Bachelor thesis
Computing Science

Radboud University

Exploring taint analysis methods for the grey-box learning of Java systems

Author:
M. F. L. (Maris) Galesloot
s4634098
`m.galesloot@student.ru.nl`

First supervisor/assessor:
Prof. Dr. F. W. (Frits) Vaandrager
`f.vaandrager@cs.ru.nl`

Second supervisor/assessor:
B. (Bharat) Garhewal MSc
`b.garhewal@cs.ru.nl`

August 21, 2020

Abstract

Model learning is a branch of research involved in inferring interface semantics of software components, where semantics are often captured in register automata. Grey-box dynamic taint analysis has proven useful to reduce the number of actions needed to both construct a representative automaton as well as verify its correctness. We show that the technique used in Python can also be exploited in other languages, that is in Java, albeit with some extensions and modifications to the language. The implementation in its current state proves limited to equality relations. However, we show that our taint analysis implementation of the tree oracle results in a large improvement in the number of inputs needed to learn the interfaces of the `java.util` library classes compared to the black-box methods of RALib. Furthermore, limitations of the current implementation in Java and alternative tainting techniques for Java are discussed.

Contents

Chapter 1

Introduction

1.1 Models

Software analysis and verification provides the basis on which we trust the applications we use. In our daily lives we constantly use all different sorts of machines, from the simple kitchen device that prepares your morning coffee to the complex systems your smartphone consists of and the inter-communicating applications your respective device runs. All these systems have some sort of control mechanism that define their behaviour for certain inputs. Your coffee machine reacts to your input and you decide on these operations because, as a human, systematic derivations of possible inputs and output of such a simple system are easily constructed.

These conceptual models that we create are not as easily constructed by computer systems. In the research area of model learning, methodologies are being designed to improve a computer's capability of constructing such a state diagram. These models are constructed to represent a model of the program's behaviour. The program or system learned is also known as the System Under Learning (SUL). These models can be inferred without access to the SUL's source code, also known as a black-box setting. In this setting, the program can only be referenced by observing and recording output values to certain input values. In black-box environment settings, these algorithms often depend on a trial-and-error approach to figure out what outputs correspond to what inputs. Scalability becomes an obstacle, since the number of membership queries grows cubic in the size of the model, and the number of test queries grows exponential in the number of states of the SUL [1]. Meaning that the more complex a learned system gets, the more state transitions have to be learned by providing different inputs. This is due to the fact that in a black-box setting, providing plain inputs results in plain outputs. A goal in model learning research is to extract as much information as possible out of the system, such that the algorithms can decide on entering a valid and interesting value while also reducing the number of possible input values. There are a number of valid ways to incorporate extra information in the model learning process. This extra information, such as relations between input values, can often be formalised as constraints and presented to a solver. The reason for doing this, is that solving for the constraints on input values produces a concrete assignation to these values that, when presented to the SUL, could produce system behaviour to gain knowledge about.

1.2 Research context

In this thesis we look at taint analysis and the process of tainting input objects within the approach of dynamic analysis, which can reduce the number of (inputs in) queries necessary to construct a model, specifically for the Java language. Essentially, we try to extract run-time information on the behaviour of programs. With this information we are able to, with the help of learner algorithms such as the RaLib [2] framework based on SL*, and with making certain assumptions about how variables are processed during execution and to what extend this influences what locations of the program are reached, hypothetically use less interactions to learn models. Because we need less queries to construct a model, we can also move to hypothesis construction earlier. Which in turn leads to a quicker result: either a finished model or a counter example that will help adapting faulty states and or transitions of the model. As real-life applications of model learning include the refactoring of legacy software components, the relevance of the extension of taint analysis methods for learning industry languages such as Java is evident. Beyond this, we will look into extending the taint process to incorporate richer data operations, e.g. with arithmetic operations such as addition, multiplication, division and subtraction. Current automata learners do not support the inclusion of these "rich" operations, but SMT solvers, such as Z3, provide sufficient computational solving power to use arithmetic relations as an asset for future model learning endeavours.

1.3 Outline

In the second chapter (2), we provide the preliminaries in understanding the techniques that underlie this thesis. In the third chapter (3) tainting methods are discussed and in the fourth chapter (4) a tainting implementation for Java is demonstrated. In the fifth chapter (5), we show the experiments that were done to evaluate the results of the research. In the sixth chapter (6), we show our understanding of the research area by elaborating in related works and similar implementations. Also, future work is presented based on limitations in this thesis and possible further research. In the final seventh chapter (7), we conclude and discuss our research.

Chapter 2

Preliminaries

In this chapter, we present the prerequisites to understanding the concepts of this thesis.

2.1 Model learning

2.1.1 Black, grey, and white-boxes

Model learning can be utilised to infer program behaviour from computer based systems, that is, systems whose behaviour is prescribed by computer programs. Since learning of program models is the most prevalent use case. Model learning techniques can be applied to computer systems that have predictable behaviour. When you input certain values, the resulting output then is based on a set of internal instructions (i.e. code) of the program. Most of the time this is invisible for the user, imagine your coffee machine again, but there is an underlying model to be inferred that specifies the system's behaviour. As mentioned before, there are two types of scenarios

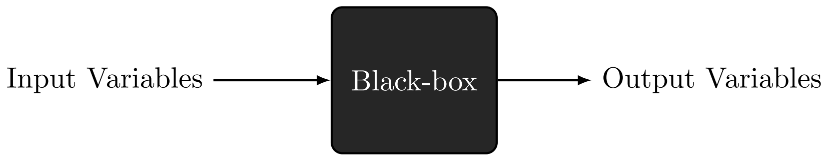

Figure 2.1: Black-box setting, only perceived entities are input and output variables.

in software analysis, namely white- and black-box settings. The black-box techniques for model learning assume that everything inside the target machine is hidden. White-box techniques depend on certain information to be gathered out of a program's inner workings. In this thesis we focus on *active* model learning using *dynamic analysis.* This means we actively construct partial models and queries that we base on the information we have gathered during dynamic analysis in a white-box setting. A white-box setting often means you have access to the code/program you are analysing. Sometimes, there is a mix between the two, were only part of the program is visible (i.e. white-box) and another part is hidden (i.e. black-box). This is also known as a grey-box environment. Dynamic analyses methods often introduce grey-box methods of learning, where part of the program state becomes available to the learning algorithm. Theoretically this would include constraints on input variables and other program state constraints. Only this particular context of information about the program execution is then known to the learner on top of the input and output values, not necessarily the entirety of information one could possibly deduce from reasoning about the source code. Contrary to classic black-box I/O approaches, the grey-box settings implies there is extra information being gathered apart from input and output values. As an example of code you do not have access to, think of a program that uses standard library functions you do not have the sources of, or uses a third-party dependency with a public API but a private source code repository. Moreover, consider the examples of embedded devices, such as coffee machines and electronic passports. If not specified, assume in this thesis the context implies a grey-box setting.

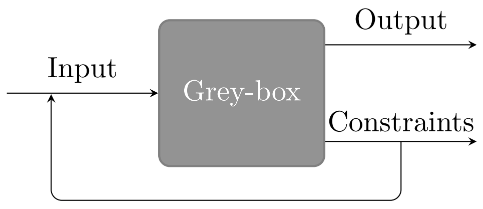

Figure 2.2: Grey-box setting, input variables produce output variables and a set of constraints. The constraints can help formalise new input variables.

2.1.2 Models, learners and teachers

If we practice model learning, we assume a *learner* and a *teacher* from the **M**inimally **A**dequate **T**eaching framework (MAT) [3]. The framework, as seen in Figure 2.3 together with the well known algorithm L^* (for learning DFA's) from 1987, is a fundamental model learning paradigm. Whilst there are also other learning paradigms, a large part of contemporary model learning research work still builds on the theories and groundwork that were produced. In essence, it is the teachers responsibility to interact with the model (SUL). The learning is done in the learner, which sends membership and equivalence queries to the teacher. *Membership queries* contain input values that the teacher should present to the model, the teacher then receives some returned output value, which it translates to the language the learner understands. When a certain limit of membership queries is reached,

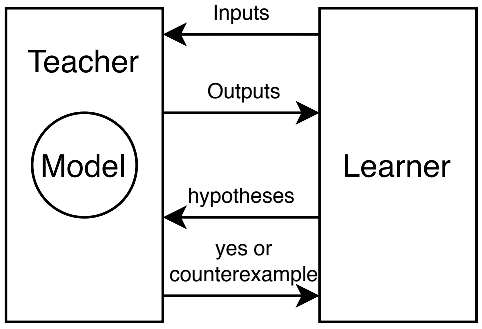

Figure 2.3: Minimally Adequate Teacher framework for automata learning.

which can be numerical or some set of constraints, the learner presents its learned model to the teacher. The teacher then formulates a set of *equivalence queries* that determines whether the learned model and the actual program behaviour is deemed equivalent. If yes, the hypothesis of the model is correct and learning is done. If no, the teacher calculates a counter example that reveals an error in a state and/or transition and thus disregards the hypothesis model. Learning must then continue.

Concluding, most of the optimisation for learning frameworks is realised by:

- reducing the number of inputs in membership queries, and
- extracting as much information as possible from the execution behaviour with each membership query.

The second point in itself improves the criterion of the first point, since we can provide more intelligent queries to the teacher. These bullet-points also apply to the eventual equivalence queries.

2.2 Automata Theory

2.2.1 From DFA's to EFSM's

The MAT framework and the L^* algorithm for learning Deterministic Finite State Automata, which have been successfully used to learn behavioural models of software components [4], also have their limitations. DFA's, with their finite input alphabet, struggle to deal with parameterized actions, whose input values can range over large or infinite domains [5]. Also, these classical automata lack the tools to model control flow in programs. For many applications, models should express control flow, which are the states of the program that is being learned, data flow, which translates to the constraints on data parameters that are distributed between a component's internal and external variables and when the component interacts within the context it has access to. On top of that, the interactions between the control and data flow of a to be modelled component are vital characteristics that such a model should describe [6]. These factors become more prevalent in modern software, which employs more complex data structures and ranges over more various input variables. Consider the follow simple example of control flow in three lines of standard Java code. Assume that there is some arbitrary maximum size of the set, *one* in this case:

```
java.util.HashSet<T> hashset = new HashSet<>();
hashset.add(x); // true
hashset.add(x); // false
```

Figure 2.4: Java `HashSet` with two `add` operations.

The symbolic variable x is added to the `java.util.HashSet`, and then variable x is added again. The first time, the value of the variable is hashed and stored in the set. In this case the add operation returns true. We would record, and possibly inform the teacher depending on the query, that the action was successful. The second time, the set refuses to store the variable since it already exists and returns false. In this case, the query to add value x again is not unaccepted behaviour. It simply means that when we add the same variable again, the automaton should register for this transition that the output for input variable x_1 and local register value x_0 is accepted when $x_0 = x_1$. Simple program behaviour like this is not captured easily in classical automata, who have no guards on their transitions. In order to capture simple but essentially complex to learn behaviour like this, we use extended finite state machines that can store, compare and manipulate *data values*. In the case of the active learning framework RALib [2], which is a library based on the SL^* algorithm for EFSM's, these are *Register Automata* (see 2.2.4).

2.2.2 Data Languages

When we learn certain programs or protocols, we define a *theory* within we learn the automata that defines the model. In our model learning setting, we parameterize the learning algorithm with a certain theory, which is a pair $\langle \mathcal{D}, \mathcal{R} \rangle$ where $\mathcal{D}$ is a domain of data values and $\mathcal{R}$ is a set of relations on $\mathcal{D}$. The arity of the relations can be of arbitrary length. Arithmetic operations $(++, -, +)$ can be defined by relations on data values [1], constants can be expressed with relations of arity one. An example of a simple theory is the following:

- $\langle \mathbb{Z}, \{=\} \rangle$, theory of the set of integers with respect to the equality relation. We can also say: "the equality theory of integers".

We assume a set of *actions* Σ that takes a set of parameters $\{d_1, \dots, d_n\}$ from the domain $\mathcal{D}$, depending on its arity n. A data word w then is a collection of data symbols. A *data symbol* is a term $\alpha(d)$ with $\alpha \in \Sigma$ and $d \in \mathcal{D}$. The data symbol represents an action that is performed on a certain data value. A data language $\mathcal{L}$ is a set of data words. Data languages are often represented as a mapping from the set of words to the tuple $\{+, -\}$. Representing an accepting or rejecting computation, e.g. see the λ map of definition 1 of the Register Automaton. The acceptance of data words can be determined with an EFSM such as a Register Automaton.

2.2.3 Guards

Assume we have some countably infinite set of variables $V = \{v_1, v_2, \dots\}$, that represent program variables. Also assume some variable $p \notin V$, which is the formal parameter of input symbols. A guard is a conjunction of (negated) relation symbols over variables V. We define guards inductively as defined in [6]:

- If $r \in R$ is a n-ary relation on input variables $\{x_0, \dots, x_{n-1}\}$ from $V \cup p$, then $r\{x_0, \dots, x_{n-1}\}$ **and** $\neg r\{x_0, \dots, x_{n-1}\}$ are guards.
- if g_0, g_1 are guards, then $g_2 = g_0 \wedge g_1$ is also a guard.

Guards commonly range over binary (dis)equality relations, such as one of {`==, !=, <`}. In this case, guards are conjunctions of these binary constraints.

2.2.4 Register Automata

Now that we know how to represent more complex input values as data symbols and have defined guards, we can look at the definition of a Register Automaton:

Definition 1 (**Register Automaton**). *A register automaton $\mathcal{A}$ is defined by the tuple $\mathcal{A} = \{L, l_0, \mathcal{X}, \Gamma, \lambda\}$, where:*

- *L is a finite set of locations, with $l_0 \in L$ the initial location,*
- *$\mathcal{X}$ maps each location to a finite set of registers $\mathcal{X}(l)$ with $\mathcal{X}(l_0) = \emptyset$,*
- *Γ is a finite set of transitions of the form $\langle l, \alpha(p), g, \pi, l' \rangle$, where*
  - *l and $l' \in L$ are source and target location respectively,*
  - *$\alpha(p)$ is a paramaterized symbol, with action α and data value p,*
  - *g is a guard on data value p and $\mathcal{X}(l)$,*
  - *π, an assignment mapping from $\mathcal{X}(l')$ to $\mathcal{X}(l) \cup \{p\}$, updates registers in $\mathcal{X}(l')$ with the parameters from p and $\mathcal{X}(l)$, and*

- λ *maps every location* $l \in L$ *to the set* $\{+, -\}$.

Instead of "normal" state machines, such as finite state machines **without** registers, register automaton have differing states and locations. In a register automaton a state is defined by the pair $\langle l, i \rangle$ where $l \in L$ is the location and i is the assigning function of the registers of that location. In simpler terms: i is a valuation of the set of registers for some location l. For example, if $\mathcal{X}(l_1) = \{x_1, x_2, x_3\}$ and i valuates this set of registers to $\{2, 4, 8\}$ then the **state** of the register automata would be described as follows: $\langle l_1, \{x_1, x_2, x_3\} \mapsto \{2, 4, 8\} \rangle$. It becomes clear that whether a certain transition is chosen can depend on the values of the variables in the register for some location. This means that the same action (e.g. `add(x)`) can induce different transitions depending on the state. As we can see can Register Automata, being part of the set of Extended Finite State Machines, describe much richer models, including control flow paradigms such as S_0 `:= if` *expr* `then` S_1 `else` S_2 and the storage of data values. For instance, a Register Automaton can capture the third line of code from code example 2.4 as a different transition, remembering whether symbolic variable x was already added before.

Inputs and Outputs

When we discuss action with data symbols, we generally imply I/O actions to and from the model. We often describe a textual action to the teacher, such as "input variable x to the model" as `IPut(x)`. Generally, data symbols that input something start with `I` and as expected start with `O` for certain outputs. When we "put" something for example, the general expected outcome is either "succes" or "rejected", these are represented by `OOK()` and `ONOK()` Then, the teacher translates this to the (code) model in a way it understands. In the case our code example of Figure 2.4, you would think the output `false` leads to a not accepting state. This is not the case however, since the internals of the `HashSet` function properly and adding the same value twice is completely normal behaviour, but does not result in the value actually being in the set two times. Ideally we would record that, when putting a value in the set twice, we would capture the transition guard on the input constraint derived from these two values (the internal `HashSet` value and the input value).

We can see in Figure 2.5 how we translate symbols to actual actions on the model. We would encapsulate the set model in Java class, we will leave the interaction between the wrapper class and the model to your imagination, since it does not really matter for this level of abstraction. The statement `add(x)` represents the call to the HashSet and `true` represent the return value of the actual component under testing, not necessarily the return value you would translate and return to the learner. Keep in mind we generally represent error or failures with `ONOK()`, so if for example the

```
Symbol:      IPut(x)       OOK()   ->  IPut(x)        OOK()
               |             ^           |              ^
               V             |           V              |
Program:     add(x)   ->   true        add(x)   ->   false
```

Figure 2.5: Schematic overview of the relation between symbols and code that is called on the model.

model for whatever reasons throws some Java exception this would be the way to represent this. Occurrence of `ONOK()` would also be the case when we would add any other variable y after adding x, since the maximum size is set to one. We would not actually try to insert y but merely return `ONOK()`. The following data words demonstrate the difference:

(i) `IPut(x) OOK() IPut(y) ONOK()`

(ii) `IPut(x) OOK() IPut(x) OOK()`

Both traces are accepted queries in the model of the Java `HashSet` of size one.

Example

See Figure 2.6 for a learned model of a `Java.util.HashSet` with a artificial maximum size of one.

2.3 Tree Queries

As we have now established, *active learning* concerns the process of an automaton Learner that sends queries to a Teacher which then can formulate these queries and communicate them with the target system, the System Under Learning (SUL). As we explained, in the classical L^* algorithm, the target language is a regular language over some finite alphabet. Contrary to the learning of regular languages, the answer to membership queries in a data language, for example using the algorithm SL^*, is determined by the relations between data values. Not necessarily the concrete values themselves [2]. The discovery of this relation is done via *tree queries*, where several membership queries can be grouped in to a single tree query.

In tree queries we presume input as a list of two certain entities:

- *Prefix*: A data word w which is a collection of data symbols $(\alpha(d))$. Recall that α is an action and d a parameter from some domain $\mathcal{D}$.
- *Suffix*: A sequence of not instantiated data parameters.

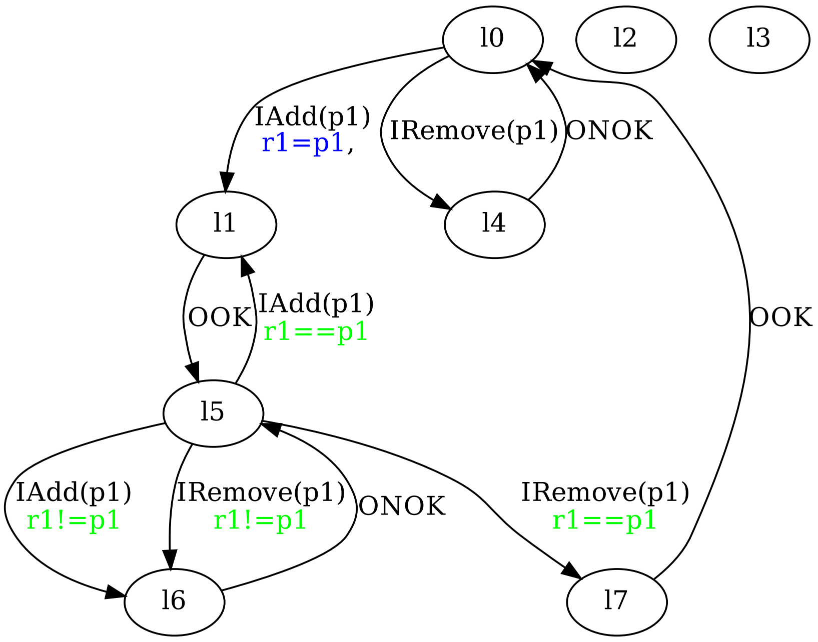

Figure 2.6: Learned model of HashSet with maximum size one.

2.3.1 Symbolic Decision Trees

The answer to such a query is a *Symbolic Decision Tree* (SDT). As with other decision trees, it is a tree data structure which can represent control statements as a branch in the tree. In this case, the values of the tree are symbolic. This means they are not actual concrete values but rather define program structure for some input language (i.e. data language). Using the tree, data values of the prefix are stored in registers. The edges of a SDT are similar to the transitions of a Register Automaton. In the branches, these data values are compared to the parameters of the suffix. Doing this, we can determine from the comparisons of the relations of the prefix and the suffix whether the subsequent paths of the prefix exists in the target data language.

Example

Consider the example from the code example of Figure 2.4 again. Assume we have a prefix `IAdd(x)` and a symbolic suffix `IAdd(p) IRemove (p)`, then the tree from Figure 2.7 show the accepting and non accepting computations of this query. After the prefix the state of the registers is $\{x_1\}$. The right sub-tree is never accepted, since the first action involves adding a value to the set of size one that is not already in the set. The other side of the tree involves adding an equal value, as seen by the guard $p = x_1$. Then the `IGet` action decides whether the trace is accepted (i.e. it reaches state $\{+\}$), the action is only valid iff $p = x_1$.

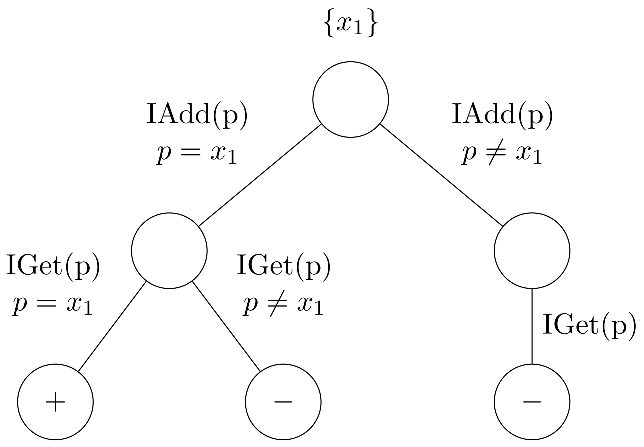

Figure 2.7: Symbolic Decision Tree for prefix `IAdd(`x`)` and symbolic suffix `IAdd(p) IGet(p)` for a java.util.HashSet of maximum size one.

2.4 SMT Solvers

As mentioned before, *SMT solvers* have the potential to play a large part in the learning process. This is because the learning method in grey-box setting, whether using tree queries or otherwise, aims to produce a list of constraints after every input query. We know that Boolean satisfiability (SAT) is defined as the problem of determining whether there exists a valuation for a Boolean formula such that the result of the formula yields `true`. If this is the case, the formula is *satisfiable*. Which means there is a certain set of concrete assignments to the variables in the formula such that all the constraints in the formula are met.

2.4.1 Generalising to SMT

In our context, this translates to the following: there exists some input value, data word $w = \alpha(p_1) \cdot \alpha(p_2) \cdot \ldots \cdot \alpha(p_n)$ that satisfies the conjunction of the inverse of every constraint $\neg\phi_1 \wedge \neg\phi_2 \wedge \cdots \wedge \neg\phi_m$ derived from the inputs on the model, where these constraints describe relations between input variables within a certain theory. In relation to the tree queries of section 2.3, we stop the querying process and formalise a SDT when there does not exists such an input. Because the target language of the systems under learning naturally are not always part of the language of Boolean expressions, we need some sort of generalisation of the logic that can translates to other (more elaborate) data languages, such as integers or Strings. Satisfiability modulo theories (SMT) is a generalisation of the Boolean satisfiability problem by adding support for first-order theories, such as arithmetic and equality reasoning [7]. SMT solvers enable us to reason about complex input languages and the constraints on the input domain returned from the model. Because these solvers can range over a set of complex data types, the bottleneck of learning often comes from the actual learning algorithm itself. Currently, the `Z3` solver is the prevalent SMT solver. The biggest challenge of grey-box active automata learning is to extend learning algorithms, either by extending the algorithm itself or extensive analysis surrounding the algorithm, to interpret constraints from complex relations that can be solved by a SMT solver such as Z3.

Chapter 3

Dynamic Taint Analysis

In this chapter, our research into dynamic analysis methods is presented. In particular, we are interested in taint analysis, which can be seen as a subset of dynamic analysis methods, and its usage in a learning setting.

3.1 Extracting information

Taint analysis aim to extract run-time information about certain systems, commonly computer programs. The main usage of taint analysis revolves around the propagation of certain taint values, in either a control or data flow sense. It often involves modification of program assets or platform sources, since tainting is not inherently supported in most programming languages (except for Ruby and Pearl). Some of the challenging aspects of dynamic taint analysis are the following:

- The tainting process, interaction between tainted entities, and eventual analysis of taints, should *not* influence program behaviour, and
- Tainting should ideally be as resource efficient as possible.

The first item should make sense, since we intent to observe the output generation of a certain software component, not influence program behaviour or introduce other observable program artefacts. This is essential as we are generating a model based on the semantics of the interface of the model. If the taint analysis mechanics would interfere with the program's behaviour, the learned model would not be deemed correct. The second item concerns possible slowdowns of the run-time speed of the software component, when taint analysis methods are applied to its internals. While relatively slow tainting techniques that reduce the number of inputs required are definitely desired, as the complexity of the program provides a greater challenge than the run-time, the overhead should be within certain bounds of the component's performance decrease. Concluding, ideally an implementation is found that satisfies the aspects mentioned in this paragraph.

3.2 Information flow

In a security setting tainting is typically an information flow problem, regarding the propagation between input and output values. If a certain program value has been entered where there could be malicious intent, such as user input, then this value should be tainted. When this tainted source value interacts with any other instance of the program assets, the tainted flag should be duplicated to this other value. Most often this involves some propagation function that decides how taint marks are distributed between interacting variables. This way you can check during, or before, run-time execution whether there is some flow of information from source inputs that could cause program havoc. The objective then is to make sure possibly malicious inputs from source states cannot enter "sink states", that is, states that perform actions which could be used for malicious intent (I/O, system calls, etc.) [8]. Taint flags are added to program variables and propagated on interactions. You could say that tainted variables "infect" other values with the taint flag, therefore you can observe the spread of tainted input values. The results of this information flow analysis, is that after termination of the program the taint flags of the input and output variables can be analysed, and a trace of input to output variables can be constructed. Commonly, this only includes the information that variable x *interacted* with variable y. When y is an output variable and x and input variable we say that y has a dependency on x. Ideally, we would also know the actual interaction between the two values, how y can be constructed from x and possibly other values in the component. We would then focus on constructing constraints on external variables, i.e. I/O variables, not internal program variables.

3.3 Symbolic execution

Symbolic execution is a software execution method where program variables are not *concretely* instantiated but *symbolically*. This means that the variables do not represent a concrete parameter, but rather a symbolic data value. Symbolic values are propagated throughout the executed program without actually directly influencing program execution. For example, an `if`-statement inducing some control flow on a symbolic variable would introduce the separate execution of two different paths. One path would have the constraint of the `if`-statement added to its list of guards (with or without a symbolic variable as argument), while the other execution path would have the negation of the constraint of the `if`-statement added to its guards. The latter execution path would then also skip the semantics of the body of the `if`-statement, since the assertion of the guards would render the statement unreachable. This automatic execution method has a high coverage of test inputs [9], since traversing all program paths creates a list of input constraints that can be translated to a list of concrete input values. Meaning all control statements should be methodically reached.

In the research setting of model learning, the path constraints PC that are returned from symbolic execution frameworks such as the Symbolic Path Finder (SPF) [10] can be reasoned about. In particular, we would translate the path constraints returned from the symbolic execution run to actual I/O (input output) constraints. The learner focuses on external variables, i.e. input and output variables instead of internal variables. You can translate a constraint on internal and external variables to a I/O constraint by quantifying the internal variables. The resulting constraint would then apply to either the input or output variable, which is information essential to feed to the learner. We would then evidently know what variables, and with what operations, influenced the outcome of the program's execution. For

```
void symbexc() {
    a = input();
    a = a * 5;
    b = a + 17;
    output(b);
}
```

Figure 3.1: Pseudo-code Java I/O program.

instance, instead of constructing counterexamples in the Equivalence Oracle in random fashion, e.g. the random walk algorithm from RALib [2], path constraints PC can be solved using an SMT solver, such as Z3, for a certain theory. Figure 3.1 is a pseudo-code example of a Java program that takes an input value and produces an output value. When we would execute said program symbolically we would retrieve a path constraints of the form of the following [example inspired by email communication with Frits Vaandrager (2020)]:

$$PC := a_0 = v_{in} \wedge a_1 = 5 * a_0 \wedge a_2 = a_1 + 17 \wedge v_{out} = a_2$$

Because we want to constrain I/O variables and not internal variables, we can quantify $(\exists_{a_0,a_1,a_2 \in V}\ PC\ (a_0, a_1, a_2))$ these and construct an I/O constraint:

$$v_{out} = v_{in} * 5 + 17$$

3.3.1 Concolic execution/testing

Difficulties in symbolic execution methods lie in path explosion. When programs reach a certain level of cyclomatic complexity, e.g. nested statements, the number of paths to explore becomes incredibly large. Instead of exploring every available path, which means diverging at every branch, a symbolic value can made concrete by solving the current path constraints with an SMT solver. The branch can then be executed normally as if the program had actually reached that state in plain execution. By instantiating the symbolic values, a depth-first approach to reaching all program states can be achieved since the amount of breadth of symbolic execution is limited by actually execution branch paths concretely. In learning, concolic execution is used. Symbolic paths are calculated from concrete execution. Before actually learning the SUL with queries, a concolic execution run would generate a list of path constraints, these path constraints would be suitable to translate to a list of information on the control flow operations extracted from the SUL. That is, generate a list of I/O constraints. Symbolic execution tools already make use of SMT-solvers to solve intermediate constraints. Another version is the so called concolic testing method. Usage is in input generation for test cases such that all possible program states are reached. Similarly in active automata learning, it can be used to generate counter examples to the hypothesis by traversing the SUL whilst gathering input values. Then these generated inputs, that are theoretically guaranteed to access all program states in the SUL, can be fed or traversed in the hypothesis model (e.g. register automaton) to see whether the inferred semantics of the interface matches the actual behaviour of the SUL.

3.4 Control flow on tainted inputs

As became evident earlier in this chapter, the most useful information about program execution is gathered when regarding control flow of the respective program. If you need to infer the behaviour of some program P, a great part is managing to actually reach all possible states of P. Part of the process of reaching new states is done by computing new input values. The constraints that were learned by previous runs can solved and provide input values that can learn new behaviour. Ideally, as we need to feed inputs (in the form of tree queries or otherwise) to the SUL anyway, constraints are recorded on inputs objects. We then do not need to do separate runs for path constraint generation and actual input. An efficient way to both learn similar constraints of the symbolic execution methods and to not incorporate an execution in advance of a run is to record comparisons on tainted objects.

3.4.1 Inferring semantics from comparisons

We can keep track of operations, i.e. binary comparisons, on the inputs we process from the input symbols of the tree queries. We can assign taint values to input values from the data symbols in the tree query, starting from zero and incremented by one every time. Then, if we have a list of comparisons made on an input value, and the associated taint values of these values we can construct constraints on the input value. Assume x_1

with taint value one was fed to the SUL, and in the same tree query variable x_2 is added and compared to x_1. We would then retrieve the comparison, e.g. $x_1 == x_2$. We would then know that these variables were tested on equality. With each query, we analyse these comparisons to construct constraints. If x_1 is actually *equal* to x_2, i.e. the program variables they represent are equal, we record a constraint that $x_1 = x_2$. Otherwise, $x_1 \neq x_2$. Also, due to the differing taint values for every input, even if we have more than two variables that all have the same concrete value, we still know which variables were actually compared.

Constraints from tainted membership queries

In Section 2.1.2, we discussed the MAT framework for learning. In a context based on [6] which in turn is implemented on SL*, the framework uses a *tainted* tree oracle and a *tainted* equivalence oracle. The extension is based on MAT but uses two external oracles that use membership queries to either form an SDT in the tree oracle, or a CE/accepted hypothesis in the equivalence oracle. Tree queries consist of several membership queries that use variables as their parameters and return yes/no as well as a set of constraints on the variables. The same goes for the tainted equivalence oracle, where the parameters of the input symbols are tainted. In doing this, the SDT from Figure 2.7 can be derived. The taint analysis shows us that variable from suffix `IAdd(p)` was compared to some value in the register of the SUL x_1. We can deduce from the comparison that the parameter and register value were checked for equality. It can then analyse that suffix `IAdd(p)`, where $p \neq x_1$ is not accepted and can finish the right side of the tree. Also, because of the analysis of the constraints, we know that after the suffix action `IGet(p)` for some parameter $p \neq x_1$ the check $p = x_1$ is done. It learns the non-accepting (right) state from the left sub-tree and can verify the accepting left side with action `IGet(p)` where $p = x_1$. Otherwise, without dynamic analysis, it would not be known why the "get" action failed and presumable multiple queries would be needed to derive that the parameter must equal the value in the register to produce an accepting data word.

Chapter 4

Java

4.1 Java context

The target of this thesis is a non-trivial set of Java systems. Because the domain of our learning setting is Java systems, we need to practice our dynamic taint analysis within the context of the Java programming language. In this chapter, we discuss our research in to tainting Java for learning purposes.

4.1.1 Inherent support & research context

Out of the box, Java does not have any pre-designed support to use tainting inside Java programs. As we discussed in the control flow Section 3.4.1, we want to record the relational and arithmetic operations that are applied to the input values we query to the SUL. This means we want to instrument or add some code when any of the relation and arithmetic operators are used on our input objects. This code should add some value that records the binary operation and the two objects involved, and store it in some data structure inside the object. Most academic and industry tools available to taint Java programs are focused on information flow.

Python reference

Earlier efforts for learning register automata using taint analysis were researched for the Python language [6], [11], [12]. In Python, comparison analysis was done via an available package `taintedstr`. A "tainted String" object `tstr` o basically consists of the tuple `(s, t, comps)`, where

1. s is the original String value,
2. t is the associated taint integer value,
3. *comps* is the list of comparison that consists of each comparison made on o. Each comparison $c \in comps$ consists of a tuple `(lhs, rhs, op)` where:

(a) `lhs` represents the object on the left side of the operation (copy of *o*),

(b) `rhs` the object on the right hand side, and

(c) `op` the operation between the two objects.

The `__eq__`, `__lt__`, ... functions that are called when relational operators are used are overloaded to:

1. Add the comparison type, and the two objects involved to the list of comparisons, and then
2. Return the usual relational method implementation of the Python String class as to not interfere with ordinary program behaviour.

4.1.2 Java design principles

Java, with its unique virtual machine bytecode interpreter design and widespread usage, does not always relate to other common programming languages such as Python and C++. Java is used in all sorts of systems, mostly because of the virtualisation approach of machine instructions and embedded object oriented design. These features make it portable and usable in many scenarios and settings. At the foundation of these features are some design principles that make one's life easier or harder, depending on the context. Java was designed with simplicity in mind, removing some features from its predecessor languages C and C++. At its core, Java is presented as a "small, simple language" that is "still sufficiently comprehensive to address a wide variety of software application development"[1]. In our case, where we would ideally translate the Python taint implementation to Java, the design principles of Java do not really benefit the process. Fortunately, there some ways to circumvent these limitation in practise.

Primitive types

Java primitive types, `int, boolean, float, long` for example, are not represented as objects in the language. They do not play a part in the `java.lang.Object` hierarchy, where every instance of a Java class is a subclass of the Object class. There are primitive wrapper classes however, such as `Integer, Boolean, Float`, that all extend the `java.lang.Number` class. In turn, the `java.lang.Number` class extends the `java.lang.Object` class. These primitive wrapper classes encapsulated primitive types in Java objects, providing implementations of common Object methods (e.g. `toString()`) and methods with extended functionality (e.g. `Integer.valueOf(java.lang.String)`, which returns the Integer value of a String value). In ordinary Java, there

[1]As read on `https://www.oracle.com/java/technologies/simple-familiar.html`

is no way to explicitly interact with primitive types. In most cases primitive types are either boxed to the wrapper classes or vice versa, where the wrapper classes are unboxed to the primitive type inside the wrapper. This automatic (un)boxing is done by the Java compiler, naturally during compile-time. In Figure 4.2, we can see the compiled bytecodes of a nu-

```
public class Example {
  public Example() {}

  public boolean comp(Integer a, Integer b) {
    return a > b;
  }
}
```

Figure 4.1: Example of Java compiler unboxing when comparing two instances of `java.lang.Integer`.

meric comparison. After the variable is pushed to the stack, the compiler optimises such that virtual method `Integer.intValue()` is invoked on the `Integer` objects. The relational comparisons is thus called on the primitive int type, not on the wrapper class.

```
Compiled from "Example.java"
public class Example {
  public Example();
    Code:
       0: aload_0
       1: invokespecial #1  // Method java/lang/Object."<init
    >":()V
       4: return

public boolean comp(java.lang.Integer, java.lang.Integer);
    Code:
       0: aload_1
       1: invokevirtual #2  // Method java/lang/Integer.intValue
    :()I
       4: aload_2
       5: invokevirtual #2  // Method java/lang/Integer.intValue
    :()I
       8: if_icmple     15
      11: iconst_1
      12: goto          16
      15: iconst_0
      16: ireturn
}
```

Figure 4.2: Compiled byte-code instructions of Figure 4.1

Final Classes

Possibly induced by this behaviour, all the primitive wrapper classes of Java that represent Integers and Strings et cetera are protected by the `final` keyword. These classes are immutable and can not be extended or sub-classed. Java as a language, contrary to most other languages, limits the freedom of expressing standard classes as super-classes, either from a security or maintainability principle. We can however produce a variant of the Oracle JDK or OpenJDK by compiling our own `.java` source files and re-archiving these to the same structure as the original .jar they originated from. If using the OpenJDK, which is open source, a possible solution would be to compile a patched JDK from source. Alternatively, we can just override the respective .jar, in the case of java.lang.Integer this is rt.jar, from the JDK with our modified version. Patching the JRE is done by specifying the boot classpath (`-Xbootclasspath:<directories and archives>` at run-time, the classes from the modified .jar will then override the default JRE source class files. This is supported for JDK's up to and including JDK-8, since from 9 on out the JDK has become modular. The `--patch-module` flag, which should be able to patch the `java.base` modules and provide the same behaviour as the boot classpath option, interferes with our addition of a compiler plugin[2]. Therefore, in the case of JDK-9+, a patched OpenJDK can be built. Then, we can extend the Integer class and add our own code for tainted Integer objects. Modifying JDK library classes is also how Chin et al. managed to propagate taints on String's and String related builders/buffers in 2009 [13].

4.1.3 Library components and generics

Although the extension of Java's wrapper classes is highly discouraged, it is necessary to capture and store the relations we retrieve from operators, as well as the control over the `equals`, `compareTo` and `hashCode` functions for our tainted Integer class. Especially if the target model is one of Java's library components. Because of generics, comparison heuristics in the `equals` and `compareTo` methods need to be tracked and the `hashCode` needs to be overridden to always return the same value in order to learn the Java library classes from `java.util` that use the hash function. Most of the comparisons to be recorded from these library classes is based on the equality of objects, to be gathered in the equals method. If the target system makes use of the `hashCode` method, it should always return the same value in order to ensure tainted Integer's are actually compared on equality instead of the value of the hash.

[2]This was patched as a result of our findings in Manifold `2020.1.24`, making the `--patch-module` option viable again.

Operator Overloading

Operator overloading is something available in most industry-wide languages, but not in Java. Java does not allow user-defined overloaded operators by design. Operator overloading from programming languages has both advantages and disadvantages. Some of the advantages lie in the fact that they allow programmers much more freedom in expressing semantics of their programs. One of the disadvantages that comes with freedom is that it introduces a whole range of possible misuses of the language. This in turn often leads to confusing and buggy code. On the other hand, one could argue that the misuses introduced by operator overloading can be equated with any other custom public method behaviour. The only form of overloading available is method and class overloading. Method overloading is recommended for similar behaviour to overloading operators (e.g. implementing and overloading custom `greaterThan(T other)` methods), but is not supported for Java's primitive relational and arithmetic operators. Exceptions to the above paragraph are String concatenation, as well as some other operations on primitive Boolean and integral types. These however are a form of internal overloading. Resulting, there are no possible ways to override operators in vanilla Java. Thus, we need to look in to instrumenting and the manipulation of source code and the compiler instance of the JDK.

There are however some external modifications to incorporate overloading in to the language. The original implementation for Java was built for specific versions of the JDK and is not actively maintained. The following paper [14] was published alongside the release of the compiler extension, and includes semantics of overloading operators in Java. The implementation can be summarised as follows, the jar library produced by the operator overload project is added to the class path of the Java compiler. The library uses a form of annotation processing to attach itself to the compilation procedure. There the extensions of the compiler classes are added to the respective compile stages. This implementation, although interesting and unique at the time, was not really versatile as it had to be developed for specific versions.

4.2 Tainting Java

4.2.1 Java library classes

As we hinted at in Section 4.1.3, learning library components that make use of generics force us to look at the set of {equals, compare} from the Object class and Comparable interface respectively. We can gain knowledge on how to taint these components by looking at their source code. Operators are never called on generics, because evidently the type is unknown. We can however extract useful information from these components. If we take the `HashSet` for example again, we can learn the constraints on the behaviour of its internal code by recording the equals method call. If we set the `hashCode` of every tainted Integer to 42, the objects **must** be compared for equality. In doing this we can record these equality comparisons. Another example is when we would look at a `PriorityQueue`, which uses either the natural ordering of the parameters it receives as input or the order as specified by a custom `Comparator`. From the source code we can see the code snippet from Figure 4.3, which shows the code of the method call the queue executes when an item is inserted. We see that the comparison of the items is evaluated as greater than or equal to zero. From this, we infer that the item to be inserted is compared to items in the queue with the $\geq$ relation. Capturing these inferred comparisons is done by specifying a custom Comparator as argument to the queue instantiation. The code snippet below shows how

```java
private void siftUpUsingComparator(int k, E x) {
    while (k > 0) {
        int parent = (k - 1) >>> 1;
        Object e = queue[parent];
        if (comparator.compare(x, (E) e) >= 0)
            break;
        queue[k] = e;
        k = parent;
    }
    queue[k] = x;
}
```

Figure 4.3: Internal "upsifting" method of `java.util.PriorityQueue`

such an instantiation would look like:

```java
new PriorityQueue<TaintInteger>((taintL, taintR) -> {
    Comparison<?,?> c = new Comparison<>(taintL, taintR, "GE");
    taintL.getComparisons().add(c);
    return taintL.compareTo(taintR);
});
```

Some of the limitations of these kind of white-box methods to capture relations from compare methods is that in a `PriorityQueue` the same procedure is also used for the "sifting" down method. Only, in the downwards movement the result of the compare method is compared as less than or equal to zero. This would imply a "LE" comparison would need to be captured in this case, instead of "GE". Finding out what relation is used on the result of the compare method is a hard problem.

4.2.2 Manifold System & Operator Overloading

The Manifold System[3] is a Java Compiler plugin that extends the Java language with extra features, support the JDK/JRE range of 8 to 14. It adds features to the language by manipulating the default compiler using the defined Java compiler plugin framework. Therefore it is a more versatile approach than [14]. It is specified at compile-time to the compiler via the `-Xplugin:Manifold` option. Using Manifold's compiler plugin, we can overload relational and arithmetic operators for custom classes by implementing the `CompareToUsing<T>` interface. Relational operations that are overloaded by Manifold are handled via the `compareToUsing(T other, Operator op)` method, where the comparison of the object is handled via the `compareTo` method from the default `java.lang.Comparable` interface. Equality (`==` and `!=`) is then handled by one of three possibilities: the return boolean of the `equals()` method, whether the return from the `compareTo()` method was zero, or the reference equality `==`.

```
public class TaintInteger extends Integer implements
    ComparableUsing<Integer> {
     ...
     @Override
     public boolean compareToUsing(Integer that, Operator op) {
          this.getComparisons().add(new Comparison<>(this, that,
    op.toString()));
          return ComparableUsing.super.compareToUsing(that, op);
     }
     @Override
     public EqualityMode equalityMode() {
          return EqualityMode.CompareTo;
     }
     ...
}
```

Figure 4.4: Methods overridden from the Manifold `compareToUsing` interface.

We can implement the Manifold interface and override the operator function for our tainted objects as shown in Figure 4.4, as well as implement the override-able `equals` method. Doing so makes us able to record what relational operations were called on the object. In the case of the tainted tree oracle, which supports equality constraints currently [6], we capture the relations on the tainted object we instantiate out of the parameters we receive from the learner. When a program call in a `HashSet` is done for equality on objects, this is registered. If x_1, x_2 are tainted Integer's with taint value one and two respectively, and they represent some arbitrary concrete value,

[3]See `http://manifold.systems/`

and $x_1.equals(x_2)$ is a call from the internals of the `HashSet` to check for equality, we can deduce the constraint $x_1 = x_2$ or $x_1 \neq x_2$ by checking the comparisons of the tainted Integer's.

Algorithm 1: Constraint generation of comparisons made in SUL.

```
Data: List of tainted parameters $\mathcal{P}$ from an action and list of
      tainted parameters currently in the SUL $\mathcal{S}$
Result: List of constraints $\mathcal{C}$
Initialise $\mathcal{C}$;
for $p$ in $\mathcal{P}$ do
    for $c$ in $p.comparisons$ do
        $l_t$ = c.left.taint;
        if c.right is tainted then
            $r_t$ = c.right.taint;
            if c.left and c.right in $\mathcal{S}$ then
                if not $l_t == r_t$ then
                    $\phi = (l_t, r_t, \text{inverse(c)})$;
                    Add $\phi$ to $\mathcal{C}$;
                end
            end
        else
            $\phi_c = (l_t, \text{c.right}, \text{inverse(c)})$;
            Add $\phi_c$ to $\mathcal{C}$;
        end
    end
end
return $\mathcal{C}$;
```

The actual comparison is stored in x_1. When we construct the constraints as demonstrated by Algorithm 1, we loop over the list of tainted inputs from the list of input actions, enter a second loop of the list of comparisons per tainted object and construct the derivable constraints. If both sides of the comparison equate to a tainted object, we can compare the taint values and construct a constraint on the operation, otherwise we encountered a constant (Integer). We inverse the operation as show in Algorithm 2: if the operation between the (concrete) values of the (tainted) objects yield true, then the comparison is added to the set of constraints as is. Otherwise the inversion of the relational operations is added as the constraint. If for example we have the comparison $x_1 <= x_2$ recorded in the program, where the taint values are the variable index and $x_1 = 7$, $x_2 = 5$, then the constraint would be $x_1 > x_2$ because $7 \not\leq 5$.

Algorithm 2: Construct constraint

```
Data: A comparison $c$ from a tainted parameter
Result: An operation $\omega$
l = c.left, r = c.right, op = c.op;
if the operation on l and r yields true then
    $\omega$ = op;
else
    $\omega$ = invert(op);
end
return $\omega$
```

Due to the incremented nature of the taint values, i.e. no variable has the same taint value, a trace of constraints between the variables can be constructed. Whether two different inputs had the same concrete assignment does not influence the trace.

4.2.3 Complications of tainting operations

```
if((!x1.equals(x2)) || x2 <= x3) {
    // do something
}
```

Figure 4.5: The inverse operator '!' and possibly the RHS of the "or" are out of reach relations on tainted objects.

There are some inherent complications when using relational dynamics on tainted objects in constraint generation. Consider the code example of Figure 4.5, there are two operation calls that can be out of reach for an implementation based on object tainting. First of all, the `!` operator inverting the results of the equals call is not recorded by any of the tainted objects in the equation. Secondly, as a result of Boolean optimisation, the RHS of the `||` ("or") operator is not evaluated when the LHS equates to `true`. Rather the valuation is short circuited because the RHS has no effect on the outcome of the valuation, since the LHS already valuates the entire equation to `true`.

4.2.4 Arithmetic

Future learning algorithms might be able to wonder about complex relations like (non-)linear arithmetic. Symbolic execution methods as discussed in Section 3.3 allow us to derive input constraints from operations such as arithmetic. In the example of Figure 3.1 we showed how a symbolic execution method would come to this constraint. In Figure 4.6, we see how we can use the Manifold Operator Overloading to also capture arithmetic operations. We can see that $42 = 17+25 = 17+5*5 \Rightarrow v_{out} = 17+5*v_{in}$. Similarly to the symbolic execution method, we can reason about the arithmetic relations between input variables. Arithmetic operators are overloaded with

```
TaintInteger a = new TaintInteger(5, 0);
a = a * 5;
a = a + 17;
printComparisons(a);
```

```
Object Value: 42, Taint Value: -1
Left: Object Value: 5, Taint Value: 0 | Right: 5 | Operation: *
Left: Object Value: 25, Taint Value: -1 | Right: 17 | Operation:
    +
```

Figure 4.6: Source code and output of arithmetic comparisons.

the Manifold plugin by defining methods for `add`, `times`, `div` and `minus`. These methods are structurally defined, you can overload these operators separately for every type you wish to support the operation for.

4.2.5 Limitations

The limitations of this implementation are the following:

- **Unboxing:** If the compiler receives two Integer's (or instances of other wrapper classes), these are unboxed and compared on primitive level. The overload from the Manifold plugin does not reach the primitive operation between `int`'s. Subsequently if some function f receives two Integer parameters a and b, which are actually tainted Integers, to compare with some operator the compiler still translates the Integer's to their primitive value. Possible solutions involve compiler modification, such that the unboxing of these Integer's does not happen. A problem that would then occur is that the arithmetic of the wrapper classes is defined as the arithmetic of their primitive values. Because of this, the modification of the compiler must only modify unboxing when a subclass of the `java.lang.Integer` is found (e.g. a `TaintInteger`). Leaving ordinary Integer comparisons defined by their primitive counterpart.

- **Primitive types:** If a program requires a primitive type of input and a wrapper class is supplied, the same unboxing happens as discussed above. Also, when we define the operator overload for a sub-class of the Integer class, overloading of the comparative operations between the Integer sub-class and other primitive types (`double`, `long`, ...) is not defined. If a component is learned within a numeric domain other than the set of integers, a solution is needed. Possible resolutions for this problem are to extend the `java.lang.Number` class which super-classes al numeric (wrapper) classes. Then, the overload of the operators would be defined for all instances of the `Number` class.

- **Shortened operators:** In programs involving arithmetic operations are often shortened, i.e. `a = a + b` is written as `a += b` where `+` could be any operator from the set of Java operators. In Java, this is a form of syntactic sugar that also involves a cast. Unfortunately, the Manifold system currently does not reach the operations in this syntax. Resolving this would involve the extension of the Manifold plugin.

Concluding, we can not record all of the desired comparisons as of yet. For all of the limitations a possible solution and reconsideration is presented. A more brute approach to solving the problems would be to change the types of the SUL from `Integer` to `TaintInteger`. Whilst this would solve most of the problems, and program behaviour would be identical due the nature of our presumably purely additive modifications, this would involve the modification and recompilation of the source code of the software component to be learned which is generally not very versatile and not a good practice when inferring semantics of software components.

4.3 Bytecode & ASM

In this section we will look at the bytecode level of comparisons in Java programs, and how these bytecode instruction might be able to provide an alternative implementation to the one we implemented with the use of operator overloading and standard Java methods.

4.3.1 Bytecode comparisons

Java source code generates byte-codes when compiled and runs on the Java Virtual Machine, which translates the abstract byte-codes to machine code. There are numerous different instructions, some of which are of interest to us when executed. In particular, the byte-codes that introduce branches such as `if_icmp<condition>`[4]. If we look at the compiled boxing example of the code example of Figure 4.2, we see that actual comparison of the byte-code instruction `if_icmple` is done on the primitive types of the Integer class. First, the arguments of the method are loaded on to the stack, and, as we discussed in the boxing section are unwrapped to their primitive value. Then, the `if_icmple` instruction is executed to compare the two top most

[4]See "6. The Java Virtual Machine Instruction Set" at `https://docs.oracle.com/javase/specs/jvms/se14/html/index.html`

variables on the stack. The result of this instruction decides whether a constant zero or one is pushed to the stack before returning (the constant). The primitive Boolean type is represented as either zero (false) or one (true).

4.3.2 Phosphor

Phosphor[5], “Dynamic Taint Tracker for the JVM” [15], is a instrumentation tool that can be used to instrument JDK/JRE’s to enable these to have a taint tracking feature. It works by instrumenting the source files with additional tags, and uses the ASM library API[6] to propagate taint values during run-time at the bytecode level. It does this by analysing the instructions called, and pushing the taint values of the object to the stack in order to propagate these when they interact. If the control tracking extension is enabled, Phosphor also propagates taint markings to values that depend on the branch statement. The propagation (control and data flow options) only includes information dependencies however, so if v_t is a tainted value, then there is a list of dependency variables $dep(v_t) = \{v_0, v_1, \ldots, v_n\}$ for all n variables that v_t interacted with. This does not include the relational dependencies between the variables, only that variable v_t depends on the value of some arbitrary value v_i from the set. If the variables from the set $dep(v_t)$ are also tainted, thus have a list of dependencies, a recursive scheme of dependencies is formed. This scheme can be used to interpret the variables that influenced v_t. To illustrate, we can construct an output $\{v_t \odot v_0,\ v_0 \odot v_1,\ \ldots\ ,\ v_{n-1} \odot v_n\}$, where $\odot$ is some unknown relation between the variables and variables $v_0 \ldots$ are parents of v_t in the recursive dependency scheme. If the relation between the variables is known, then input constraints can be derived from the dependency relations.

An alternative to our implementation would be to override Phosphor’s `ControlFlowManager` class, and push an object to the stack that contains the relation, as derived from the bytecode instruction, and the variables that were on on the stack when this instruction was called. Limitations include the complexity of such an implementation compared to the overload method, and the slowdown of Phosphor’s control flow instrumentation. Adding on top of the already slow propagation logic would induce even more overhead. Some possible resolution would be to limit the extensive propagation logic of Phosphor in a custom extension of Phosphor’s code base, only focusing on the information necessary to us. That is, the information necessary to construct a constraint on (input or output) parameters for specific types or data structures.

Alternatively, a custom dynamic bytecode analyser could be developed. One possible implementation would be to create a custom instance of ASM’s `MethodVisitor` class, to analyse the instructions of the methods of the SUL.

[5]Found at `https://github.com/gmu-swe/phosphor`.

[6]`https://asm.ow2.io/`

This approach would be similar to RALib's class-analyser, although the analyser is built on the restrictive interface of the Java Reflection API. Thus, with ASM, dynamic analysis could be accomplished by building on top of a similar system to the class-analyser but with the inclusion of analysis of comparisons on variables in the instruction source of the SUL.

Chapter 5

Experiments

5.1 Introduction

In order to test our implementation of tainting for Java, we run some benchmarks and compare the number of inputs to the number of inputs RALib [2] uses without any tainting. Instead of the black-box method of RALib, the setup from this thesis uses tainted inputs when querying the membership queries necessary to deduct an SDT as also seen in [6]. As for the equivalence oracle, both instances use the original random walk equivalence oracle from RALib. The equivalence testing part is thus equal for both instances. Hypothetically, when the optimisations of the two implementation are of equal significance, our tainted tree oracle should outperform the black-box oracle of RALib in learning the libraries from `java.util`. In particular, we will focus on a FIFO and LIFO (`LinkedList`) model and a SET (`HashSet`) model.

5.2 Set-up

In their paper introducing RALib [2], they used a maximum artificial size of three for their models. In our setting, we use a SUT that encapsulates the model and translates (input/output) actions to actual execution calls. Subsequently, in order to compare the results, our SUT's are also set to a maximum size of three. The FIFO and LIFO models are represented as a `java.util.LinkedList`, the SET model is a `java.util.HashSet`. The tainted implementation makes use of equality relations on input parameters. We do not consider execution times in these experiments. In order to evaluate the execution times rightfully, one must consider that the current set-up to learn Java components from tainted inputs uses a rather latency-heavy construction. As we stated in the preliminaries, most is gained from minimising the amount of interactions with the model, i.e. inputs and resets. The benchmarks run on twelve different seeds, where the only optimisation method enabled on both instances is a cache. Some of the SET-03 sets were not learned by the non-tainted implementation as they timed-out. Additionally, the LIFO-03 model with random seed 20 was not learned by both instances of the tree oracle. These cases have not been included into the tabled and graphed results. An input represents the handling of an action, a reset represents the clean slate of the system before a membership query is processed.

5.3 Results

Model (locs/trans)	Tree Oracle	Inputs Mean	Inputs Std. Dev.	Resets Mean	Resets Std. Dev.
FIFO-03	RALIB	1.54e+03	8.52e+02	2.80e+02	1.36e+02
(14/16)	TAINTED	5.19e+02	6.82e+01	1.42e+02	1.45e+01
LIFO-03	RALIB	8.42e+03	1.30e+04	9.37e+02	1.20e+03
(14/28)	TAINTED	6.36e+02	1.65e+02	1.62e+02	2.70e+01
SET-03	RALIB	4.39e+06	9.45e+06	3.63e+05	7.65e+05
(14/16)	TAINTED	2.79e+04	2.93e+04	2.74e+03	2.10e+03

Table 5.1: Number of inputs and resets and total actions on the models for RALib with and without the tainted tree oracle. These totals include the inputs and resets for both the membership and test queries.

The benchmarks run for every seed and twice for every model. Once with the tainted tree oracle and once with RALib's black-box oracle. The equivalence oracles are both from RALib, this also means that the testing phase produces the near exact same results in terms of inputs and resets. Difference from the number of inputs and resets comes from the difference in the tree query process.

5.4 Discussion

As we can see from Table 5.1 and Figure 5.1, the tainted tree oracle drastically improves on the black-box method of RALib, greatly reducing the number of interactions on the model. The results of the tainted oracle on FIFO-03 and SET-03 are very similar to those of the Python taint implementation from Garhewal et al.[6], the reason is clear: the Java implementation is based on the framework that was produced in Python and uses a great part of the code that was developed. Essentially, in Java we execute the membership queries and return the comparisons and whether the query was accepted to the Python implementation for further processing. Moreover, the Java version is thus also able to learn the FIFO-05 model with the tainted tree oracle. Part of the reason for the significant difference between the results of the black-box tree oracle and the tainted tree oracle is that the black-box oracle of RALib has a hard time finding the necessary queries within the constraints on inputs compared to our implementation. Also, much of the improvements of RALib are realised by their optimisation techniques and their custom class analyser. In these benchmarks, we focus purely on the difference between a black-box (RALib) and a grey-box (tainted) setting. Concluding, these results indicate that the tainting technique proves significant improvement over the black-box learning of the three measured models.

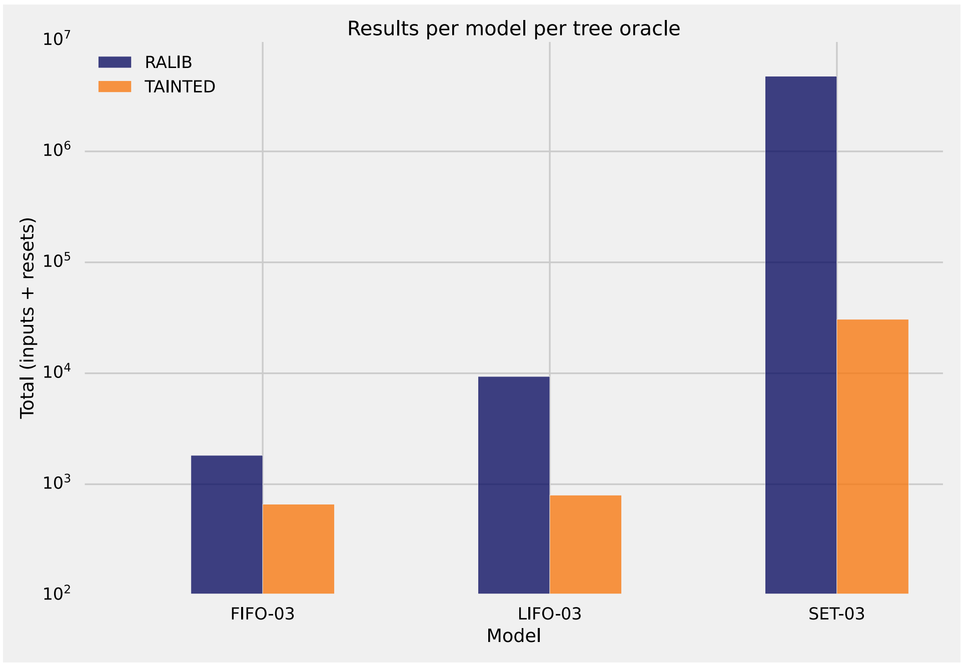

Figure 5.1: Graph with results from tainted and RALib tree oracle per learned model, y-axis is logarithmic.

Chapter 6

Related & Future Work

6.1 Related Work

There have been a wide variety of works focusing on tainting Java (or the JVM/JRE) or using dynamic analysis methods or symbolic execution for learning purposes. In a learning setting, our input constraints derived from comparisons on tainted inputs is best compared to procedures that incorporate symbolic execution methods. As stated previously, most practices of dynamic tainting methods are in a security context. Difference is made between static and dynamic taint analysis, static analysis focuses on application source code to find the information flow risks of all execution paths. Dynamic taint analysis focuses on dynamic analysis in a run-time environment, tracking taints on actual input. In Tainer [16], a similar approach to Phosphor [15] of byte-code taint instrumentation is used on top of ASM. Both in the user's Information flow of taints in handled by creating a new field for every object of a class that concerns its tainted state. However, as it is an information flow taint tracking framework similarly to Phosphor, the use-case does not correspond to input constraint generation. In Sigma* [17], they extend Angluin's L^* algorithm [3] as Σ^* to symbolically execute targeted components. Input constraints and the terms generating output values are derived from the symbolic path predicates. Scaling is out-of-the-box supported by the dynamic symbolic execution, as it does not require a concrete input alphabet. Abstractions of program behaviour are built however, which require algorithmic processing to derive finite state machines. Moreover, symbolic execution methods are costly [1] compared to the lightweight dynamic taint tracking methods of input values also demonstrated in this thesis. In PSYCO [18], L^* is also combined with symbolic execution. The symbolic execution instance is based on the Java PathFinder (JPF) [10], and is developed as an extension of JPF named `jpf-psyco`. None of the works on dynamic input constraint generation for Java are based on the overloading of operators.

6.2 Future Work

This thesis manifests the portability of the tainting implementation to Java. With that, it become evident that the dynamic analysis method on comparisons improved on the methods from RALib. It also shows that tainting is in some ways more complex in languages other than Python, since the language properties of looseness in type and overload restrictions can not be relied upon in every other language. Some of the future work stems from the limitations mentioned in Section 4.2.5, while others range to problems associated with the tainting implementation chosen in this thesis. First of all, when the direction of overloading is deemed promising enough, the limitations of the current implementation need to be lifted. Currently, the tainting implementation only suffices for equality relations. Complications with the interaction between the overloaded operators and the primitive types in Java need to be further investigated. Either an extensions of the Manifold plugin must be made or other means of compiler modification must be sought in order to properly deal with the issues related to primitive types in Java. Also, possible further endeavours are to investigate the option to use an implementation of the Comparable class to find relations in a PriorityQueue for example. Currently, the tainted tree oracle only supports the presence of equality relations and respective constraints. When tainting for Java additionally fully supports the recording of other relations examined in this thesis, the tree oracle needs to be extended to also account for these. Possibly another direction must be examined, as, after providing the limitations in Section 4.2.5, we hinted at other implementations in the subsequent section in chapter four. These include an extension of Phosphor to record the instruction of the comparison atop of the already implemented taint propagation, or a custom framework either analysing AST's from the Java compiler or byte-code instructions from methods with ASM. Furthermore, an initial step towards a re-implementation of the tainted equivalence oracle [12] was made in Java. The current state however was too buggy to include in the experiments. Efforts should be made to debug, and possibly extend with richer relations, the tainted oracle for the Java language.

The above paragraph focused mostly on the techniques discussed in this thesis. In Section 3.3, we discussed symbolic execution methods. The direction of this thesis was mostly fixed on capturing comparisons from inputs in Java, which was accompanied with quite a number of technical obstacles. A future endeavour for tainting Java in a learning setting might benefit from an approach focusing on symbolic execution.

Chapter 7

Conclusions

We have shown that we are able to port the dynamic taint analysis methods to the Java language domain, be it with several limitations. Possible solutions to these limitations have been given, as well as options for alternative implementations for achieving the same goal. We manifested how a compiler plugin modification and extension of Java source classes can enable us to achieve I/O constraints from tainted Java objects, and learn real sets and (F|L)IFO models from their `java.util` library representation. Additionally, we have demonstrated how we can extend the taint analysis to keep track of arithmetic, and the relation to symbolic execution methods. We can thus conclude that the grey-box taint analysis learning framework is also suited for Java, one of the most used languages in the industry. In doing so we also connected our tainting implementation, together with the framework from Garhewal et al. [6], to the register automaton learning library RALib [2]. This means that the framework in theory could also be used to extend taint analysis to other widely industry-used languages such as C++. As from the experimental evaluation: compared to the black-box approach of RALib, the grey-box methods does significantly better in constructing models from the Java library in terms of the number of inputs and resets required. However, as the solution was proven not to be able to fully capture all desired comparisons in its current state, there is still work to be done to improve upon the tainting implementation for Java.

References

[1] F. Howar, B. Jonsson, and F. W. Vaandrager, “Combining black-box and white-box techniques for learning register automata,” in *Computing and Software Science - State of the Art and Perspectives*, ser. Lecture Notes in Computer Science, B. Steffen and G. J. Woeginger, Eds., vol. 10000, Springer, 2019, pp. 563–588. DOI: `10.1007/978-3-319-91908-9\_26`. [Online]. Available: `https://doi.org/10.1007/978-3-319-91908-9%5C_26`.

[2] B. J. Sofia Cassel Falk Howar, “Ralib: A learnlib extension for inferring efsms,” in *International Workshop on Design and Implementation of Formal Tools and Systems*, 2015.

[3] D. Angluin, “Learning regular sets from queries and counterexamples,” *Inf. Comput.*, vol. 75, no. 2, pp. 87–106, 1987. DOI: `10.1016/0890-5401(87)90052-6`. [Online]. Available: `https://doi.org/10.1016/0890-5401(87)90052-6`.

[4] R. Alur, P. Cerný, P. Madhusudan, and W. Nam, “Synthesis of interface specifications for java classes,” in *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, J. Palsberg and M. Abadi, Eds., ACM, 2005, pp. 98–109. DOI: `10.1145/1040305.1040314`. [Online]. Available: `https://doi.org/10.1145/1040305.1040314`.

[5] F. Howar, M. Isberner, B. Steffen, O. Bauer, and B. Jonsson, “Inferring semantic interfaces of data structures,” in *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*, T. Margaria and B. Steffen, Eds., ser. Lecture Notes in Computer Science, vol. 7609, Springer, 2012, pp. 554–571. DOI: `10.1007/978-3-642-34026-0\_41`. [Online]. Available: `https://doi.org/10.1007/978-3-642-34026-0%5C_41`.

[6] B. Garhewal, F. Vaandrager, F. Howar, T. Schrijvers, T. Lenaerts, and R. Smits, “Grey-box learning of register automata,” *To appear in Pro-*

ceedings 16th International Conference on integrated Formal Methods (iFM), pp. 16–20, 2020. [Online]. Available: `https://www.sws.cs.ru.nl/publications/papers/fvaan/TaintingRALib/document.pdf`.

[7] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340, ISBN: 978-3-540-78800-3.

[8] V. Haldar, D. Chandra, and M. Franz, "Dynamic taint propagation for java," in *21st Annual Computer Security Applications Conference (ACSAC 2005), 5-9 December 2005, Tucson, AZ, USA*, IEEE Computer Society, 2005, pp. 303–311. DOI: `10.1109/CSAC.2005.21`. [Online]. Available: `https://doi.org/10.1109/CSAC.2005.21`.

[9] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later," *Commun. ACM*, vol. 56, no. 2, pp. 82–90, 2013. DOI: `10.1145/2408776.2408795`. [Online]. Available: `https://doi.org/10.1145/2408776.2408795`.

[10] C. S. Pasareanu and N. Rungta, "Symbolic pathfinder: Symbolic execution of java bytecode," in *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*, C. Pecheur, J. Andrews, and E. D. Nitto, Eds., ACM, 2010, pp. 179–180. DOI: `10.1145/1858996.1859035`. [Online]. Available: `https://doi.org/10.1145/1858996.1859035`.

[11] T. Schrijvers, "Learning register automata using taint analysis," Bachelor's Thesis, Radboud University, 2018.

[12] R. Smits, *Software science research internship*, 2019.

[13] E. Chin and D. A. Wagner, "Efficient character-level taint tracking for java," in *Proceedings of the 6th ACM Workshop On Secure Web Services, SWS 2009, Chicago, Illinois, USA, November 13, 2009*, E. Damiani, S. Proctor, and A. Singhal, Eds., ACM, 2009, pp. 3–12. DOI: `10.1145/1655121.1655125`. [Online]. Available: `https://doi.org/10.1145/1655121.1655125`.

[14] A. Melentyev, "Java modular extension for operator overloading," *CoRR*, vol. abs/1406.4087, 2014. arXiv: `1406.4087`. [Online]. Available: `http://arxiv.org/abs/1406.4087`.

[15] J. Bell and G. E. Kaiser, "Phosphor: Illuminating dynamic data flow in commodity jvms," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, A. P. Black and T. D. Millstein, Eds.,

ACM, 2014, pp. 83–101. DOI: 10.1145/2660193.2660212. [Online]. Available: https://doi.org/10.1145/2660193.2660212.

[16] M. Ashouri, "Practical dynamic taint tracking for exploiting input sanitization error in java applications," in *Information Security and Privacy - 24th Australasian Conference, ACISP 2019, Christchurch, New Zealand, July 3-5, 2019, Proceedings*, J. Jang-Jaccard and F. Guo, Eds., ser. Lecture Notes in Computer Science, vol. 11547, Springer, 2019, pp. 494–513. DOI: 10.1007/978-3-030-21548-4_27. [Online]. Available: https://doi.org/10.1007/978-3-030-21548-4%5C_27.

[17] M. Botincan and D. Babic, "Sigma*: Symbolic learning of input-output specifications," in *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, R. Giacobazzi and R. Cousot, Eds., ACM, 2013, pp. 443–456. DOI: 10.1145/2429069.2429123. [Online]. Available: https://doi.org/10.1145/2429069.2429123.

[18] D. Giannakopoulou, Z. Rakamaric, and V. Raman, "Symbolic learning of component interfaces," in *Static Analysis - 19th International Symposium, SAS 2012, Deauville, France, September 11-13, 2012. Proceedings*, A. Miné and D. Schmidt, Eds., ser. Lecture Notes in Computer Science, vol. 7460, Springer, 2012, pp. 248–264. DOI: 10.1007/978-3-642-33125-1_18. [Online]. Available: https://doi.org/10.1007/978-3-642-33125-1%5C_18.

Appendix A

Appendix

A.1 Comparisons from TaintInteger

```
TaintInteger taintInteger = new TaintInteger(0, 0);
TaintInteger taintInteger1 = new TaintInteger(1, 1);
Integer integer = new Integer(0);
Integer integer1 = new Integer(1);
taintInteger.equals(integer);
if (taintInteger > taintInteger1) {}
if (taintInteger <= taintInteger1) {}
if (taintInteger > integer1) {}
if (taintInteger <= integer1) {}
if (taintInteger > 5) {}
for(Comparison<?,?> c : taintInteger.getComparisons()) {
    System.out.println(c);
}
```

```
Left: Object Value: 0, Taint Value: 0 | Right: 0 | Operation
: EQ
Left: Object Value: 0, Taint Value: 0 | Right: Object Value:
 1, Taint Value: 1 | Operation: GT
Left: Object Value: 0, Taint Value: 0 | Right: Object Value:
 1, Taint Value: 1 | Operation: LE
Left: Object Value: 0, Taint Value: 0 | Right: 1 | Operation
: GT
Left: Object Value: 0, Taint Value: 0 | Right: 1 | Operation
: LE
Left: Object Value: 0, Taint Value: 0 | Right: 5 | Operation
: GT
```

Figure A.1: Demonstration of captured comparisons on tainted integers. Above is the code, below is the output.

A.2 Symbolic Execution Example

```
void main(String[] args) {
    int x = input()
    int y = input()
    foo(x, y);
}

void foo(int a, int b) {
    c = 2*a;
    if (c == b) { // if (2*a == b)
        if (a + 10 > 2*b) {
            output()
        } else if (a*b > a+b) {
            throw new RuntimeException();
        }
    }
}
```

Figure A.2: Java program to run Symbolic Execution on.

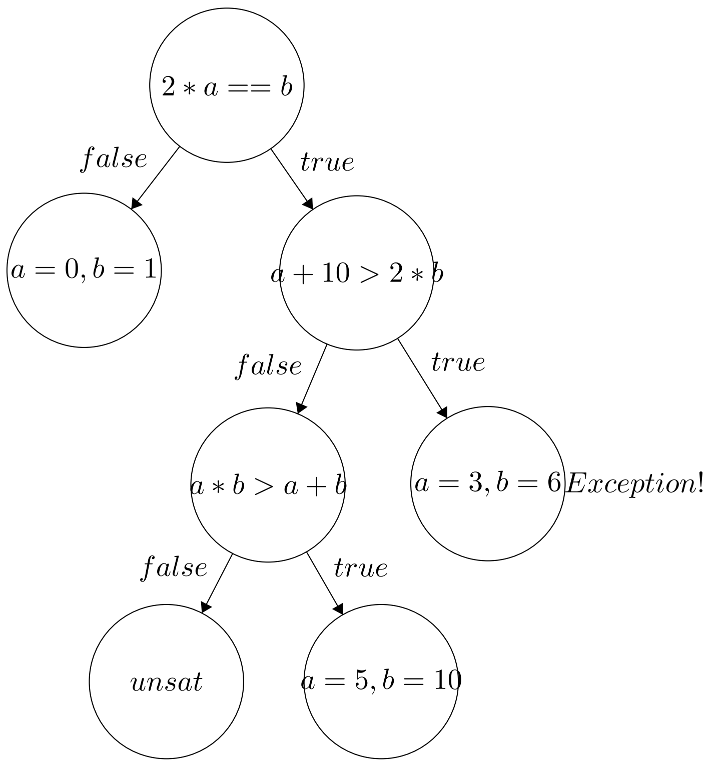

Figure A.3: Symbolic Execution Tree of code example of Figure A.2

A.3 R-indistinguishability of Data Languages

Definition 2 (**Data Languages**)**.** *Any two data words w and w' are $\mathcal{R}$-indistinguishable if they have the exact same sequence of values that can not be distinguished by any of the relations in $\mathcal{R}$. Then, a data language $\mathcal{L}$ is a set of data words that hold the following property:*

$$\textbf{\textit{if}}\ w \text{ and } w' \text{ are } \mathcal{R}\text{-indistinguishable } \textbf{\textit{then}}\ w \in \mathcal{L} \iff w' \in \mathcal{L}.$$

A.4 Overhead experiment

In order to ensure the tainting implementation does not provide too much overhead, we ran some experiments to experiment on this property. We ran a `for` loop of one million iterations on a `java.util.HashSet<Integer>` with four operations, two `add`'s and two `remove`'s. Both the additions and removes of on the set impose relational comparisons captured on the tainted objects. The results proved that the additional operations done when the operators are overloaded and the extensions of the Integer class do not impose a great performance decrease when these tainted Integer objects are short-lived. In our setting, the live span of such an object only spans the trace of a membership query. Naturally, the more comparisons you would add to the benchmark, the worse the distinction between the tainted and ordinary Integer becomes. In this case, were there are two comparisons done on the tainted Integers per iteration which translates to two million comparisons, the slowdown is 16%.

Object	Execution time (avg. seconds)	Standard deviation
`Integer`	$8,06117 \cdot 10^{-2}$	$4,2763 \cdot 10^{-3}$
`TaintInteger`	$9,32241 \cdot 10^{-2}$	$4,1684 \cdot 10^{-3}$

Table A.1: Benchmark of the tainted and ordinary Integer.