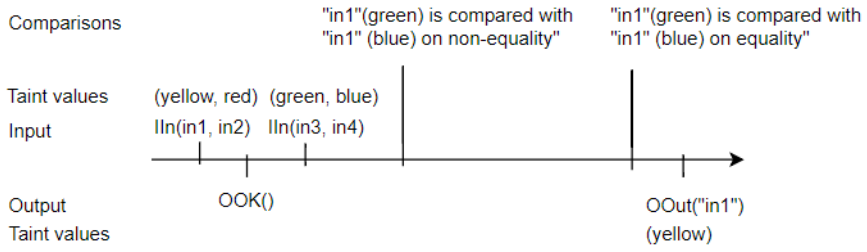
In our last example the taint value of the input is rather redundant, because

Figure 3.5: Learned model



the string value also describes which input is responsible for which output or comparison. If we look again at the model from Figure 3.3, but we change the string values of in1, in2, in3 and in4 all to "in1" and leave in5 and in6 out, the importance of the taint values becomes visible. Figure 3.6 shows a new flow of input, output and comparisons. Because every input parameter has the same value, it is impossible to find out which input corresponds to which comparison value or output. Due to the taint values we can still find out which inputs are actually compared.

Figure 3.6: Stream for importance taint values



Chapter 4

Learning Register Automata

To learn register automata we use RALib [3](described in section 4.1) in combination with taint analyses. Originally RALib uses a tree oracle, to construct decision trees for register automata. With these decision trees RALib can construct models of EFSM's (Extended Finite State Machines) using the SL* algorithm [4]. One downside is that the tree oracle of RALib interacts with the SUL as if it is a black box machine. Black box learning requires an enormous number of test queries, especially when models get larger and more complex. A solution is to replace the tree oracle from RALib with a new tree oracle, that can extract information from the SUL using taint analysis.

4.1 RALib

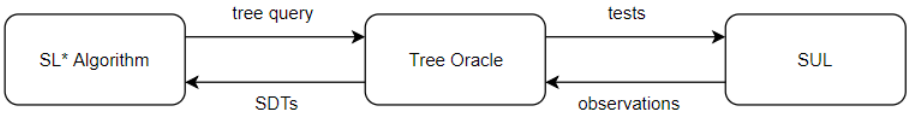
RALib [3] is a tool for active learning of EFSMs (Extended Finite State Machines) including register automata. Active learning means that the learning algorithm is able to interactively query the system under learning. To learn EFSMs, RALib uses the SL* algorithm. SL* has an observation table that stores SDTs for prefix-suffix combinations. With this observation table SL* learns models in a three phases: Hypotheses construction, hypothesis validation and counterexample processing. In the hypothesis construction the interaction with the tree oracle takes place. SL* sends prefixes and suffixes to the tree oracle and stores the resulting SDT's in the observation table. When SL* has stored enough SDT's in the observation table it constructs an hypothesis model. During hypothesis validation, SL* sends the hypothesis to an equivalence oracle that validates the correctness of the hypothesis or provides a counterexample. When the equivalence oracle provides a counterexample the counterexample analysis starts. During counterexample analysis SL* determines from the counterexample which part of the model is missing in the hypothesis. Then new prefixes and suffixes are added to the observation table to generate new SDT's for the missing parts of the

hypothesis. These phases repeat until no counterexample can be found.

4.2 Tree Oracle

A tree oracle is the bridge between the SL* algorithm from RALib and the SUL. The SL* algorithm from RALib sends tree queries to a tree oracle and receives SDTs (symbolic decision trees) from the tree oracle. SDT's and tree queries are a variant of the original membership queries that can be interpreted by the SL* algorithm. An SDT is a tree that has nodes that are either accepting or rejecting and transitions as edges. Every edge contains a parameterized symbol and a guard. A guard can be true, false, a comparison between two data values, a negation of a guard, a conjunction of two guards or a disjunction of two guards. The tree queries consist of a concrete

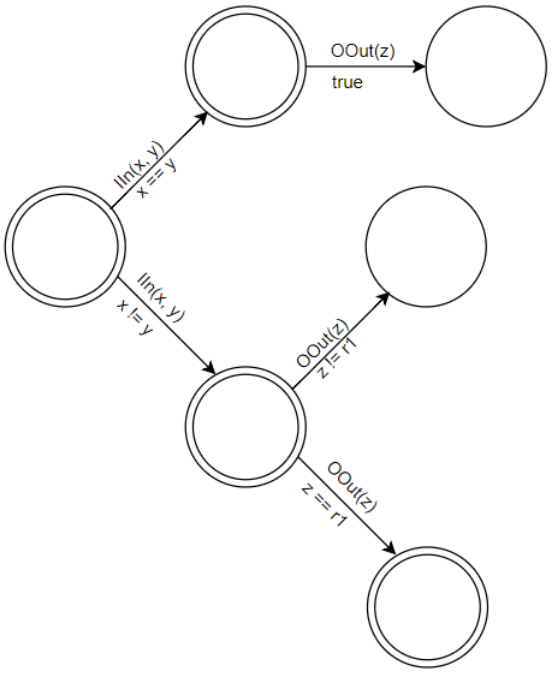
Figure 4.1: Inputs and outputs of a tree oracle



prefix and a symbolic suffix, which are both data words. The prefix contains the actions that are already taken from the initial state in the SUL, with concrete parameters. The suffix contains the actions that still need to be taken in the SUL, with parameters that do not have a value yet. For Figure 3.3 an example prefix and suffix would be IIn(2, 5) OOK() and IIn(x, y) OOut(z) respectively. The tree oracle sends test queries to the SUL, based on the prefix and suffix, and constructs an SDT from the responses. For the example above, the tree oracle could construct the SDT shown in Figure 4.2.

First the actions in the prefix will be fed to the SUL and from the resulting state the oracle will construct an SDT with the suffix. We say that the first input parameter of the prefix is stored in register r_1 and the second in r_2. As we can see only the branch with OOut(z), when z is equal to r_1 and x is not equal to y, results in an accepting state. In the case that x is equal to y the output is ONotEqual(), so it does not match the suffix output OOut(z) and in the case that z is not equal to r_1, the output parameter of the suffix does not correspond to the output parameter of the SUL.

Figure 4.2: Example of SDT



4.3 Tree Oracle with tainting

4.3.1 Algorithm

With the SL* algorithm and a SUL in hands, only the tree oracle has to be adapted to implement taint analysis in RALib. Algorithm 1 shows the pseudo-code of the tree oracle with tainting. For the algorithm to work correctly, the register automaton that is simulated by the SUL needs to satisfy the following restrictions:

- The automaton does not contain any constants, so every register is uninitialized in the initial state.
- All input and output actions have a maximum arity of one.
- Data values can only be compared on equality.
- Input actions always lead to an accepting state.
- Output locations have at least one outgoing transition.

The algorithm has one base case and a recursive part. It builds an SDT from the leaves back to the root. Every step in the recursion constructs an

SDT with the SDT's that are received from next step in the recursion. The base case creates an SDT with one node that has no branches. Furthermore, the base case contains two important steps, namely the taint run and normalizing constraints.

```
Data: A concrete prefix u and a symbolic suffix v
Result: A symbolic decision tree sdt and a set of constraints c
if v = λ (empty) then
    c = taint_run(u)
    normalize(c)
    return new sdt, c
else
    d = fresh value not in u
    sdt, c = SDT(u.i(d), v') where v = i(p).v'
    φ = constraint for i(p) in c
    tree = new sdt with branch (i(p), φ, sdt)
    for constraint in φ do
        d = second value in constraint
        sdt, c = SDT(u.i(d), v') where v = i(p).v'
        add branch (i(p), constraint for i(d) in c, sdt) to tree
    end
    return tree, c
end
```

Algorithm 1: building an SDT

4.3.2 Tainting

The taint run is where the actual taint analysis takes place. When the base case is triggered, the suffix is empty and the prefix is filled with actions that have parameters with a concrete value. In the taint run all parameters will be tainted with an increasing value. For example, with a prefix IIn(2)OOut(7)IIn(4)OOut(1), the 2 will be tainted with value 1 and the 7 will be tainted with value 2 and so on. Then, the inputs will be fed to the SUL one by one. For every input the corresponding output symbol of the SUL will be compared with the next output symbol of the prefix. Because every input value is tainted, all comparisons within the SUL will be recorded. When all inputs in the prefix are fed to the SUL, the resulting comparisons will be added to a list of constraints and returned.

4.3.3 Normalization

After obtaining the list of constraints from the taint run, the constraints need to be normalized for the recursion step. The parameters in the constraints are numbered with an increasing value, just like the taint values. A

possible constraint list of parameters x_1 til x_6 is $[x_1 == x_3, x_1 == x_2, x_3 == x_2, x_5 != x_6, x_4 == x_4]$. The normalization consists of the following steps:

1. for all constraints: the left and right side of the equation will be swapped if the left side is smaller than the right side.

2. The constraints will be sorted on the left side of the constraint in descending order.

3. for all constraints: if a constraint is of the form $x_i == x_i$, then remove this constraint from the list.

4. for all pairs (c_1, c_2) in constraints: if c_1 is equal to c_2, remove c_2 and if the pair is of the form $(x_i == x_j, x_j == x_k)$ or $(x_i == x_j, x_i == x_k)$, change this to $(x_i == x_k, x_j == x_k)$ and if the pair is of the form $(x_i == x_j, x_j != x_k)$ or $(x_i == x_j, x_i != x_k)$, change this to $(x_i != x_k, x_j != x_k)$.

If we take the constraint list example above, then after step 1 the constraint list is $[x_3 == x_1, x_2 == x_1, x_3 == x_2, x_6 != x_5, x_4 == x_4]$, after step 2 it is $[x_6 != x_5, x_4 == x_4, x_3 == x_2, x_3 == x_1, x_2 == x_1]$ and after step 3 and 4 it is $[x_6 != x_5, x_3 == x_2, x_2 == x_1]$. After normalizing the constraints, the constraints are in order from large to small, do not contain duplicates and for each j, constraints for x_j are either of the form $x_j == x_k$ or $x_j != x_{k1}, x_j != x_{k2}, ..$.

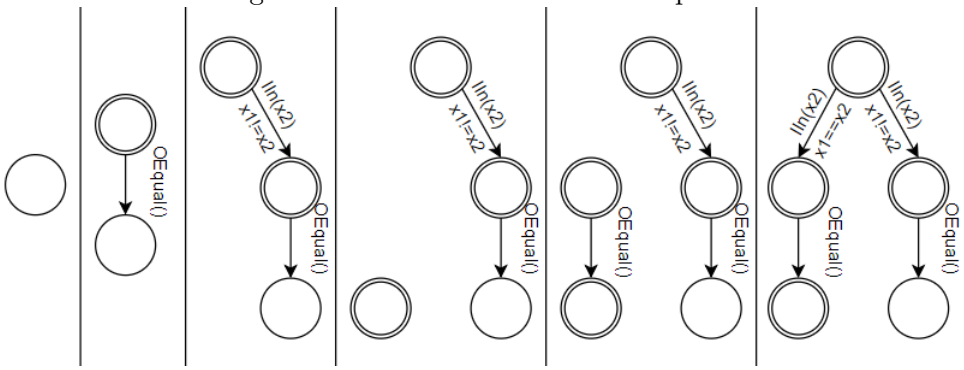
4.3.4 Recursion

In the recursive case a fresh value d (a value that is not assigned to any of the parameters in the prefix) is assigned to the next parameter of the suffix and added to the prefix. Then the recursion yields an SDT and a list of normalized constraints. Because d is freshly chosen, the constraint for $i(d)$ is either true or a conjunction of inequalities. If the constraint is true, a new SDT with an edge, containing a true guard, to the SDT from the recursion is returned. Otherwise the branch has the conjunction of inequalities as guard and for every inequality in the conjunction a new branch will be created to a new tree, where one of the inequalities is an equality. For example, if the constraint is $x_i != x_j$ and $x_i != x_k$, a transition with $x_i != x_j$ and $x_i != x_k$ as guard is added to the yielded SDT. Then two new SDT's are created, where the value of d is equal to the value of x_j in the first SDT and the value of d is equal to the value of x_k in the second SDT. The first SDT then has a transition to it with $x_i == x_j$ as guard and the second SDT has a transition to it with $x_i == x_k$ as guard.

4.3.5 Example

We illustrate how this algorithm works with an example. Figure 4.3 shows how the SDT is step by step. When an edge has no guard in the Figure, the guard is true. Let's assume a prefix IIn(1).OOK() and suffix IIN(x_2).OEqual(). In the first step of the recursion the suffix is not empty, so we end up in the else branch. A fresh value, let's say 12, is chosen for d. Then the algorithm goes into the next step of the recursion with prefix IIn(1).OOK().IIn(12) and suffix OEqual(). This time no fresh value is assigned to OEqual(), because OEqual() has no parameters. In the next step of the recursion we have IIn(1).OOK().IIn(12).OEqual() as prefix and an empty suffix, so we end up in the base case. The parameters 1 and 12 will be tainted with the values 1 and 2 respectively. IIn(1) and IIn(12) are fed to the SUL with OOK() and ONotEqual() as output. From the taints, the comparison list $[x1! = x2]$ is returned. In the normalization the constraint list is changed to $[x2! = x1]$.

Figure 4.3: SDT construction example



When returning from the recursion sdt is a single node and c is $[x_2! = x_1]$. Because OEqual() has no parameters, ϕ will be empty, so the for loop will be skipped. The new tree is then returned to the first step of the recursion. Now ϕ will be $[x_2! = x_1]$ for action IIn(x_2). A new edge with $x_2! = x_1$ as guard is set to the received sdt. Then the for loop has one iteration in which x_2 gets the same value as x_1, namely 1. The recursion is exactly the same as the recursion outside the for loop except that the taint run outputs OOK() and OEqual(), with $[x_1 == x_2]$ as comparison. In the figure the very first node is non-accepting, because ONotEqual() is the wrong output for this prefix. Non-accepting nodes are not added in the pseudo code, because they do not contribute to better understanding of the algorithm.

4.3.6 Complexity

There are two different ways to measure the complexity of this algorithm. First, there is the usual complexity measure, namely the running time of the algorithm. We can also measure the complexity in terms of input actions that are fed to the SUL. The second way is most important for our research, because the information that is extracted from the SUL with taint analysis should lead to a lower number of input queries compared to a black box tree oracle. The time complexity is mainly dependent on the number of leafs in the decision tree. For every leaf the algorithm has to do a run on the SUL and normalize the constraints. A run on the algorithm with a prefix of length p and a suffix of length s constructs worst case an SDT with $O(\frac{(p+s)!}{p!})$ leafs. The root of the tree has maximally $p+1$ transitions. The new value can be equal to any of the previously stored values or unequal to all of them. For every case there can be a transition with the corresponding equalty as guard. The children of the root have maximally $p+2$ transitions, because one more value is added to the registers. Every step further to the leafs can have one more transition. The last step before the leafs has a maximum of $p+s$ possible transitions. We can conclude from this that length of the suffix has a large influence on the complexity of the tree oracle, while the influence of the prefix is rather small.

Chapter 5

Experiments

For our research we test the new version of RALib against the original implementation of RALib. As the original version of RALib uses a black box oracle for computing SDT's and the new version uses an oracle that can extract information from the SUL, we expect that the new version uses less input queries to learn models. To test the different implementations of RALib we use a couple of benchmarks.

5.1 Setup

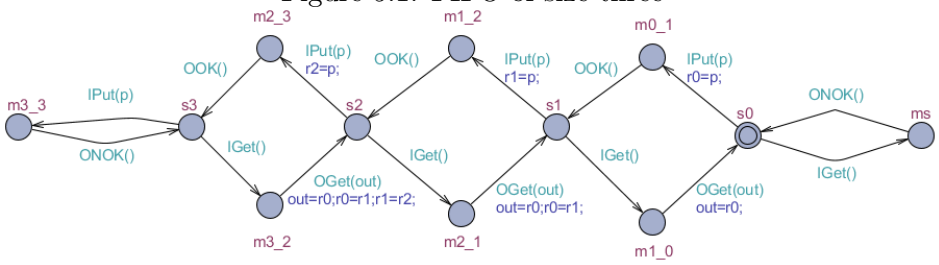
All experiments have the same setup. We ran each experiment 5 times, with a maximum time of 10 minutes. Both versions of RALib use SL* as learning algorithm. For the tainted version we use a random walk, while the black box version compares the hypothesis with the correct model to find counterexamples. A random walk starts with an empty data word. Then random concrete inputs and outputs are added to the data word step by step. Each random walk has the property to stop after every step with a probability of 10% and generates a maximum of 1000 data words. Each data word is fed to the SUL. If the data word is accepted, the random walk generates a new one and if the data word is rejected, it is a counterexample. The maximum number of steps is 15, because longer counterexamples could generate suffixes that increase the processing time of the counterexample too much. It could be that a counterexample of 15 steps still creates too long suffixes, but with a lower steps, the chance of finding counterexamples decreases too much. When a data value is needed for the the counterexample a fresh value is is created with a probability of 80%. This setup is partly adopted from [3].
To compare the results of the experiments we count the number of inputs and resets that are needed to learn a model. The number of inputs are the input queries that are sent to the SUL. The number of resets corresponds to how many times the SUL is restarted from the initial state. When count-

ing queries, we differentiate between learn queries and test queries. Learn queries are counted in the tree queries and test queries counted in the possible counterexamples generated by the random walk. We mostly focus on the learn queries, because they indicate the efficiency of the tree oracles. Also, all test queries that are used to find a counterexample against the 'correct' model are not counted, because that counterexample does not exist. The correctness of the generated models was checked manually.

5.2 Experiments

For our experiments we used a FIFO(First in First out) model. The FIFO model acts the same as a queue, which means that you can put elements in the FIFO one by one and get the first stored element from the FIFO, if the FIFO is not empty. The FIFO models that are used for the experiments have a fixed size. Figure 5.1 shows an example of a FIFO model with a size of three elements. FIFO models with a different size look similar to this one, but with another number of states. We ran both versions of RALib on a FIFO with a size of two till seven. The results are stored in Table 5.1.

Figure 5.1: FIFO of size three



The tests from RALib black box, with a FIFO size of 6 or lower all terminated within 10 minutes. All the tests with a size higher than 6 did not terminate within 10 minutes. For the tests from RALib with tainting, FIFO5 had 2 and FIFO6 had 1 run that did not terminate within 10 minutes. The rest of the tests all terminated within 10 minutes. To get a better overview of the results we ran FIFO7 from RALib black box again within 20 minutes. 4 out of 5 runs terminated within 20 minutes. Those results are stored in the table. All the values in the table are the average of the 5 or less tries.

5.3 Results

As we can see the number of learning inputs and resets used by the tainted version of RALib is larger than those from black box RALib. For testing the

	RaLib black box				RALib with tainting			
	Learning		Testing		Learning		Testing	
model	inputs	resets	inputs	resets	inputs	resets	inputs	resets
FIFO2	260.8	95.6	128.4	24.8	512.8	183.2	115	23.2
FIFO3	744	193.2	273.6	43.2	1243.2	342	270.6	41.6
FIFO4	1209	255.6	458.8	67.2	2122.2	484	352.5	52.3
FIFO5	2178.6	386.8	1074.8	143.6	4403	832.6	713.3	95.6
FIFO6	3197	503.8	2333.4	300.4	6722.8	1112.2	1779.8	228.8
FIFO7	6976	912	5508	707.5	10355.6	1498.2	4055.6	524

Table 5.1: RALib comparison on FIFO models

results look slightly better for the tainted version of RALib. While running the experiments the bottleneck for RALib with tainting was the processing of long counterexamples and large SDT's. Because of the random walk a long counterexample could be found at any point of the learning process. Long counterexamples lead to long suffixes, which increases the number of tree queries drastically.

5.4 Discussion

The number of learning queries of the tainted version of RALib is larger than those of the black box version of RALib, while in theory it should be the other way around. The main reason for this is that the two versions of RALib use a different method to find counterexamples against the hypothesis. The black box version has the complete model and compares the hypothesis with the complete model. From this it can generate short counterexamples which lead to short suffixes and therefore less tree queries. The tainted version of RALib does not learn from the model itself, but from a SUL that simulates the model. Therefore the model cannot be used to find short counterexamples. Although results indicate that the tainted version of RALib requires more input queries for learning, the tainted tree oracle could still use less tree queries than the black box oracle. An option would be to measure the number of inputs used per tree query, but that would also give unreliable results. The length of the prefix and suffix would then influence the number of input queries per tree query. We could dig into RALib to extract the black box tree oracle and compare the amount of input queries for specified prefixes and suffixes, but that requires a deeper understanding of RALib, which is beyond the scope of this thesis.

Chapter 6

Future Work

We've shown that it is possible to integrate taint analysis in model learning, but there are many improvements that can be made in this area.
First of all, the current implementation of the tainted tree oracle supports learning for only a small set of models. The tainted tree oracle does not support any models that contain constants or actions with more than one parameter. Optimizing the tree oracle will extend the range of models that can be learned. Models that contain these features are generally more complex and require much more learning queries with black box learning. The extraction of information through tainting could then optimize the learning process even more.
Second, the results of this thesis did not satisfy the expectations, because black box RALib uses an equivalence checker, that contains the model, to generate short counterexample, while the tainted version is dependent on a random walk, which generally generate larger counterexamples. If we could use the equivalence checker from black box RALib in the tainted version, the results can be measured correctly.
Third, the tool that is used for tainting is now implemented in python. Therefore the tainted tree oracle and the SUL are also written in pyhton. Because RALib is implemented in java, the data that is sent to and received from the tree oracle needs to be communicated between the two languages. This is now done through json files, which causes two difficulties. First, reading and writing from files is very time consuming compared to direct communication, especially when this has to be done many times. Second, the data structures need to be translated, which is more difficult than using the data structures of RALib. A solution to these difficulties is to translate the current tainting tool from python to java. This way the entire tainted tree oracle can be implemented in java. As a result no more communication through files is needed and the data structures from RALib can be directly used to construct SDT's.

Chapter 7

Conclusions

We analyzed the integration of taint analysis into model learning and compared the efficiency of learning EFSMs against RALib. First, we managed to extract information from a SUL using a taint analysis library from the Pygmalion framework. Then, we implemented a tree oracle that uses tainting to construct SDTs from a concrete prefix and a symbolic suffix. After that, we integrated this tree oracle in RALib. Then we compared the original RALib with the tainted version on a few FIFO models of different sizes. From the results we could not conclude whether the tainted version of RALib preforms better, because both versions of RALib use a different way of generating counterexamples for an hypothesis of the model, which lead to unreliable results. What we can conclude is that taint analysis can be integrated in model learning to learn models with information that is extracted from the SUL. We've discussed some improvements for the tainted tree oracle and the integration in RALib in the future work section that could lead to more promising results.

Bibliography

[1] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computing*, page 87–106, 1987.

[2] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Uppaal - a tool suite for automatic verification of real-time systems. *Hybrid Systems III*, pages 232–243, 1996.

[3] Sofia Cassel, Falk Howar, and Bengt Jonsson. Ralib: A learnlib extension for inferring efsms. 2015.

[4] Sofia Cassel, Falk Howar, Bengt Jonsson, and Bernhard Steffen. Active learning for extended finite state machines. *Formal Aspects of Computing*, page 233–263, 2016.

[5] Sofia Cassel, Falk Howar, Bengt Jonsson, and Bernhard Steffen. Extending automata learning to extended finite state machines. *Machine Learning for Dynamic Software Analysis: Potentials and Limits*, page 149–177, 2018.

[6] Rahul Gopinath, Björn Mathis, Matthias Höschelle, Alexander Kampman, and Andreas Zeller. Sample-free learning of input grammars for comprehensive software fuzzing. *Communications of the ACM on Programming Languages, Vol. 1, No. OOPSLA, Article 1*, 2018.

[7] William G.J. Halfond, Alessandro Orso, and Panagiotis Manolios. Using positive tainting and syntax-aware evaluation to counter sql injection attacks. 2006.

[8] Falk Howar, Bengt Jonsson, and Frits Vaandrager. Combining black-box and white-box techniques for learning register automata. *Communications of the ACM*, pages 86–95, 2017.

[9] Matthias Höschele and Andreas Zeller. Mining input grammars from dynamic taints. *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 720–725, 2016.

[10] CAI Jun, ZOU Peng, MA Jinxin, and HE Jun. Sworddta: A dynamic taint analysis tool for software vulnerability detection. *Wuhan University Journal of Natural Sciences*, pages 010–020, 2016.

[11] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 89–100, 2007.

[12] James Newsome. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. 2005.