AUTOMATIC MODEL LEARNING AND ITS APPLICATIONS IN MALWARE DETECTION

HAO XIAO

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

August 2017
AUTOMATIC MODEL LEARNING AND ITS APPLICATIONS IN MALWARE DETECTION

HAO XIAO

School of Computer Science and Engineering

A thesis submitted to the Nanyang Technological University in partial fulfillment of the requirement for the degree of Doctor of Philosophy

August 2017
Acknowledgements

First and foremost, I am very lucky and grateful for having two supervisors: Dr. Yang LIU and Dr. Jun SUN. Both of them are always supportive and responsible towards my research and personal life during the whole course of my doctoral study and years before the doctoral study when I was working as a research assistant with them. They gave me great freedom to explore research topics that I considered important and interesting. They are always available for discussions and they showed me how critical thinking can improve my initial research ideas. Their styles of writing and revising paper showed me how great research ideas and results should be conveyed to a larger research community. Their love for their families and hobbies showed me a great example how work and life can be balanced.

I sincerely thank my master thesis supervisor Dr. Geguang PU and my academic collaborators: Dr. Ting SU, Dr. Shang-Wei LING, Dr. Sanjeev Kumar DAS, Dr. Shuang LIU, Dr. Yinxing XUE, Dr. Chengnian SUN, and Dr. Shengchao QIN. I really learned a lot from your insightful discussions and helpful comments on my research work. Without your help the research work in this thesis would not have been possible.

It has been a good experience to work with members in the Cyber Security Lab at NTU and members in the System Analysis and Verification Group at SUTD. Special appreciation of my fellow students and friends: Guozhu MENG, Dongxia WANG, and Zhimin WU. Thank you for the help on various course projects and discussions on non-academic issues. Without your friendship and company my Ph.D. journey would not be easy and fun.

I am deeply indebted to my family for their unconditional love and support. They are always fully supportive for the years long process of my Ph.D. study. Especial thanks to my wife for all the love and encouragements during downturns of the Ph.D. journey and for the remaining years to come.
# Contents

Contents ........................................... ii
List of Figures .................................... vi
List of Tables ...................................... viii
Abstract ........................................... ix

1 Introduction ...................................... 1
   1.1 Motivations ................................... 1
   1.2 Objectives .................................... 3
   1.3 Challenges ................................... 4
   1.4 Contributions ................................ 4
   1.5 Outline ...................................... 7
   1.6 Publications ................................ 7

2 Background and Preliminaries .............. 10
   2.1 Behavior Models .............................. 10
      2.1.1 Finite State Machines and Behavior Models ....... 10
   2.2 Inference Techniques for Behavior Models .......... 12
      2.2.1 Active Inference Techniques .................. 12
         2.2.1.1 The $L^*$ Algorithm ..................... 13
         2.2.1.2 Symbolic Execution ...................... 19
      2.2.2 Passive Inference Techniques ................ 22
      2.2.3 Invariant Generation Techniques .............. 24
         2.2.3.1 Support Vector Machine ................. 24

3 Learning Stateful Models ..................... 27
   3.1 Introduction .................................. 27
   3.2 Approach Overview ........................... 30
   3.3 Definitions .................................. 36
   3.4 The Tester .................................... 37
   3.5 The Refiner .................................. 40
   3.6 The Learner .................................. 45
      3.6.1 $L^*$ with Lazy Alphabet Refinement .......... 45
   3.7 Implementation ................................ 48
### Contents

3.8 Evaluation ................................. 50
   3.8.1 Results ................................... 51
   3.8.2 Comparison with Related Tools ............... 52
   3.8.3 Limitations and Discussions ................. 54
3.9 Summary .................................. 55

4 Learning Sound and Complete Stateful Models .................. 57
   4.1 Introduction ................................ 58
   4.2 Approach Overview ............................ 60
   4.3 Problem Definition ............................ 65
   4.4 Testing and Learning ......................... 66
   4.5 Validation .................................. 70
   4.6 Abstraction Refinement ....................... 74
   4.7 Evaluation ................................. 77
      4.7.1 Performance of TLV ...................... 77
         4.7.1.1 Comparison with Related Tools ....... 80
      4.7.2 Effectiveness of Testing and Learning .... 82
      4.7.3 Capability of Predicates Generation .... 84
      4.7.4 Limitations and Discussions ............. 85
   4.8 Summary .................................. 85

5 Learning Models via Passive and Active Learning ............... 88
   5.1 Introduction ................................ 88
   5.2 Approach Overview ............................ 89
   5.3 Call Sequences Extraction ..................... 95
   5.4 Passive Learning ............................. 99
      5.4.1 Hidden Markov Models .................... 99
      5.4.2 Training HMMs ........................... 100
         5.4.2.1 Convergence Criteria ................. 101
         5.4.2.2 Initial Model ........................ 101
         5.4.2.3 Optimal HMMs ......................... 101
      5.4.3 Validity Thresholds ........................ 102
   5.5 Active Learning .............................. 103
      5.5.1 Membership Query ......................... 103
      5.5.2 Candidate Query .......................... 103
      5.5.3 Random Testing ........................... 104
   5.6 Evaluation ................................. 105
      5.6.1 Settings ................................ 105
      5.6.2 Performance of HiSTAR .................... 107
         5.6.2.1 Performance of Call Sequences Extraction 107
         5.6.2.2 Performance of HMMs Training ........ 108
         5.6.2.3 Performance of DFAs Learning ........ 108
      5.6.3 Quality of Generated Models .............. 109
      5.6.4 Comparison with Related Tools ............ 111
5.6.5 Threat to Validity and Discussions .................. 112
5.7 Summary .................................................. 112

6 Learning Models for Malicious JavaScript Detection 115
6.1 Introduction .............................................. 115
6.2 Attack Behavior Modelling ............................... 119
6.3 Approach Overview ...................................... 121
6.4 Trace Preprocessing .................................... 123
6.5 JS* Learning Framework ................................. 125
   6.5.1 Membership Query ................................. 125
      6.5.1.1 Browser Defense Rule ...................... 126
      6.5.1.2 Data Dependency Analysis ............... 126
      6.5.1.3 Trace Replay ............................... 128
      6.5.1.4 Membership Query Algorithm ............ 129
   6.5.2 Candidate Query .................................. 131
   6.5.3 The Learned DFA and Refinement ................ 133
6.6 Implementation ......................................... 134
   6.6.1 Capturing System Calls ........................... 134
   6.6.2 Trace Replay ..................................... 135
      6.6.2.1 Running JEIS with Arguments ............ 136
      6.6.2.2 Execution Results of JEIS ............... 137
6.7 Evaluation .............................................. 138
   6.7.1 Settings .......................................... 138
   6.7.2 Attack Models Learning ......................... 139
   6.7.3 Discussion ....................................... 146
   6.7.4 Threats to Validity ............................... 147
6.8 Summary ................................................ 147

7 Learning Models for Hardware-assisted Malware Defense 150
7.1 Introduction ............................................ 150
7.2 Approach Overview .................................... 152
7.3 Attack Behavior Models ................................. 153
7.4 Behavior Models Learning .............................. 154
   7.4.1 Membership Query ............................... 155
   7.4.2 Candidate Query ................................ 155
7.5 Microarchitecture Design .............................. 156
   7.5.1 System Call Bin (sysbin) ....................... 157
   7.5.2 Buffer ............................................ 157
   7.5.3 DFA Checker ...................................... 158
7.6 Evaluation .............................................. 158
   7.6.1 Evaluation of Learning ......................... 158
   7.6.2 Evaluation of Detection Capability .......... 159
      7.6.2.1 Results ..................................... 160
   7.6.3 Threats to Validity ............................... 161
Contents

7.7 Summary ................................................................. 161

8 Conclusions .............................................................. 163
  8.1 Thesis Summary ...................................................... 163
  8.2 Future Works ......................................................... 164

Bibliography ................................................................. 166
List of Figures

1.1 The road map of this thesis ........................................ 7
2.1 The DFA accepts strings with an even number of 0’s and an even number 1’s .................................................. 11
2.2 The conceptual framework of $L^*$ algorithm .................... 13
2.3 The first observation table and candidate DFA .................... 17
2.4 The second closed and consistent observation table and its corresponding DFA .................................................... 18
2.5 The third closed and consistent observation table and its corresponding DFA .................................................... 18
2.6 Example program for symbolic execution ........................... 20
2.7 The symbolic tree ..................................................... 21
2.8 A linearly separable two-dimensional binary classification SVM 25
3.1 The framework of our approach ...................................... 29
3.2 The first observation table and candidate DFA .................... 30
3.3 The second closed and consistent observation table and its corresponding DFA .................................................... 31
3.4 The third closed and consistent observation table and its corresponding DFA .................................................... 32
3.5 (a) The first closed and consistent observation table and (b) its corresponding candidate stateful model ....................... 33
3.6 (a) The second closed and consistent observation table and (b) its corresponding candidate stateful behavior model ....................... 34
3.7 The numerical type graph for class java.util.Stack ............... 42
3.8 The numerical value graphs for java.util.Stack .................. 43
3.9 The observation table generated by the lazy $L^*$ algorithm ....... 47
3.10 Performance statistics for mining stateful behavior models ....... 53
4.1 The architecture of TLV ............................................ 58
4.2 A bounded stack in Java ............................................. 60
4.3 Artifacts in the 1st learning and validation iteration ............... 61
4.4 Modified push method .............................................. 63
4.5 The refined abstraction, where $p_1$ stands for size $\geq 0$, $p_2$ stands for size $= 0$ and $p_3$ stands for size $> 0$ .................. 64
4.6 The instrumentation of original code for symbolic proof ........... 72
4.7 A behavior model for java.net.Socket ............................. 83
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>The workflow of HiSTAR</td>
<td>89</td>
</tr>
<tr>
<td>5.2</td>
<td>A client program which uses the JDK class <code>java.util.Stack</code> (a) and its control flow graph (b)</td>
<td>90</td>
</tr>
<tr>
<td>5.3</td>
<td>A simplified version of the learned HMM</td>
<td>91</td>
</tr>
<tr>
<td>5.4</td>
<td>The initial table (a) and the first closed table (b) and its DFA (c)</td>
<td>92</td>
</tr>
<tr>
<td>5.5</td>
<td>The second closed table (a) and its DFA (b)</td>
<td>92</td>
</tr>
<tr>
<td>5.6</td>
<td>The learned DFA for class <code>java.util.Stack</code></td>
<td>94</td>
</tr>
<tr>
<td>5.7</td>
<td>The 3 biased coin tossing example for illustrating HMMs</td>
<td>100</td>
</tr>
<tr>
<td>5.8</td>
<td>The precision and recall plots of 30 DFAs</td>
<td>110</td>
</tr>
<tr>
<td>6.1</td>
<td>The JS code of a web-based attack</td>
<td>120</td>
</tr>
<tr>
<td>6.2</td>
<td>A concrete execution trace and the common actions <code>{a, b, c, d, e, f}</code> in the malicious traces of the attack in Figure 6.1</td>
<td>120</td>
</tr>
<tr>
<td>6.3</td>
<td>The general workflow of our approach JS* to modelling the behaviors of an attack type</td>
<td>122</td>
</tr>
<tr>
<td>6.4</td>
<td>Representing traces as sequences of common actions (π_m8 is the counterpart of the trace in Figure 6.2)</td>
<td>125</td>
</tr>
<tr>
<td>6.5</td>
<td>The JEIS statements reverse-engineered from action sequence π_m1: <code>{c,a,b,d,e,f}</code></td>
<td>128</td>
</tr>
<tr>
<td>6.6</td>
<td>The learned candidate DFA C_1 (a) and C_2 (b)</td>
<td>132</td>
</tr>
<tr>
<td>6.7</td>
<td>Firefox JS execution environment</td>
<td>134</td>
</tr>
<tr>
<td>6.8</td>
<td>The JEIS for the malicious action sequence <code>{a,b,d,e,f}</code></td>
<td>135</td>
</tr>
<tr>
<td>6.9</td>
<td>The JEIS for the benign action sequence <code>{a,b,c,d,f,e}</code></td>
<td>137</td>
</tr>
<tr>
<td>6.10</td>
<td>The DFA of binding shell using TCP, which serves a partial DFA of all attack types</td>
<td>140</td>
</tr>
<tr>
<td>6.11</td>
<td>The partial DFA of Type I attack, which models the exploit to CVE-2013-1710</td>
<td>141</td>
</tr>
<tr>
<td>7.1</td>
<td>The workflow of our approach</td>
<td>151</td>
</tr>
<tr>
<td>7.2</td>
<td>The block diagram of Malguard</td>
<td>157</td>
</tr>
</tbody>
</table>
List of Tables

3.1 The runtime statistics for TzuYu ........................................ 51
3.2 Program invariants generated by Daikon, Psyco and TzuYu ........ 52

4.1 Statistics on TLV abstracting the classes, where N.A. stands for not available ......................................................... 78
4.2 Comparing TLV with TAUTOKO on abstraction ......................... 81
4.3 Comparing TLV’s abstraction and manual abstraction ................. 83
4.4 Statistics for automatic predicate generation ........................... 84

5.1 Statistics of artifacts used and generated by HiSTAR ............... 106

6.1 Statistics about the learning framework ................................. 139
6.2 The predication results of JS*, JSAND and 56 tools provided by VIRUS-TOTAL ...................................................... 142
6.3 Detection ratio of our approach and other tools on 156 malicious samples in the testing data set .......................................... 144

7.1 Detection performances of Malguard with 50% (middle part) and 30% (right part) of samples as the training dataset ............. 159
7.2 Resource utilization of DFA checker ..................................... 160
Abstract

A behavior model of a program captures the correct ways of invoking its Application Programming Interfaces (APIs). For instance, one way for a Java programmer to read a text file is to open the file and then read the contents of the file and finally close the file after reading. Automatic learning of behavior models of programs can benefit many applications such as software verification by generating the models of the target program for verification, software testing by generating models of standard libraries for test generation, security analysis by generating the attack models of malicious software (malware) for malware detection, and software maintenance by generating models of legacy programs for code comprehension.

The first part of this thesis is dedicated to automatic and efficient techniques for learning accurate behavior models of program libraries. In our first work, we developed a fully automatic approach to learning more expressive behavior models more efficiently. The learned model can capture behaviors of a single class in object-oriented programs. Existing approaches for learning such models are often not efficient due to the use of model checking or symbolic execution. In this approach, testing and active learning are used to efficiently learn behavior models. A machine learning technique is used to efficiently synthesize the Boolean conditions for invoking an API. To solve the low coverage problem of testing of the first approach without compromising too much on efficiency, the second approach uses symbolic execution to verify and refine the behavior models which are actively learned through testing. Another contribution of the second approach is that learned models are not only precise and accurate but also give users the option to specify the appropriate level of abstraction. The second approach inherits limitations from symbolic execution, which lacks capability in handling commonly used program features such as heap data structures and program loops.
To circumvent the problems in the first and second approaches, the third approach harvests the experiences of existing example programs which use the program library to learn behavior models. We adopt a statistical machine learning technique to learn behavior models from API usages in example programs of the library to generate human interpretable behavior models.

The second part of the thesis is dedicated to the application of the proposed approaches for malware detection in order to demonstrate the usefulness of automatic model learning. The attack behavior of a malware often consists of several sub-tasks in sequence to achieve malicious intents and thus the attack behavior of malwares can be modelled with deterministic finite automata. Thus we apply the proposed techniques to automatically learn the attack behavior models of malwares. The first application is for the detection of malicious JavaScript programs embedded in Web pages. We extend the active learning algorithm with dynamic analyses to learn behavior models which capture the attack pattern of the malware. Then the learned attack behavior models are used to detect malware variants. The detection part in the first application is implemented in software which can be easily bypassed or disabled by malicious JavaScript programs. The second application is for the hardware-assisted detection of malicious desktop applications. In this application, we use static analyses with active learning to learn attack behavior models of malwares and then encode learned behavior models of known malicious programs in hardware to detect malicious applications. Compared with the first application, the hardware-assisted detection in the second application cannot be easily bypassed, if not impossible. The central idea behind these two applications is that the attack behavior of malicious software can be modelled as behavior models. Then the attack behavior models can be learned automatically and can be used to detect other malicious software.

In a nutshell, we developed three automatic approaches for efficient learning of behavior models of program libraries and demonstrated the strengths of the proposed approaches by applying them to learn attack behavior models of malwares for the detection of malware variants.
Chapter 1

Introduction

1.1 Motivations

The principle of building complex software out of existing ones is widely accepted by software vendors and software developers because it helps to contain the complexity of, to reduce the overall cost of, and to shorten time-to-market of developing software. As the functionality and complexity of modern software grow, most software is built with the support of existing components and libraries. To use existing libraries, developers need to invoke the Application Programming Interfaces (APIs) of these libraries.

Most libraries require programmers to invoke their APIs following a set of rules. These rules are often implicitly or explicitly documented in the accompanying API documentation and demonstrated with example programs. The API documentations and example programs make it easier for other programmers to understand the behavior of the library and thus the correct way of invoking its APIs. A program which invokes the APIs of a program library is called a client program of the library. In order to use or reuse the functionalities provided by the library correctly, programmers of client programs have to read the API documents and check the example programs to build a mental model on the correct way of invoking the APIs. This model captures the behavior of the APIs in the library and thus is called the library’s behavior model.
However, in practice, due to objective constraints such as hard deadlines and time-to-market pressure [47], or subjective constraints such as programmers’ reluctance to write documentation, most program libraries do not come with a correct and up-to-date documentation, not to mention example programs. Even for program libraries with correct and up-to-date documentations and running examples, compared with the documentation and example programs, the behavior model is an abstract model of the program’s behaviors and is easier for human understanding and machine automation. Due to the pervasiveness of libraries, automatically learning behavior models of libraries benefits many important applications. The following (partial) list highlights some representative applications of behavior models.

- **Program verification.** The software model checking approach [96] requires a model of the program under verification as input and the program often uses third-party libraries whose source code is not available. Thus precise behavior models of third-party libraries and even of the program under verification can reduce search space of model checking tools. Behavior models could benefit compositional model checking [57, 113, 114, 116] by learning the models or assumptions of each component [60, 126].

- **Bug finding and code recommendation.** The behavior models of a library can be used to statically check whether a client program uses the APIs of the library correctly and report bugs if incorrect use is detected [155]. The behavior models can also be used to recommend more accurate next API for programmer while programming [143].

- **Program testing.** Symbolic execution [81, 102] is a widely used white-box test generation technique [82, 180]. However symbolic execution has difficulties in handling third-party programs for which the source code is not available, or too complex to be handled even if the source code is available. Behavior models of the third-party program could enable more accurate and efficient symbolic execution [198]. To date, most of the models for standard program libraries (e.g., libraries in the .NET framework) used in model checking tools are constructed
manually [42]. Thus atomic learning of behavior models for standard program
library could benefit model checking [95].

- Security analysis. A malicious software (malware) often is characterized by its
attacking pattern and malwares in the same family often have common attack
patterns. The behavior model of a malware’s attacking pattern can be viewed as
a semantic model instead of a syntactic model and therefore the behavior model
could be used to detect and classification of malwares more accurately [55, 167].
The second part of this thesis also demonstrates the applicability of learning attack
behavior model for malware detection.

- Legacy program comprehension. Legacy programs are programs that are still in
use but their underlying technologies (such as programming language) or envi-
ronments are no longer supported or outdated [61]. In order to upgrade legacy
programs, understanding of legacy programs is the very important first step. Be-
havior models of legacy programs make the understanding of legacy programs
easier [131, 181].

1.2 Objectives

The objective of this thesis is automatic learning of accurate behavior models of pro-
grams efficiently. In the first part of the thesis, we aim to learn behavior models for
a single class in object-oriented programs. The behavior models capture legal call se-
quences of the interfaces (e.g., public methods and externally visible data fields) defined
in the class and may as well as exceptional call sequences whose execution results in
exceptions. Efficiency is always desirable and often an important but difficult objective
to achieve. Efficiency is also one necessary condition for the scalability of the proposed
approaches. Automation reduces the much manual effort involved in generating behav-
ior models and thus is another necessary condition for scalability. The second part of the
thesis demonstrates some useful applications of the learned behavior models in terms
of malware analysis, where we use the proposed approach to learn behavior models of
a malware’s attack behaviors and use the learned behavior models to detect malware variants.

1.3 Challenges

To achieve the three goals is a difficult task due to their competing nature with each other. Accuracy and efficiency is competing with each other. Although there are approaches which automatically learn behavior models for libraries, they are either inefficient or less accurate. Existing approaches which learn accurate behavior models with heavy weight formal methods such as model checking and symbolic execution are not efficient. These approaches are not efficient because the heavyweight techniques sacrifice efficiency for accuracy. There are also dynamic approaches which can learn behavior models for a class efficiently, but the learned models are either incorrect or inaccurate. These approaches are efficient because they use concrete traces which are generated with concrete execution. These approaches compromise correctness and accuracy for efficiency. Besides, as the complexity of modern software increases, the programs are becoming more and more large in terms of code size. The efficiency goal requires the new approaches to be more scalable.

1.4 Contributions

The first approach proposes to learn a variant of the deterministic finite automaton, which is called stateful behavior model, for a single class in object-oriented programs efficiently. The stateful behavior model captures both normal and exceptional sequences of method calls to public methods defined in the class. Stateful behavior model extends the classic deterministic finite automaton by associating a Boolean formula with the action in a transition. Thus the learned model can capture behaviors for most classes which access their data fields. Unlike existing approaches which use heavy-weight techniques such as model checking and symbolic execution to learn the behavior models,
our approach runs the target class by generating test cases automatically with the feedback directed random testing to learn its behavior model. Although symbolic execution can also be used to generate test cases with higher coverage, the feedback directed test generation technique used in this approach is much more efficient. Besides testing, the guard conditions in transitions are learned automatically with an efficient machine learning technique other than the template-based invariant generation techniques used in existing approaches. Template-based approaches need to apply each of the many templates to all runtime data whereas our machine learning based invariant generation technique needs to traverse all the runtime data only once to generate the invariants. Thus these two techniques in combination make our approach much more efficient than existing approaches.

Testing has its advantage of efficiency as well as its disadvantage of incompleteness in exercising all the behaviors of the target class. Thus, the benefit and consequence of using testing in the first approach are its efficiency in learning behavior models and the learned models are not always complete, respectively. In order to solve the completeness problem without compromising too much on efficiency, the second approach proposes to learn precise and accurate models for classes efficiently. Although it also uses symbolic execution to ensure that the learned models are refined to a complete model, the efficiency is not compromised too much for completeness due to the central idea of using testing as much as possible. Because concrete execution is much faster than symbolic execution, our approach is more efficient than existing approaches which rely on pure symbolic execution. Another contribution of the second approach is that learned models are not only precise and accurate but also it allows users to specify the appropriate level of abstraction. Although this approach is efficient compared with approaches based on symbolic execution, it still uses symbolic execution to prove and refine the behavior models generated by testing and thus suffers the fundamental limitations of symbolic execution and testing.

The third approach circumvents symbolic execution but leverages knowledge of existing programs to learn behavior models. The third approach first uses passive learning to learn an initial model from existing client programs which invoke the APIs of a class
and then uses active learning to learn a behavior model from the initial model and testing results. This approach does not suffer the problem of symbolic execution and often learns an accurate model for libraries with many client programs. This approach is efficient and scalable to learn accurate models for more than 800 classes due to the following reasons: the existence of large number of client programs enables the passive learning algorithm to generate high-quality models; most of the membership queries are answered through querying the passive models which are much faster than testing.

We applied the active learning approach to learn models which capture the attack pattern of malicious JavaScript (JS) programs. We propose to use behavior model of the browser-level APIs used by JavaScript malwares as a semantic model for the malware and use these models to detect JS malwares variants of the same attack type. The main contribution is that we combine several program analysis techniques to check whether a sequence of browser-level APIs is malicious or not. Because the behavior models are both semantic-related and compact enough, our approach achieves better detection results in a more efficient way.

The second application also extends active learning to learn the attack behavior models for desktop malwares. The attack behavior models capture the common attack behaviors of malware of the same attack type using the system calls. Instead of using dynamic analyses, this approach uses static analyses to derive rules from dynamic execution traces and uses these rules to check whether an execution trace is malicious or not. The learned models are then implemented in hardware so as to counter anti-detection techniques often used by malwares. Although the detection functionality is implemented in hardware, our approach achieves the same flexibility for deploying new behavior models to detect new attack types. Because we only encode the attack behavior models in hardware, the hardware takes less time to check and detect malicious traces.
Chapter 1. Introduction

The rest of the thesis is organized as the following: we review some techniques used in this thesis and survey related work in Chapter 2. The road map of our technical chapters is shown in Figure 1.1. We introduce our first efficient approach to learning incomplete models for a single class in Chapter 3. Then we introduce our second approach to learning accurate and correct models for a single class in Chapter 4. In Chapter 5, we introduce the third approach which learns the behavior models for a program library from its many client programs using a machine learning technique. Chapter 6 reports the first application of the proposed approaches to detect malicious JavaScript programs embedded in web pages. Chapter 7 reports the second application of our proposed approaches to learn behavior models of conventional malicious software and implement the learned behavior models in hardware to detect new malicious applications at runtime. We conclude this thesis and provide some future research topics in Chapter 8.

1.5 Outline

The rest of the thesis is organized as the following: we review some techniques used in this thesis and survey related work in Chapter 2. The road map of our technical chapters is shown in Figure 1.1. We introduce our first efficient approach to learning incomplete models for a single class in Chapter 3. Then we introduce our second approach to learning accurate and correct models for a single class in Chapter 4. In Chapter 5, we introduce the third approach which learns the behavior models for a program library from its many client programs using a machine learning technique. Chapter 6 reports the first application of the proposed approaches to detect malicious JavaScript programs embedded in web pages. Chapter 7 reports the second application of our proposed approaches to learn behavior models of conventional malicious software and implement the learned behavior models in hardware to detect new malicious applications at runtime. We conclude this thesis and provide some future research topics in Chapter 8.

1.6 Publications

The work described in Chapter 3 and Chapter 4 are published in ASE 2013 [191] and ESEC/FSE 2015 [177], respectively. The work described in Chapter 5 has been submitted to ICSE 2018. The work described in Chapter 6 is published in ISSTA 2015 [195]. I provided the active learning idea and framework and the design of the actual implementation of the techniques used to answer membership queries and candidate queries. The work described in Chapter 7 is published in ISCAS 2016 [69]. I contributed to the
attack behavior model learning part. The other semi-automatic approach published at ICECCS 2016 [119], which applies the proposed learning approach to use case generation with human interaction, is not reported in this thesis. All the publications of the candidate during his Ph.D. candidature are listed as the following and the publications which are not included in this thesis are marked with an asterisk [117, 119, 199].


Chapter 2

Background and Preliminaries

This chapter gives a brief introduction of related work in behavior models learning and several notations and techniques that are used in later chapters of this thesis.

2.1 Behavior Models

The main goal of our research is to discover the behavior models of programs. In this thesis, a behavior model of a program is basically a finite-state machine whose alphabet consists of the program’s APIs.

2.1.1 Finite State Machines and Behavior Models

A Finite State Machine (FSM) or Finite State Automaton (FSA) is a widely used mathematical model of computation. An FSM $M = (S, \Sigma, s_0, \delta, F)$ consists of a finite set of states $S$, an initial state $s_0 \in S$, a finite set of input symbols $\Sigma$ called the alphabet, a set of accepting states $F \subseteq S$, and a transition function $\delta$ which maps each pair of state and symbol to a finite set of state, i.e., $\delta : S \times \Sigma \rightarrow \mathcal{P}(S)$, where $\mathcal{P}(S)$ denotes the power set of a set $S$, i.e., the power set contains all subsets of $S$. At any moment $M$ is at one state which is called the current state (at the beginning $M$ is at the initial state $s_0$). It can transit to another state which is the destination state given an input symbol according to the transition function. According to the determinism of the transition function, an
**Chapter 2. Background and Preliminaries**

Figure 2.1: The DFA accepts strings with an even number of 0’s and an even number 1’s. The initial state is state $s_0$ with an incoming short arrow, accepting states are denoted as double circles and non-accepting state are denoted as circles.

An FSM could be a Deterministic Finite Automaton (DFA) if for a given pair of state and input symbol there is only one destination state to transit to (i.e., $\delta: S \times \Sigma \rightarrow S$), or a Non-deterministic Finite Automaton (NFA) if for a given pair of state and input symbol there are possibly multiple destination states to transit to (i.e., $\delta: S \times \Sigma \rightarrow P(S)$).

In this thesis, both DFAs and NFAs are used to capture legal call sequences of APIs of a class. A string $str = \langle e_0, e_1, \cdots, e_n \rangle \in \Sigma^*$, which is a finite sequence of symbols, is *accepted* by an FSM $M$ if there is a sequence of states $s = \langle s_0, s_1, \cdots, s_{n+1} \rangle$ such that $s_{i+1} \in \delta(s_i, e_i)$ for all $0 \leq i \leq n$ and $s_{n+1} \in F$. The last state $s_{n+1}$ of the string $str$ in $M$ is called the *final state*. i.e., a string is accepted by a machine iff its final state on the machine is an accepting state. The extended notation $\delta(s, str)$ denotes the state reached by starting from the state $s$ and traversing the string $str$. A *language* is a (possibly infinite) set of strings. A language is said to be *regular* if there is an FSM which accepts all its strings. The set of all accepted strings of a machine $M$ is called the *language of $M$*, denoted as $L(M)$. By definition, all languages of FSMs are regular.

**Example 2.1.** The DFA shown in Figure 2.1 accepts the regular language with an even number of 0’s and an even number of 1’s (the numbers of 0’s and 1’s are not required to be equal). The formal definition of this DFA is the following: $S = \{s_0, s_1, s_2, s_3\}$, $\Sigma = \{0, 1\}$, initial state is $s_0$, $F = \{s_0\}$ and $\delta(s_0, 0) = s_1$, $\delta(s_0, 1) = s_2$, $\delta(s_1, 0) = s_0$, $\delta(s_1, 1) = s_3$, $\delta(s_2, 0) = s_3$, $\delta(s_1, 1) = s_0$, $\delta(s_3, 0) = s_2$, $\delta(s_3, 1) = s_1$. The string $\langle 0, 1, 0, 1, 0, 0 \rangle$ is accepted by this DFA and its sequence of traversed states is $\langle s_0, s_1, s_3, s_2, s_0, s_1, s_0 \rangle$ because its final state $s_0$ (underlined) is an accepting state.
2.2 Inference Techniques for Behavior Models

The behavior models can be viewed as a special form of specification. Specification mining [23] refers to the process of mining specifications from various software artifacts such as source code [16], API documentations [148, 200], execution traces [35, 145, 153]. Thus, our research can be categorized as a special case of specification mining. There have been many research work on specification mining due to its important practical implications. The book by Lo et al. [122] presents an extensive survey of various specification mining approaches. Robillard et al. [164] also gives a more recent comprehensive survey on API properties inference techniques.

The inference techniques used for specification mining can be generally categorized into active inference techniques and passive inference techniques. Most of the algorithms in the literature are passive algorithms, which in essence generalize from a set of given inputs (e.g., execution traces and logs) to synthesize a target model. The active algorithms in contrast actively ask the artifact to generate a set of desired inputs from which they infer the target model.

2.2.1 Active Inference Techniques

The interface specification [22] is a special form of behavior model in which the transitions (labeled with methods) are augmented with Boolean formula. It captures the specification for how the methods defined in a class (the programming language abstraction, e.g., the Java class) should be invoked by client programs.

Alur et al. [22] use active learning algorithm $L^*$ and model checking to generate the interface specification for a Java class. The generated interface specification contains the valid call sequences of public methods defined in the class as well as exceptional call sequences, whose executions throw exceptions. Both membership queries and candidate queries are answered with model checking.

Giannakopoulou et al. [80, 91] propose symbolic execution based approaches to learn the interface specification. In their first paper [80], both membership queries and
candidate queries are answered with traditional symbolic execution. Their later approach [91] uses dynamic symbolic execution to generate all possible inputs for each public method first, then during learning phase, they use all combinations of those generated inputs to the methods in the call sequences to answer membership queries. Only when inconsistency discovered they use symbolic execution again to find those unexplored test inputs.

All above approaches use the \( L^* \) algorithm which is either extended or reused by all approaches in subsequent chapters of this thesis and therefore deserves a detailed introduction. Two of the above approaches also use symbolic execution together with the \( L^* \) algorithm. We will also use symbolic execution in Chapter 4 to generate accurate and complete behavior models in an efficient way and thus it also deserves a detailed introduction. In the following we first introduce the \( L^* \) algorithm which is immediately followed by the introduction of symbolic execution.

### 2.2.1.1 The \( L^* \) Algorithm

The \( L^* \) algorithm [24, 163] is an active learning algorithm for DFAs. The conceptual framework of the \( L^* \) algorithm is depicted in Figure 2.2. \( L^* \) makes three assumptions for learning DFAs: 1) it assumes that the system to be learned is modelled as a DFA \( D = (S, \Sigma, \text{init}, \delta, F) \); 2) it assumes the alphabet \( \Sigma \) of \( D \) is known and fixed; 3) it also assumes there is an oracle (a.k.a., teacher) who knows \( D \). It learns a DFA with the minimal number of states that accepts the same language of \( D \). During the process of learning, the \( L^* \) algorithm asks two kinds of queries for the teacher for getting information about \( D \): membership queries and candidate queries. A membership query asks
Chapter 2. Background and Preliminaries

whether a string \( str \) is an accepting string of \( D \), i.e., \( str \in \mathcal{L}(D) \); a candidate query asks whether a candidate DFA \( C \) accepts the same language of \( D \), i.e., \( \mathcal{L}(C) = \mathcal{L}(D) \).

During the learning process, the \( L^* \) algorithm stores results for the membership queries in an observation table \((P, E, T)\), where \( P \subseteq \Sigma^* \) is a prefix-closed set of finite strings (i.e., for each string \( str \in P \), all its prefixes are also in \( P \)) and \( P \) corresponds to the row indexes of the observation table; \( E \subseteq \Sigma^* \) is a suffix-closed set (i.e., for each string \( str \in E \), all its suffix strings are also in \( E \)) and \( E \) corresponds to the column indexes of the observation table; \( T \) is a mapping function such that \( T(str, str') = 1 \) if \( str \) is a string in \( P \) or a string in \( P \) appended with a symbol in \( \Sigma \), and \( str \cdot str' \) is a string accepted by \( D \), where \( str \cdot str' \) stands for the string obtained by concatenating string \( str \) and string \( str' \); otherwise, \( T(str, str') = 0 \). Intuitively, \( T \) maps the table cell indexed by row \( str \) and column \( str' \) to either 1 or 0 according to whether the string \( str \cdot str' \) is an accepting string of \( D \) or not.

In the actual implementation of the \( L^* \) algorithm, the rows of the observation table are separated into two categories and their corresponding row indexes are denoted by \( R \) and \( RA \) rows (i.e., \( P = R \cup RA \)), where for each row index \( str' \in RA \), there is a string \( str \in R \) and a symbol \( e \in \Sigma \) such that \( str \cdot e = str' \). In the observation table, the \( L^* \) algorithm categorizes strings based on the Myhill-Nerode Theorem [89].

Definition 2.1. (Myhill-Nerode Relation). A Myhill-Nerode Relation \( \equiv_D \) on \( \Sigma^* \) with respect to a DFA \( D = (S, \Sigma, init, \delta, F) \) with no inaccessible states, is an equivalence relation defined as \( str_1 \equiv_D str_2 \iff \delta(init, str_1) = \delta(init, str_2) \) for any two strings \( str_1, str_2 \in \Sigma^* \).

A Myhill-Nerode Relation is an equivalence relation i.e., it is reflexive, symmetric, and transitive by definition. This equivalence relation satisfies three additional properties:

(i) right congruence: for any string \( str_1, str_2 \in \Sigma^* \) and \( e \in \Sigma \), \( str_1 \equiv_D str_2 \Rightarrow str_1 \cdot e \equiv_D str_2 \cdot e \). Given the premise, we have \( \delta(init, str_1) = \delta(init, str_2) \), which means \( str_1 \) and \( str_2 \) lead to the same state, say \( s \). The transition \( \delta \) is deterministic and we have \( \delta(init, str_1 \cdot e) = \delta(s, e) = \delta(init, str_2 \cdot e) \), i.e., the consequent is proved.
(ii) It refines the language of \( D \): \( \text{str}_1 \equiv_D \text{str}_2 \Rightarrow \text{str}_1 \in \mathcal{L}(D) \iff \text{str}_2 \in \mathcal{L}(D) \). Suppose \( \delta(\text{init}, \text{str}_1) = \delta(\text{init}, \text{str}_2) \) lead to the same state \( s \), thus if \( s \) is an accepting state then both \( \text{str}_1 \) and \( \text{str}_2 \) are accepted by \( D \). From this and the previous property we can infer that \( \text{str}_1 \cdot \text{str}' \) is accepted by \( D \) iff \( \text{str}_2 \cdot \text{str}' \) is accepted by \( D \), for all \( \text{str}' \in \Sigma^* \).

(iii) There are only finite number of equivalence classes: There is exactly one equivalence class \( \{ \text{str} \in \Sigma^* | \delta(\text{init}, \text{str}) = s \} \) corresponding to each state \( s \) in \( D \). We denote the equivalence class for string \( \text{str}_1 \) as \( [\text{str}_1]_D \) as \( \{ \text{str}_2 | \text{str}_2 \equiv_D \text{str}_1 \} \) and \( \text{str}_1 \) is called a representing string for the equivalence class \( [\text{str}_1]_D \).

**Theorem 2.2. (Myhill-Nerode Theorem).** The language \( R \in \Sigma^* \) is regular (by definition, it is accepted by a DFA \( D \), i.e., \( R = \mathcal{L}(D) \)) iff there is a Myhill-Nerode relation for \( R \). \( \square \)

The above theorem states that a Myhill-Nerode relation is equivalent for a DFA. Thus the observation table used in the \( L^* \) algorithm is to find the Myhill-Nerode relation for the unknown DFA \( D \).

**Definition 2.3.** For a given observation table \( (P, E, T) \), the values for the row indexed by \( \text{str} \in P \) is denoted by \( \text{Row}(\text{str}) \) is a bit-vector \( B \) of the same length as that of \( E \), i.e., \( |B| = |E| \). The values in \( \text{Row}(\text{str}) \) is the concatenation of all cell values of \( T(\text{str}, \text{str}') \) where \( \text{str}' \in E \).

The purpose of \( L^* \) asking membership queries for the teacher is to make the observation table closed and consistent such that it can constructs a candidate DFA from the observation table. An observation table is said to be closed if for all \( \text{str} \in R \) and any \( e \in \Sigma \), there is a string \( \text{str}' \in P \) such that \( \text{Row}(\text{str} \cdot e) = \text{Row}(\text{str}') \). Note that \( \text{str}' \) could be indexed by \( RA \) as well as \( R \). An observation table is said to be consistent if for every pair of strings \( \text{str}_1, \text{str}_2 \in R \) such that \( \text{Row}(\text{str}_1) = \text{Row}(\text{str}_2) \), then \( \text{Row}(\text{str}_1 \cdot e) = \text{Row}(\text{str}_2 \cdot e) \) for all \( e \in \Sigma \). The \( L^* \) algorithm constructs a candidate DFA \( C = (S_c, \Sigma, \text{init}_c, \delta_c, F_c) \) from a closed and consistent observation table according to the Myhill-Nerode Theorem in the following way:
Chapter 2. Background and Preliminaries

- $S_c$ contains exactly one state for each different row indexed by a string in $R$. Note that one state standards for an equivalent class defined by the table. Equivalent rows in the observation table correspond to the same state because they belong to the same equivalence class;

- $init_c$ is the state corresponding to the row indexed by the empty row string $\langle \rangle$, i.e., $Row(\langle \rangle)$;

- for any state $s$ in $S_c$ which corresponds to a row indexed by string $str \in R$ and a symbol $e \in \Sigma$, $\delta_c(s, e) = s'$, where $s'$ is the state for row indexed by the string $str \cdot e$ in $P$;

- a state $s$ is in $F_c$ iff its corresponding row indexed by string $str \in R$ such that $T(str, \langle \rangle) = 1$.

After constructing the candidate DFA $C$, $L^*$ asks a candidate query for the teacher to check whether $L(C)$ is equivalent to $L(D)$.

If $C$ accepts the same language of $D$, then $L^*$ returns $C$ and terminates. Otherwise, the teacher returns a counterexample string $ce \in \Sigma^*$ such that one and only one of the two conditions $ce \in L(D) \land ce \notin L(C)$ and $ce \in L(C) \land ce \notin L(D)$ is true. Then $L^*$ analyzes $ce$ to find a witness suffix. A witness suffix is a string which, when appended to the two strings in the same equivalence class, provides enough evidence for the two strings to be classified into two equivalence classes under the Myhill-Nerode Relation. Let $str$ be the concatenation of two strings $str_0$ and $str_1$, i.e., $str_0 \cdot str_1 = str$. Let $s$ be the state reached from state $init$ via string $str_0$, i.e., $\delta(init, str_0) = s$. Thus $str_1$ is the witness suffix of $str$, denoted as $WS(str)$, if $\delta(s, str_1) = s'$ and $s' \neq \delta(init, str)$. Once the witness suffix $WS(ce)$ is obtained, $L^*$ uses $WS(ce)$ to refine the observation table (as well as the candidate DFA $C$) in an attempt to generate another candidate DFA $C'$ such that $L(C') = L(D)$.

It has been proved by Angluin [24] that as long as $L(D)$ is regular, $L^*$ can learn a minimum DFA which accepts the same language of $D$. $L^*$ at most asks $n - 1$ candidate queries and $O(|\Sigma| n^2 + n \log m)$ membership queries, where $m$ is the maximum length.
of the counterexample strings returned by the teacher and \( n \) is the number of states of the minimal DFA.

**Example 2.2.** Suppose the DFA \( D \) to be learned is the one shown in Figure 2.1. At the beginning, \( L^* \) fills the row indexes and column indexes with the empty string \( \langle \rangle \) and ask whether the empty string is accepted by \( D \). Thus at this moment, \( R = \{ \langle \rangle \} \), \( RA = \emptyset \), and \( E = \{ \langle \rangle \} \). The empty string is accepted by \( D \) and thus the value of cell indexed by \( (\langle \rangle, \langle \rangle) \) is 1.

Then for each symbol in \( \Sigma \) it asks a membership query and inserts the results into \( RA \) rows. In this case it asks one membership query for each of string \( \langle 0 \rangle = \langle 0 \rangle \cdot \langle \rangle \) and string \( \langle 1 \rangle = \langle 1 \rangle \cdot \langle \rangle \). The answers are both 0. The observation table is shown in Figure 2.3 (a), for which \( R = \{ \langle \rangle \} \), \( RA = \{ \langle 0 \rangle, \langle 1 \rangle \} \), and \( E = \{ \langle \rangle \} \). The table is not closed because the Row(\( \langle 0 \rangle \)) = 0 is not indexed by any string in \( R \). Thus the algorithm moves the row \( \langle 0 \rangle \) from \( RA \) to \( R \).

It then asks membership queries for string \( \langle 0, 1 \rangle = \langle 0, 1 \rangle \cdot \langle \rangle \) and string \( \langle 0, 0 \rangle = \langle 0, 0 \rangle \cdot \langle \rangle \) and adds the results to the RA. The table is shown in Figure 2.3 (b), for which \( R = \{ \langle \rangle, \langle 0 \rangle \} \), \( RA = \{ \langle 1 \rangle, \langle 0, 1 \rangle, \langle 0, 0 \rangle \} \), and \( E = \{ \langle \rangle \} \) and the table is closed and consistent.

\( L^* \) constructs the first candidate DFA, as shown in Figure 2.3 (c), from the table shown in Figure 2.3 (b) by following the following procedure: the two states labeled with 1 and 0 correspond to the two rows in the \( R \) rows, i.e., state 1 = \([\langle \rangle]\) and state 0 = \([\langle 0 \rangle]\). State 1 is the initial state because its row index is \( \langle \rangle \) and it is also an accepting state because its cell value indexed by the column index \( \langle \rangle \) is 1. The transitions labeled
0, 1 from state 1 goes to state 0 because the row indexed by rows index $\langle \rangle \cdot 0 = \langle 0 \rangle$ and $\langle \rangle \cdot 1 = \langle 1 \rangle$ are 0. The transition labeled 1 from state 0 goes to state 0 because the representing string for state 0 is $\langle 0 \rangle$ and the row indexed by row index $\langle 0,1 \rangle = \langle 0 \rangle \cdot 1$ is 0. The transition labeled 0 from state 0 goes to state 1 because the row indexed by row index $\langle 0,0 \rangle = \langle 0 \rangle \cdot 0$ is 1.

Next, $L^*$ asks a candidate query for the teacher to check whether the constructed DFA accepts the same language as $D$. The teacher answers “no” with a counterexample $\langle 1,1 \rangle$, which is accepted by $D$ but not accepted by the candidate DFA. For finding the witness suffix $WS(\langle 1,1 \rangle)$, since the longest prefix of the counterexample in $P$ is $\langle 1 \rangle$, thus $WS(\langle 1,1 \rangle)$ is $\langle 1 \rangle$, we add all its suffixes $\{\langle 1 \rangle, \langle \rangle\}$ into $E$, but the suffix $\langle \rangle$ is already in $E$, thus only $\langle 1 \rangle$ is added into $E$. Then $L^*$ asks membership queries to close the table. The intermediate tables are not shown to save space but the second closed and consistent
observation table is shown in Figure 2.4 (a). From this observation table, \( L^* \) constructs the second candidate DFA, as shown in Figure 2.4 (b), and asks another candidate query for the constructed DFA.

The teacher answers “no” again with another counterexample \( \langle 0, 1, 0 \rangle \) which is rejected by \( D \) but accepted by the candidate DFA. The witness suffix \( \langle 1, 0 \rangle \) is added into the set of suffixes \( E \) of the observation table, and the closed and consistent observation table is shown in Figure 2.5 (a). From this observation table, \( L^* \) constructs the third candidate DFA, as shown in Figure 2.5 (b), and asks a candidate query for the third DFA. This DFA accepts the same language as the one shown in Figure 2.1, thus \( L^* \) successfully learns the unknown DFA and terminates.

2.2.1.2 Symbolic Execution

Symbolic execution refers to the process of executing a program with symbolic input values instead of the normal execution (i.e., concrete execution) of the program which takes concrete input values. During execution of a program, symbolic execution engine maintains and updates the local variables with symbolic expressions over the symbolic input values according to the symbolic semantics of program constructs. It also maintains a path condition which records the branches it took until now. When it reaches a branching statement (e.g., “if” statement), it has to try both two possible values that the symbolic path condition is true and false because the result of the condition is symbolic and unknown. It adds the assumption to the path condition and follows each path independently. When it reaches the exit statement (e.g., “return” statement), symbolic execution engine stores in the symbolic path tree the path condition for which any concrete assignment to the symbolic input variables satisfying the path condition, then concrete execution of the program with these concrete inputs will follow the same path as the symbolic execution did. Thus a path condition collected by symbolic execution engine represents several concrete inputs that traverse the same program path. When it arrives at the exit statement, it negates one of the conditions in its path condition and follows the other branch of the corresponding branching statement, which may lead to discover
Example 2.3. We use the example shown in Figure 2.6, which is also used by Cadar and Sen [43], to illustrate how the classic symbolic execution works. The program starts execution from line 16, where `sym_input()` tells the symbolic execution engine that the variable on the left should be treated as a symbolic variable. Symbolic execution assigns a symbolic value to it (say $\alpha_0$ for $x$); Assume in line 17, $y$ is assigned with a symbolic value $\alpha_1$. Function `testme()` is the target function for symbolic execution. Given that all its parameters are assigned with symbolic values. The symbolic path tree of `testme()` is shown in Figure 2.7. The rounded rectangles denote symbolic states and the ellipses denote the branching conditions.

After line 5 (and before line 6), $x = \alpha_0$, $y = \alpha_1$ and $z = \bot$, i.e., the value of $z$ is undefined (see the symbolic state $n_5$). The symbolic semantic of assignment statement is that, if any right-hand-side (rhs) variable is symbolic value, the left-hand-side (lhs)
variable of the assignment statement is also symbolic, pertaining to its normal semantic by substituting symbolic variable in the rhs expression with its corresponding symbolic value. Thus line 2 returns symbolic value $2 \times \alpha_1$. The function call to function twice() in line 6 assigns $2 \times \alpha_1$ to $z$, thus the symbolic state after line 6 is depicted in $n_6$.

Line 7 is a branching statement whose condition has two possible values $2 \times \alpha_1 = \alpha_0$ or $2 \times \alpha_1 \neq \alpha_0$. Symbolic execution engine invokes an SMT solver to decide whether the two conditions are possible. In this case both conditions are satisfiable, thus symbolic execution engine “forks” its execution with two path conditions ($2 \times \alpha_1 = \alpha_0$ and $2 \times \alpha_1 \neq \alpha_0$ correspond to the right and left branch of $n_7$, respectively). Assume that symbolic execution engine explores the left branch first which continues to line 12 (its symbolic state is depicted in $n_{12}$ and returns. It continues to explore the right branch and comes to line 8, which is another branching statement. Then symbolic execution engine forks two branches (the left and right branch of node $n_8$). Assume symbolic execution explores the left branch first, in which case it returns. Then it starts to explore
the right branch and reaches line 9, which is the target error statement.

In order to check whether the ERROR statement is reachable, symbolic execution invokes an SMT solver to decide the satisfiability of current path condition $2 \ast \alpha_1 = \alpha_0 \land \alpha_0 > \alpha_1 + 7$. The SMT solver could return a concrete assignment for $\alpha_0 = 16, \alpha_1 = 8$. Under the concrete assignment for $x$ and $y$, `testme()` will reach the ERROR statement. By systematically traversing (e.g., in a similar way as depth-first search on the tree) the paths in a program, symbolic execution can be used for test case generation and program verification.

### 2.2.2 Passive Inference Techniques

The paper by Robillard et al. [164] gives a comprehensive survey of passive inference techniques for behavior models or finite state automata (FSA) for the last decade. There are also other older inference techniques [71, 88, 151] based on heuristics such as Regular Positive and Negative Inference (RPNI) [146] which infer finite state automata from both positive samples and negative samples. There are also passive inference algorithms such as k-tails algorithm [36] which infers finite state automata from positive samples only. The sk-strings algorithm [159] extends k-tails algorithm for learning probabilistic Finite State Automata (PFSA) passively from a given set of traces. ADABU [66] can be classified as a passive inference algorithm which requires a set of test cases as input; it abstracts the concrete states with simple templates to abstract states thus to get the abstract traces and then it merges models from abstract traces to generate a model.

Dallmeier et al. [65] generate the stateful behavior models for a Java class. The generated behavior model resembles the interface specification. The inputs to their approach are the target class and a set of test cases for the classes. They instrument the target class and execute the test cases on the instrumented class to generate program traces. From the traces they generate invariants based on a set of invariant templates. Then they merge different concrete traces to get an abstract trace for which the concrete traces have the same sequence of methods calls and the intermediate object states satisfy the same set of invariants.
Later, Dallmeier et al. [64] extend their previous work [65] with model based testing techniques to generate more test cases from the initial model which is generated with their previous approach [65]. From the initial model, they try to generate more test cases in the attempt to complete the model. If a state does not have an outgoing transition corresponding to a method, they generate the sequence of method calls by extending the existing trace which leads from the initial state to that state and appending the uncovered method. Then it executes the generated test cases to generate new concrete traces. After it generates all the sequences then it merges the generated concrete traces to the initial model to get a more complete model.

There are also work which try to generalize FSA for multiple interacting objects. Lee et al. [110] propose to learn the FSA for multiple interacting objects by consider the data dependency relation between different APIs of different classes. It first uses dynamic slicing to separate the traces into a set of sub-traces. The APIs in each sub-trace is data-dependent and each sub-trace is data-independent from other sub-traces. It then uses sk-strings algorithm [159] to learn a PFSA from all sub-traces. To alleviate the over-generalization problem in the standard sk-strings algorithm, it refines the FSA by expanding the states of more than one incoming transitions from other states and filtering out infeasible transitions according to the runtime traces and merging states that have the same outgoing transitions.

Lo et al. [124] propose to learn statistically significant temporal formula from traces and then use these formula to guide the behavioral model inference algorithm to infer precise FSA. The temporal rules are inferred from traces with a support greater than a threshold value and 100% confidence. The inferred rules are used as a state merging criteria in the state merging phase of the k-tails [36] algorithm. They use a stand model checking algorithm of Linear Temporal Logic (LTL) formula [26, 59] and propose a sound but incomplete LTL model checking algorithm to check whether FSA after merging satisfy the LTL formula. Thus the final FSA is precise than those generated with other state merging criteria.

Le et al. [109] propose to synergize existing FSA-based specification mining algorithms to generate better FSA through model fissions and fusions. In the model fission step, they specify 6 LTL template formula between two events and enumerate all
possible instantiations of the templates for each pair of events saw in the traces. The instantiated LTL formula are subject to model checking against each FSA generated by existing FSA-based specification mining algorithms such as k-tails [36], CONTRAC-TOR++ [105], SEKT [105] and TEMI [105]. For each FSA, the model fission step generates one set of LTL formula which are satisfied by the FSA. In the model fusion step, a set of heuristics such as union, intersection, majority and minimum count is used to select LTL formula from all sets of LTL formula. Then Each of the selected LTL formula is transformed to a FSA with two states. The final FSA is the intersection of the individual two-state FSA, i.e., it accepts a string which is accepted by all the two-state FSA.

2.2.3 Invariant Generation Techniques

The interface specification is a DFA augmented with predicates in either transitions or states. Different approaches need some way of discovering those predicates. Most dynamic approaches employ the templates based approaches to generate predicates from concrete traces by trying every template for all program states [75,179]. Most static approaches generate the predicates from the program source code by considering possible combinations of the branching conditions such as the conditions in the if and while statements. The model checking research community proposes different ways to generate predicates such as interpolants based approaches, which uses an SMT solver to infer predicates for unsatisfying program paths such that the inferred predicates witness why the path cannot be satisfied. In particular, Aiken et al. [170] first propose to use machine learning techniques to learn invariants. In the following, we introduce Support Vector Machines which are used to generate invariants in this thesis.

2.2.3.1 Support Vector Machine

The Support Vector Machine (SVM) is a supervised machine learning model used for classification and regression. It takes both positively and negatively labeled data points as input and generates the output as a hyperplane which separates the positively labeled data points from the negatively labeled data points if they are separable. Each input
data point could be a $p$-dimensional data with a label, denoted as $d_i(a_1, a_2, ..., a_p, L)$, where $L$ is the class label which could be one of $+\text{ and } -$ for a binary-class dataset and data points in the positive set are labeled with $+$ and data points in the negative sets are labeled with $-$.

In this thesis, we are only interested in the cases where the positive data points and the negative data points are linearly separable, for which there is a $(p−1)$-dimensional hyperplane that separates the positive and negative data points perfectly. In its simplest form, SVMs for binary-class linearly separable inputs. It first finds the data points in the positive (and negative) set that are closest to data points in the negative (positive) set. Those data points are called support vectors of the positive and negative data points, respectively. Then it generates the maximum margin $(p−1)$-dimensional hyperplane perpendicular to the lines connecting the support vectors in the positive and negative set. The support vectors are points lying in the rim of convex hulls of the positive and negative data points and are closest to each other. The generated hyperplanes are of the form $\Sigma_{i=1}^{p} c_i * x_i = c$, where $c$ and $c_i$ are constant real numbers and $x_i$ is a real typed variable denoting the $i$-th dimension, such that for each positive data point $d_i(a_1, a_2, ..., a_p, +)$ we have $\Sigma_{i=1}^{p} c_i * a_i > c$ and for each negative data point $d_i(a_1, a_2, ..., a_p, −)$ we have $\Sigma_{i=1}^{p} c_i * a_i < c$. 

\textbf{Figure 2.8:} The diagram of a 2-dimensional linear SVM. The cyan polygon and blue polygon is the convex hull of the positive data and negative data, respectively. It shows 3 hyperplanes and the maximum margin hyperplane is the one in purple.
Example 2.4. We assume the dimension of the inputs is 2 such that we can show the inputs and outputs in a two-dimension coordinate. The positive data points and negative data points are shown as + and −, respectively, in Figure 2.8. The support vectors in the positive set are data point (2, 3, +) and (3, 4, +); the support vectors in the negative set are the data point (3, 2, −) and (4, 3, −). Thus the maximum margin hyperplane for this example is the hyperplane $x_2 - x_1 = 0$. 
Chapter 3

Learning Stateful Models

Classic behavior models in the form of DFAs for a program library mainly focus on the finite control part of the library and lack expressiveness for capturing data dependencies. In this chapter, we propose to use a more expressive model (dubbed as stateful behavior model) for capturing the data dependencies as the preconditions of invoking an API. The data dependencies are in the form of first order Boolean formula (a.k.a., propositions). There have been attempts to learn stateful behavior models in the literature. However, these approaches are based on learn the DFA and the propositions independently. There have been approaches which generate classic behavior models based on active automata learning. However, we are the first to combine the invariant learning with automata learning to simultaneously learn the stateful behavior model in an efficient way.

3.1 Introduction

Most programs contain both data and methods which are used to manipulate the data and thus whose behavior often depends on the data. A behavior model is in essence a finite state automaton which can only recognize a regular language and thus is unable to capture behaviors of data-rich programs. Consider a simple example of the Java class `java.util.Stack` with one instance field `elementCount` which denotes the number of elements in the stack object and two methods `push` and `pop`. A behavior model of the `Stack` class should specify the following language: the number of
method calls to the `push` method in any valid trace of the model must be no less than the number of method calls to method `pop`. This language is known to be irregular because the number of method calls could be infinite and therefore beyond the expressiveness of finite-state automata. On the other hand, this language can be recognized by an augmented finite state automaton (which is called stateful behavior model, see definition 3.1) whose transitions for the `pop` method are extended with a Boolean formula \( \text{elementCount} \geq 1 \).

There are two main challenges in automatic generation of stateful behavior models for a program. The first challenge is how to automatically and systematically find these Boolean formula. The second challenge is how to inference the stateful behavior model with these Boolean formula when given.

As introduced in last chapter, there are two groups of approaches for learning stateful behavior models, namely passive learning approaches and active learning approaches. Passive learning based approaches assumes there are a set of traces and try to generate a behavior model from the given traces using different Boolean formula inference techniques and passive automata inference algorithms. For instance, Dallmeier et al. [66] propose to learn stateful behavior models for Java classes by using template based invariant inference techniques to infer the Boolean conditions for methods first and then using passive automata inference on the abstracted traces to infer the stateful behavior model. In general, these approaches are scalable but the quality of the generated stateful behavior models depends on the quality of the given traces as well as the inferences techniques for Boolean formula and finite state automata.

The other group of approaches attack the two challenges simultaneously by extending active automata learning algorithm, in particular the \( L^* \) algorithm (cf. Section 2.2.1.1), with relatively heavy-weight techniques to learn accurate stateful behavior models. For instance, Alur et al. [22] propose to synthesize interface specifications (can be viewed as a special form of stateful behavior model) for Java classes by combining \( L^* \) algorithm and predicate abstraction, which relies on theorem proving. Similarly, Giannakopoulou et al. [80] propose to learn interface specifications by extending \( L^* \) with symbolic execution, which relies on SMT solving. The generated stateful behavior models are precise at cost of efficiency because current theorem proving and SMT/SAT
techniques incur high performance overhead for handling complicated data structures and control flows. Thus the approaches in this group can only handle small programs (with less than 200 lines of source code [80]).

In this chapter, we propose an efficient approach to learning stateful behavior models for single classes. The central idea of this approach is to combine the $L^*$ algorithm [24,163], which learns the behavior model, and automatic Boolean formula learning, which learns the guard conditions on data states, together to learn stateful behavior models simultaneously. Given the source code of a class as the only input, our approach generates a stateful behavior model for the class through an orchestrated sequence of learning and refinements. The framework of our approach is shown in Figure 3.1. There are three main modules. The learner and tester acts as the $L^*$ algorithm and the teacher, respectively. However, the two in combination can only learn behavior models. The refiner is used by the learner to generate the guard conditions on-the-fly in order to learn stateful behavior models. The main difference between our approach and the original $L^*$ algorithm are the following: in the original $L^*$ algorithm, the model to be learned is assumed to be a DFA but the model to be learned in our setting is the source code of the class. A trace (a sequence of method calls) can be either valid or invalid but never both according to the DFA. However, in our setting, two executions for the same sequence of method calls on the same object may result in different outcomes (i.e., error or non-error). This is because different parameters for the methods in the trace are used in the two executions and may result in different data states. If the same trace results in different execution results, the learner uses the refiner to perform an alphabet refinement by splitting one alphabet symbol (a method) into two symbols. Each of the two new symbols has a different guard condition so that the two executions of the same trace are distinguishable. The refiner automatically generates a proper guard condition when given different data states.
Chapter 3. Learning Stateful Models

3.2 Approach Overview

In the section, we use an example to show that the $L^*$ algorithm does not work when the system to be learned cannot be captured as a DFA and to show how and why our new approach can generate a stateful behavior model. The Java class `java.util.Stack` in the Java standard library is used as a running example throughout this chapter. To simplify the presentation, we only consider the methods `push` and `pop` and one instance field `eleCount` of the `Stack` class. The `push` method takes an integer object as the input and the `pop` method pop up the top-most element of the internal array of the stack and takes no input; The instance field `elementCount` which is defined in and inherited from the class `java.util.Vector` represents the number of elements in the internal array of the stack. At the beginning of learning, the alphabet contains only two events which correspond to the two methods.

First, $L^*$ fills the row indexes and column indexes with the empty string $\langle \rangle$ and ask whether the empty string is accepted by $D$. Thus at this moment, $R = \{\langle \rangle\}$, $RA = \emptyset$, and $E = \{\langle \rangle\}$. We use the execution result of the default constructor of class `Stack` to represent the whether the empty string is accepted by $D$. Thus the execution of constructor is accepted, thus the value of cell indexed by $\langle \rangle, \langle \rangle$ is 1.

Then for each symbol in $\Sigma$ it asks a membership query and inserts the results into $RA$ rows. In this case it asks membership queries for string $\langle push \rangle = \langle push \rangle \cdot \langle \rangle$ and string $\langle pop \rangle = \langle pop \rangle \cdot \langle \rangle$. The answer is 1 and 0, respectively. The observation table is shown in Figure 3.2 (a), for which $R = \{\langle \rangle\}$, $RA = \{\langle push \rangle, \langle pop \rangle\}$, and $E = \{\langle \rangle\}$.
Chapter 3. Learning Stateful Models

31

Figure 3.3: The second closed and consistent observation table (a) and its corresponding candidate DFA (b) generated by the $L^*$ algorithm.

The table is not closed because the $Row(\langle \text{pop} \rangle) = 0$ is not indexed by any string in $R$. Thus the algorithm moves the row $\langle \text{pop} \rangle$ from $RA$ to $R$.

It then asks membership queries for string $\langle \text{pop, push} \rangle = \langle \text{pop, push} \rangle \cdot \langle \rangle$ and string $\langle \text{pop, pop} \rangle = \langle \text{pop, pop} \rangle \cdot \langle \rangle$ and adds the results to the $RA$. The table is shown in Figure 3.2 (b), for which $R = \{\langle \rangle, \langle \text{pop} \rangle\}$, $RA = \{\langle \text{push} \rangle, \langle \text{pop, push} \rangle, \langle \text{pop, pop} \rangle\}$, and $E = \{\langle \rangle\}$ and the table is closed and consistent.

$L^*$ constructs a candidate DFA $C_1$ shown in Figure 3.2 (c) from the table shown in Figure 3.2 (b) by following the above procedure: the two states labeled with 1 and 0 correspond to the two rows in the $R$ rows, i.e., state 1 = $[\langle \rangle]_r$ and state 0 = $[\langle \text{pop} \rangle]_r$. State 1 is the initial state because its row index is $\langle \rangle$ and it is also an accepting state because its cell value indexed by the column index $\langle \rangle$ is 1. The transition $\text{push}$ from state 1 goes to state 1 because the representing string for state 1 is $\langle \rangle$ and the row indexed by row index $\langle \rangle \cdot \text{push} = \langle \text{push} \rangle$ is 1. The transition $\text{pop}$ from state 1 goes to state 0 because the row indexed by row index $\langle \rangle \cdot \text{pop} = \langle \text{pop} \rangle$ is 0. The transition $\text{push}$ from state 0 goes to state 0 because the representing string for state 0 is $\langle \text{pop} \rangle$ and the row indexed by row index $\langle \text{pop, push} \rangle = \langle \text{pop} \rangle \cdot \text{push}$ is 0. The transition $\text{pop}$ from state 0 goes to state 0 because the row indexed by row index $\langle \text{pop, pop} \rangle = \langle \text{pop} \rangle \cdot \text{pop}$ is 0.

Next, $L^*$ asks a candidate query for the teacher to check whether $C_1$ accepts the same language as $D$. The trace $\text{push} \cdot \text{pop}$ is accepted by $D$ but not accepted by $C_1$ and thus it is returned by the teacher as a counterexample for $L^*$. For finding the witness suffix $WS(\langle \text{push, pop} \rangle)$, since the longest prefix of the counterexample in $P$ is $\langle \text{push} \rangle$, thus $WS(\langle \text{push, pop} \rangle)$ is $\langle \text{pop} \rangle$, we add all its suffixes $\{\langle \text{pop}, \langle \rangle\}$ into $E$, but
Chapter 3. Learning Stateful Models

<table>
<thead>
<tr>
<th></th>
<th>⟨⟩</th>
<th>⟨pop⟩</th>
<th>⟨pop, pop⟩</th>
</tr>
</thead>
<tbody>
<tr>
<td>⟨⟩</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>⟨pop⟩</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>⟨push⟩</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>⟨push, push⟩</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>⟨pop, push⟩</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>⟨pop, pop⟩</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>⟨push, pop⟩</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>⟨push, push, push⟩</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>⟨push, push, pop⟩</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

(a)

FIGURE 3.4: The third closed and consistent observation table (a) and its corresponding candidate DFA (b) generated by the \( L^* \) algorithm.

the ⟨⟩ is already in \( E \), thus only ⟨pop⟩ is added into \( E \). Then \( L^* \) asks several membership queries to make the table closed. The intermediate tables are not shown to save space. The observation table as shown in Figure 3.3 (a) becomes closed and consistent again. From this observation table, \( L^* \) constructs another candidate DFA \( C_2 \) as shown in Figure 3.3 (b) and asks a candidate query for \( C_2 \).

The teacher returns the trace ⟨push, push, pop, pop⟩ as the counterexample for this candidate query. Only the witness suffix ⟨pop, pop⟩ itself is added into \( E \) because its suffixes ⟨pop⟩ and ⟨⟩ are already in \( E \). Then \( L^* \) makes several membership queries to close the observation table again. The closed table as shown in Figure 3.4 (a) is also consistent. From this observation table, \( L^* \) constructs the third candidate DFA \( C_3 \) as shown in Figure 3.4 (b) and asks a candidate query for it. It becomes quite clear that for the \( i \)-th candidate query where \( i \in \mathbb{N} \), the teacher always returns a counterexample string ⟨push\(^i\), pop\(^i\)⟩ which is accepted by the Stack class showing that the \( i \)-th candidate DFA \( C_i \) does not accept the counterexample string. There is always one additional state that needs be added to \( C_i \) and thus \( L^* \) does not terminate.

When given an instance of the java.util.Stack class for our new approach, the learner asks a number of membership queries. For the trace in a membership query, the tester generates several test cases which have the same sequence of method calls as the trace but each of which has different arguments and then answers the membership
query according to the results of these test cases. The trace for the query and the testing results are recorded in the observation table (refer to details in Section 2.2.1.1), as shown in Figure 3.5 (a). A trace \( \langle [\text{true}]\text{pop}, [\text{true}]\text{push} \rangle \) denotes the sequence of calling \text{pop} and then \text{push}. An 0 in the observation table denotes that all tests generated for the trace which is the concatenation the cell’s row index trace and column index trace throw an exception or violate an assertion (hereafter referred as failure). A 1 denotes that all the tests result execute normally (without any failure). For example, the 1 for the cell by row index trace \( \langle [\text{true}]\text{push} \rangle \) and column index trace \( \langle \rangle \) says that all tests for the trace \( \langle [\text{true}]\text{push} \rangle \cdot \langle \rangle \) execute normally and the 0 for the cell indexed by row index trace \( \langle [\text{true}]\text{pop} \rangle \) and column index trace \( \langle \rangle \) denotes the trace \( \langle [\text{true}]\text{pop} \rangle \cdot \langle \rangle \) result in failure. Based on the observation table, the learner generates a candidate stateful model as shown in Figure 3.5 (b). Note that the stateful model is essentially a finite-state automaton with state \( A \) as the only accepting state.

Next, the learner asks a candidate query to check whether the candidate model in Figure 3.5 (b) is consistent with the target class. The tester first uses random walk on the candidate model to generate a set of traces and then randomly generates a set of tests from these traces. A trace of the candidate model is either accepting when its final state is an accepting state or rejecting when its final state is not an accepting state. From the randomly generated tests, the tester finds an inconsistency between the candidate model and the target class: the candidate model says that calling \([\text{true}]\text{pop}\) from state \( A \) always lead to failure. However, this is not true according to the target class. For example, calling method \([\text{true}]\text{push}\) first (which leads to state \( A \)) and then \([\text{true}]\text{pop}\) results in no failure.
An inconsistency implies that the candidate model does not comply with the Stack class and thus needs to be refined. The data states of the stack at state $A$ before calling method `pop` are partitioned to two sets: objects in the first set does not lead to failure (no exception is thrown) after invoking `pop` and objects in the second set lead to failure after invoking `pop`. Then the refiner is used to find a proposition $\phi$ such that all data objects in the first set satisfy $\phi$ and all the data objects in the second set violate $\phi$. The problem of finding a divider to separate two sets of data is the classic classification problem in machine machining. The refiner uses a technique based on Support Vector Machines (SVMs) [166] to find $\phi$. In the above example, the proposition $\text{elementCount} \geq 1$ is generated for state $A$. Next, we re-start the learning process with a new alphabet which contains three events: `push`, $\text{elementCount} \geq 1$`pop` and $\text{elementCount} \leq 0`pop$, where $\text{elementCount} \leq 0$ is equivalent to $!(\text{elementCount} \geq 1)$ given the fact that $\text{elementCount}$ is an integer. The alphabet symbol $\text{elementCount} \geq 1`pop$ denotes that calling `pop` on a stack object whose data state satisfies the proposition $\text{elementCount} \geq 1$. After a series of membership queries, the learner constructs the observation table as shown in Figure 3.6 (a). Note that all tests for the trace $\langle \text{elementCount} \geq 1`pop \rangle$ do not lead to failure and therefore its result for membership query is 1 (the first row behind the dashed line in the table). All tests for the symbol
[elementCount ≤ 0]pop results in failure and therefore the results of all the traces (e.g., ([elementCount ≤ 0]pop) and ([elementCount ≤ 0]pop, [true]push)) which contain the symbol [elementCount ≤ 0]pop are marked as 0. A new candidate model is generated from the table, as shown in Figure 3.6 (b). The tester performs random walking again and finds no inconsistency. We then present Figure 3.6 (b) as the final model.

The main contribution of our approach is on using a refiner to do automatic refinement of alphabet used in the $L^*$ algorithm in order to learn stateful behavior models for data-rich programs. In particular, by adopting a classification technique to find the root cause of inconsistent executions, our approach is capable of automatically generating propositions for alphabet refinement from the data states generated when answering queries. The refiner can be viewed as an automatic mapper \cite{15} between the abstract traces used by the learner and concrete tests generated by the tester. Our approach outperforms existing active learning approaches (e.g., \cite{22, 80}) based on symbolic execution or therefrom proving as it uses testing which is much more efficient than SMT solving and theorem proving. Furthermore, to learn concise stateful behavior models efficiently. The quality of the generated models is better than models generated by existing passive learning approach as our approach can generates the Boolean conditions and the automata systematically instead using ad-hoc inference techniques. We implemented the proposed approach in a tool named TzuYu\footnote{TzuYu is named after the best student of Confucius.} and the evaluation of the tool shows that our approach is able to learn meaningful and concise stateful behavior models for data-rich programs efficiently.

The rest of this chapter is organized as follows. Section 3.3 presents the definition of stateful behavior models. Section 3.4 to Section 3.6 present the details of the three components of our approach. Section 3.7 presents some implementation details of the proposed approach and Section 3.8 presents the experiments to evaluate the performance of the proposed approach. Section 3.9 concludes this chapter and discuss some related work.


3.3 Definitions

In this section we formalize the definitions related to stateful behavior model. The only
input to our approach is a class (e.g., the java.util.Stack class) which is called
the target class or the class under analysis. A class contains a set of public methods and
a set of instance fields which could be instances of other classes. We use an object of the
target class as the main receiver of method calls in a test. The behaviors of all instances
of the target class are inspected through this object. An object state is an evaluation of all
instance fields of the main receiver. For each object, there is an initial object state which
is the valuation of its instance fields right after calling one of its constructor\(^2\). Note that
an instance field may be a reference type which contains other instance fields defined.
Thus the instance fields of an object contain all the instance fields defined directly in it
as well as instance fields defined contained in its instance fields recursively. A method
is a function which takes one object state and returns a new one. A concrete execution
\( \text{ex} \) of an object is a finite sequence
\[
\text{ex} = (o_0, m_0(p_i^0), o_1, m_1(p_i^1), \ldots, o_n, m_n(p_i^n), o_{n+1})
\]
where \( o_i \) is an object state and \( m_i(p_i) \) is a method call with concrete arguments \( p_i \).
A failed execution is an execution which results in an exception or assertion failure. A
successful execution is an execution which does not fail.

The goal of our approach is to learn a stateful behavior model for the target class. A
stateful behavior model is defined on a DFA (cf., Section 2.1.1) as the following:

**Definition 3.1. (Stateful Behavior Model)** A stateful behavior model (stateful model
for short in this chapter) of a class is a tuple \( T = (\text{Prop}, \text{Meth}, \mathcal{D}) \), where \( \text{Prop} \)
is a set of propositions which are Boolean expressions over instance fields of the class, \( \text{Meth} \)
is a set of method names in the target class, \( \mathcal{D} = (S, \Sigma, \text{init}, \delta, F) \) is a DFA such that
\( \Sigma \subseteq \text{Prop} \times \text{Meth} \).

A proposition in \( \text{Prop} \) for the java.util.Stack class is a Boolean formula
defined over instance fields of the java.util.Stack such as \text{elementCount},
\text{capacityIncrement} which is defined in and inherited from java.util.Vector,
\[^2\text{A constructor of the target class is treated in a similar way as an ordinary method except that the following rule is enforced: it can only be used as the first method call of a trace and calling it in other places of a trace leads to failure.}\]
and any instance field of the array `elementData` (e.g., `elementData.length`). The set `Meth` contains `push` and `pop`. According to this definition, stateful models are deterministic. A symbol (or event) in $\Sigma$ is a pair of a proposition $g$ in $Prop$ and a method name $e$ in $Meth$ and $g$ is called the guard condition for $e$. A transition in $T$ is written as $(s, [g]e, s')$, where $s$ and $s'$ are in $S$. A stateful model is an abstract model of all executions of an object of the target class. In particular, a trace $tr = \langle s_0, [g_0]e_0, s_1, [g_1]e_1, \cdots, s_n, [g_n]e_n, s_{n+1} \rangle$ is an abstraction of the execution $ex$ above if they have the same sequence of methods (i.e., $e_i = m_i$ for all $i$) and all the guard conditions are satisfied (i.e., $g_i$ is satisfied by $o_i$ and method arguments $\rightarrow p_i$ for all $i$). We denote the set of concrete executions of $tr$ as $con(tr)$. Given an execution $ex$ and an alphabet $\Sigma$, we can obtain the corresponding trace, denoted as $abs(ex)$, by testing which proposition in $Prop$ is satisfied for each method call in $ex$.

**Definition 3.2. (Safety or Soundness)** A stateful model $T$ is safe (or sound), if for every accepting trace $tr$ of $D$ every execution in $con(tr)$ is successful.

**Definition 3.3. (Completeness)** A stateful model $T$ is complete, if for every concrete execution $ex$ of its target class there is an accepting trace $tr$ such that $ex \in con(tr)$.

The two definitions are referenced later in this thesis to discuss the capabilities of the proposed approaches regarding whether they can learn safe or sound models or complete model. In the following three sections we first introduce the detailed design of the tester and refiner and then introduce the learner which interacts with the tester and learner to learn the stateful model.

### 3.4 The Tester

In our approach, the tester is used as the teacher to answer membership queries and candidate queries asked by the learner. In the original setting of the $L^*$ algorithm, the teacher is required to answer either $yes$ or $no$ for the trace $tr$ in a given a membership query. In our setting, the $tr$ is an abstract trace because the parameters for the method calls in it are not specified. In order to check whether $tr$ is valid or not, the tester needs
to check all concrete executions $con(tr)$ with concrete parameters for method calls in $tr$ are successful or not. The tester is required to answer yes iff all executions in $con(tr)$ are successful and to answer no iff all executions in $con(tr)$ are failed. Similarly, given a candidate query, the tester is required to answer yes iff the candidate model is safe and complete with respect to the target class.

The tester in our approach is not able to answer definitely (either yes or no) for a trace $tr$ due to the following two problems: (1) the set $con(tr)$ is infinite in general because an infinite number of arguments can be generated for method calls in $tr$. And checking whether an infinite number of executions in $con(tr)$ are successful or not is highly non-trivial, if not impossible; (2) Some executions in $con(tr)$ could be successful while the rest executions are failed because the behavior of a method call may depend on its arguments. For instance, given the target class java.util.Vector and one trace $tr = \langle [true] addAll \rangle$. A concrete execution for $tr$ with the null as the argument of the method call $addAll$ results in an exception (a java.lang.NullPointerException exception is thrown by $addAll$), whereas an execution of $tr$ with a non-null argument for $addAll$ is successful. We tackle the first problem with a guided random testing technique in the following and the second problem with an automatic alphabet refinement technique in Section 3.5.

In the following, we show how the tester answers membership queries and candidate queries. Given a membership query with a trace $tr = \langle s_0, [g_0] m_0, s_1, [g_1] m_1, \cdots, s_n, [g_n] m_n, s_{n+1} \rangle$, the tester needs to find multiple concrete executions in the form of $\langle o_0, m_1(p_1), o_2, m_2(p_2), \cdots, o_k, m_k(p_k), o_{k+1} \rangle$. That is to say, the tester needs to automatically generate the arguments for all method calls in $tr$ such that all guard conditions $g_i$ are satisfied. This task is in general highly non-trivial and requires techniques like SAT/SMT solving. For better efficiency, we instead apply a much more efficient test generation technique to generate argument in the hope that the guard conditions are satisfied. In particular, we use the Randoop [147] approach to generate arguments for all method calls in a trace.

Given a trace $tr$, we generate arguments for each method call one-by-one in sequence. Given a typed parameter of a method call, the idea is to randomly generate a value from a pool of type-compatible values. This pool composes of a set of pre-defined
value (e.g., a random integer for an integer type, \texttt{null} or an object with the default object state for a user-defined class, etc.) but also type-compatible objects that have been generated during the testing process. We remark that in order to re-create the same object, we associate each object with the execution which produces the object state. Given one value for each parameter, we then evaluate whether \( g_i \) is true or not. If \( g_i \) is true, we proceed with next method call.

There are four possible outcomes of the random testing. If all tests are successful, the answer to the query is yes, i.e., \( tr \) should be an accepting trace. If all tests are failed, the answer is no, i.e., \( tr \) should be a non-accepting trace. If there are both successful tests and failed tests (for \( tr \) or a prefix of \( tr \)), the tests are passed to the refiner for alphabet refinement as we show later. Lastly, due to the limitation of random testing (i.e., the price we pay to avoid theorem proving), it is possible that some guard condition \( g_i \) is never satisfied by the generated arguments. In other words, we fail to find any concrete execution in \( \text{con}(tr) \). In such a case, we optimistically answer yes so that the resultant stateful behavior model is more permissive.

To answer a candidate query with a stateful model \( C \), we use random walk [54, 56, 111] to generate a suite of traces from \( C \). And the approach of Randoop [147] is used again to generate multiple test cases for each trace in the suite. Test cases which are inconsistent with the candidate model are classified into two categories: positive counterexamples and negative counterexamples. A positive counterexample is a successful test whose corresponding trace \( tr \) is non-accepting according to \( C \). A negative counterexample is a failed test whose corresponding trace \( tr \) is accepting according to \( C \). If both categories are empty, we answer the candidate query with a \texttt{yes} and the learner presents the candidate model as the final output. If either of the two categories is not empty, the candidate model is not consistent with the target class and a counterexample must be returned to the learner. In the original \( L^* \) algorithm, any of the counterexamples (either a positive or a negative counterexample) can be returned. However, it is more complicated in our setting as explained in the following.

For each state \( s \) in the stateful model \( C \), we identify a set of executions in the test suite which end at the state, denoted as \( E_s \). For each \( e \in \Sigma \), we extend each execution in \( E_s \) with a method call corresponding to \( e \) and obtain a new set denoted as \( E^e_s \). If
all of the executions result in failure whereas a transition labeled with $e$ from $s$ leads to an accepting state in $C$, the tester reports that $C$ is invalid and picks one execution in $E_s^e$ and presents its corresponding abstract trace as a counterexample. Similarly, if all of the executions are successful, whereas a transition labeled with $e$ from $s$ leads to a non-accepting state, the tester presents a counterexample. Lastly, if some of the executions in $E_s^e$ result in failure and others result in success, the refiner is consulted to perform alphabet refinement.

### 3.5 The Refiner

The refiner will be used by the learner to generate dividers for alphabet refinement in two cases. The first case is when the set of executions in $con(tr)$ for the trace $tr$ in a membership query contains both failed and successful tests. We use $T^-$ to denote the set of failed executions which result in failure when the last method call is performed (with the generated arguments) and $T^+$ to denote the set of successful executions which do not result in failure when the last method call is performed. In this case, alphabet refinement must be performed so as to distinguish the two sets of executions; otherwise the traces for all the executions are the same (i.e., $tr$) and therefore the executions are indistinguishable to the learner.

Given an execution $ex = \langle o_0, m_1(p_1), o_2, m_2(p_2), \ldots, o_n, m_n(p_n), o_{n+1} \rangle$ in $T^-$ or $T^+$, we can obtain a data state pair $(o_n, p_n)$ where $o_n$ is the object state of the main receiver object prior to the last method call $m_n$ and $p_n$ is the list of arguments of $m_n$. Note that because in this case all the executions have the same trace and therefore the $p_n$s in all executions are the same. For the execution $ex \in T^-$, $o_{n+1}$ is not defined because the method call to $m_n$ failed. Let $O^-$ be the set of all pairs we collect for method $m_n$ of executions in $T^-$ and $O^+$ be the set of all pairs we collect for method $m_n$ of executions in $T^+$. Intuitively, there must be something different between $O^-$ and $O^+$ such that executions in $T^-$ failed and executions in $T^+$ succeeded. The refiner’s job is to find a divider, in the form of a proposition, such that $O^-$ can be distinguished from $O^+$. Formally, a divider for $O^+$ and $O^-$ is a proposition $\phi$ such that for all $o \in O^+$, $o$ satisfies $\phi$ and for all $o' \in O^-$, and $o'$ does not satisfy $\phi$. From another point of view,
there must be some invariant for all object states in \( O^+ \) (denoted as \( \text{inv}^+ \)) and some invariant for all object states in \( O^- \) (denoted as \( \text{inv}^- \)) such that \( \text{inv}^+ \) implies \( \phi \) and \( \text{inv}^- \) implies the negation of \( \phi \).

The above problem can be viewed as a binary classification problem in machine learning. Therefore, the refiner in our approach is based on a classification technique developed by the machine learning community, in particular, Support Vector Machines (SVMs) [166]. SVM (cf. Section 2.2.3.1) is a supervised machine learning algorithm for classification and regression analysis. We use the linear binary classification functionality of SVM. Mathematically, the linear binary classification functionality of SVMs works as follows: given two sets of data states (say \( O^+ \) and \( O^- \)) and each data state can be viewed as a vector of numerical values (e.g., floating-point numbers), SVM tries to find a separating hyperplane in the form of

\[
\Sigma_{i=1}^{n} c_i \cdot x_i = c
\]

such that 1) every positive data state \( p_1, p_2, \ldots, p_k \in O^+ \) satisfies the formula \( \Sigma_{i=1}^{k} c_i \cdot p_i > c \) and 2) for every negative data state \( n_1, n_2, \ldots, n_k \in O^- \) satisfies \( \Sigma_{i=1}^{k} c_i \cdot n_i < c \). As long as \( O^+ \) and \( O^- \) are linear separable, SVM is able to find a separating hyperplane, even if the invariants \( \text{inv}^+ \) and \( \text{inv}^- \) may not be linear. As discussed in Section 2.2.3.1 of Chapter 2, there are usually multiple hyperplanes that can separate \( O^+ \) from \( O^- \). The optimal margin classifier (see the definition in Section 2.2.3.1) is used in our approach. This separating hyperplane can be seen as the strongest witness why the two sets of data states are different.

SVM takes two sets of vectors of numerical values as input and outputs a divider. Therefore, in order to apply the SVM technique, each data state in \( O^+ \) or \( O^- \) must be casted into a vector of numerical values. In general, a Java program contains instance fields of numerical types (e.g., \text{double}), of categorical types (e.g., \text{Enum}) as well as of reference types (e.g., \text{java.util.Stack}). Thus, an automatic and systematic way of mapping an arbitrary object state to a vector of numerical values is needed so as to apply SVM techniques. Furthermore, the inverse mapping is also important to feed the SVM outputs (hyperplanes) back to the original program so that the generated dividers are amenable for human understanding. Our solution is to systematically generate a numerical value graph from each object state from the numerical type graph of its type.
and apply SVM techniques to values associated with nodes in the graph level-by-level. We illustrate this process using an example in the following.

Figure 3.7 shows part of the numerical type graph for the target class `java.util.Stack` without showing many irrelevant fields of classes for readability. A rounded rectangle represents a reference type and a circle denotes a primitive type (e.g., I and B stands for integer and Boolean respectively). We add a Boolean type field `isNull` for a reference type field to denote whether that the reference type field is null. An edge in the graph denotes that the type in the source node contains a field of type in the target node and the name of the field is the label on the edge. For instance, the `java.util.Stack` class contains an instance field named `elementData` of type `Array`. The Java `Array` type in turn contains instances of type `java.lang.Object`. That is to say the Stack class contains one instance field named `data` which is an array of `java.lang.Objects`. To obtain a vector of numerical values for an object state, we first generate a numerical value graph for this object and its numerical type graph and then traverse the numerical value graph level-by-level to collect the numerical values associated with each primitive fields. In general, the numerical value graph for an object could be huge if its class contains many fields. However, it is often sufficient to look at only the top few levels for learning stateful
models.

Figure 3.8: The numerical value graphs for an `java.util.Stack` object with one element (a) and the numerical value graph for an empty Stack object (b). The value in a circle is the value for the corresponding primitive types in its numerical type graph.

In the following, we explain how this numerical value graph is generated and used to generate vectors of numerical values as inputs for SVM. Assume the last event of the membership query is `true` `pop` and the two sets of object states are $O^+$ and $O^-$ prior to this method call. The set $O^+$ contains an object state $O_1$ which is a Stack object and contains one element before calling `true` `pop`; the set $O^-$ contains only one Stack object $O_2$ which is an empty Stack. For each object state in $O^+$ or $O^-$, we can create a numerical value graph according to the numeric type graph in Figure 3.7. The numerical value graph for $O_1$ and $O_2$ is shown in Figure 3.8(a) and in Figure 3.8(b), respectively. The value graph for the stack object in $O^+$ contains one element and thus it is not null and thus its artificial field `isNull` has value 0. The field `capacityIncrement` is initialized to 0 by the default constructor of `java.util.Stack` and thus has value 0. The field `elementCount` represents the number of elements the stack object has and thus its value is 1. The field `elementData` is initialized by the default constructor to an array of 10 elements. Therefore, its artificial field `isNull` is 0 and the field `length` has value 10. The first and only element of in the array is thus not null and the artificial field `isNull` has value 0. The second value graph can be built in the same way and
the only difference is that the stack object is empty and the field `elementCount` has value 0 and the `element.isNull` has value 1.

After generating the value graph for each object state, the refiner first abstracts $O^+$ and $O^-$ using numerical values at the first level of the graph, i.e., `receiver.isNull` for all objects. The set of numerical value vectors for $O^+$ and $O^-$ at the first level of value graphs is $\{\langle 0 \rangle\}$ and $\{\langle 0 \rangle\}$, respectively. Since the two sets of objects are not distinguishable using numerical values at the first level, the refiner proceeds to use the numerical values at the second level, i.e., `receiver.elementData.isNull`, `receiver.elementCount` and `receiver.capacityIncrement`. The set of numerical value vectors for $O^+$ and $O^-$ at the second level of value graphs is $\{\langle 0, 1, 0 \rangle\}$ and $\{\langle 0, 0, 0 \rangle\}$, respectively. The optimal margin hyperplane for the vectors is `receiver.elementCount \geq 0.5` because it is satisfied by the only numerical value vector $\langle 0, 1, 0 \rangle \in O^+$ and its negation `receiver.elementCount < 0.5` is satisfied by the numerical value vector $\langle 0, 0, 0 \rangle \in O^-$. Given the fact that field `elementCount` is integer and thus the divider is transformed and returned to learner as `receiver.elementCount \geq 1`. The refiner proceeds to the next level of field values until either it finds a divider or the maximum depth is reached. The reason for us to look for a divider over variables from top levels to lower levels in the value graph is based on the principle of encapsulation. Due to this principle, the values of fields defined directly in the target class (correspond to higher levels of fields in the value graph) is more likely than those of fields defined in its referenced classes (correspond to lower levels of fields in the value graph) to influence the behavior of the method.

The other case the refiner is used is when an inconsistency is found in a candidate query. An inconsistency in candidate query indicates that there is a set of executions which end in the same state $s$ in the candidate model $C$ but the extensions of these executions with a method call $e = [\phi]m$ result in both failure and success. Similar to the case of a membership query, for each execution we obtain a pair $(o, \overrightarrow{p})$ where $o$ is the object state of the main receiver at state $s$ and $\overrightarrow{p}$ is the arguments of $m$. Similarly, we collect two sets of those pairs $O^+$ (from successful extensions) and $O^-$ (for failed extensions). Given the two sets of pairs, the SVM is used to generate a divider in the same way as in the membership queries.
3.6 The Learner

The learner drives the learning process and interacts with both the tester and refiner. It uses an algorithm which extends the $L^*$ algorithm \[24, 163\] with lazy alphabet refinement.

In general, the behavior model for a data-rich program requires more expressiveness than DFA and therefore the $L^*$ algorithm itself is not sufficient as demonstrated in the first example in Section 3.2. We solve this problem by extending the $L^*$ algorithm with (lazy) alphabet refinement, i.e., by introducing propositions on object states into the alphabet. The details on the extended $L^*$ algorithm are presented in the following.

3.6.1 $L^*$ with Lazy Alphabet Refinement

When the refiner generates a divider $\phi$, an event $e = [g]m$ in $\Sigma$ is effectively split into two events: $[\phi \land g]m$ and $[!]\phi \land g]m$. With a modified alphabet $\Sigma' = \Sigma \setminus \{[g]m\} \cup \{[\phi \land g]e, [!]\phi \land g]e\}$, previous learning results are invalidated and therefore learning needs to be re-started. However, re-starting from scratch is costly, as we often need multiple rounds of alphabet refinement. In the following, we show how to extend the $L^*$ algorithm with lazy alphabet refinement so as to re-use previous learning results as much as possible.

Algorithm 1 shows the pseudo-code of the $L^*$ algorithm with lazy alphabet refinement, where $Q_m(tr)$ denotes the membership query with the trace $tr$ and $Q_c(C)$ denotes the candidate query of a stateful model $C$. There are two cases where the alphabet refinement takes place: (1) when a membership query triggers the generation of a divider $\phi$ (lines 5, 13, 25), which means that some alphabet $e \in \Sigma$ needs to be split into $[\phi]e$ and $[!]\phi]e$, it calls Algorithm 2 to refine the alphabet and update the corresponding results of the membership queries. (2) A candidate query may also trigger the generation of a divider $\phi$ (line 19). If so, Algorithm 2 is also called to refine the alphabet and update the corresponding results of the membership queries in the observation table.
Algorithm 1: \(L^*\) Algorithm with Lazy Alphabet Refinement

1. Let \(P = E = \{\langle\rangle\}\)
2. \(\text{for } e \in \Sigma \cup \{\langle\rangle\} \text{ do}\)
3. \hspace{1em} Update \(T\) by \(Q_m(e)\)
4. \hspace{1em} if \(e\) needs to be split then
5. \hspace{2em} Split(\(\Sigma, e, (P, E, T)\))
6. \(\text{while } \text{true} \text{ do}\)
7. \hspace{1em} \(\text{while there exists } tr \cdot \langle e \rangle \text{ where } tr \in P \text{ and } e \in \Sigma \text{ such that } tr \cdot \langle e \rangle \not\equiv tr' \text{ for all } tr' \in P \text{ do}\)
8. \hspace{2em} \(P \leftarrow P \cup \{tr \cdot \langle e \rangle\}\)
9. \hspace{2em} \(\text{for } \sigma \in E \text{ do}\)
10. \hspace{3em} \(tr'' \leftarrow tr \cdot \langle e \rangle \cdot \langle \sigma \rangle\)
11. \hspace{3em} Update \(T\) by \(Q_m(tr'')\)
12. \hspace{3em} if there is some \(e' \in \Sigma\) needs to be split then
13. \hspace{4em} Split(\(\Sigma, e', (P, E, T)\))
14. \(\text{Construct candidate stateful behavior model } C \text{ from } (P, E, T)\)
15. \(\text{if } Q_c(C) = \text{Yes then}\)
16. \hspace{1em} return \(C\)
17. \(\text{else if there is some } e' \in \Sigma \text{ needs to be split then}\)
18. \hspace{2em} Split(\(\Sigma, e', (P, E, T)\))
19. \hspace{2em} \(v \leftarrow WS(ce)\) \hspace{1em} \(\text{ce is a counterexample}\)
20. \hspace{2em} \(E \leftarrow E \cup \{v\}\)
21. \hspace{2em} \(\text{for } tr \in P \text{ and } e \in \Sigma \text{ do}\)
22. \hspace{3em} Update \(T\) by \(Q_m(tr \cdot v)\) and \(Q_m(tr \cdot \langle e \rangle \cdot v)\)
23. \hspace{3em} if there is some \(e' \in \Sigma\) needs to be split then
24. \hspace{4em} Split(\(\Sigma, e', (P, E, T)\))

We use the Stack example to illustrate the new algorithm. Initially, the alphabet is \(\Sigma = \{[\text{true}]\text{push}, [\text{true}]\text{pop}\}\). After a series of membership queries, Algorithm 1 constructs the first candidate model, as shown in Figure 3.5 (b), based on the closed and consistent observation table shown in Figure 3.5 (a). A candidate query for the first model is asked for the tester. The tester finds that tests for traces which call \([\text{true}]\text{pop}\) on state \(A\) have both failed and successful executions. Then the refiner is invoked by the learner to find a proposition \(\text{elementCount} \geq 1\) for these inconsistent tests. The event \([\text{true}]\text{pop}\) is split into two events: \([\text{eleCount} \geq 1 \land \text{true}]\text{pop}\) which is equal to \([\text{eleCount} \geq 1]\text{pop}\) and \([!(\text{elementCount} \geq 1) \land \text{true}]\text{pop}\) which is equal to \([!(\text{elementCount} \geq 1)]\text{pop}\), and
Algorithm 2: Split(\(\Sigma, e = [g]m, (P, E, T)\))

1. \(\phi \leftarrow \text{Refiner}([g]m, O^+, O^-);\)
2. \(\Sigma \leftarrow \Sigma \cup \{[\phi \land g]m, ![\phi \land g]m\} \setminus \{e\};\)
3. for \(p = \langle p_1, p_2, \ldots, p_n \rangle \in P\) do
   4. if \(e = p_k \in p\) then
      5. \(p_1 = \langle p_1, \ldots, p_{k-1}, [\phi \land g]m, p_{k+1}, \ldots, p_n \rangle;\)
      6. \(p_2 = \langle p_1, \ldots, p_{k-1}, ![\phi \land g]m, p_{k+1}, \ldots, p_n \rangle;\)
      7. \(P \leftarrow P \cup \{p_1, p_2\} \setminus \{p\};\)
5. for \(q \in E\) do
   6. Update \(T\) by \(Q_m(p_1 \cdot q)\) and \(Q_m(p_2 \cdot q)\);
5. for \(q = \langle q_1, q_2, \ldots, q_n \rangle \in E\) do
   7. if \(e = q_k \in q\) then
      8. \(q_1 = \langle q_1, \ldots, q_{k-1}, [\phi \land g]m, q_{k+1}, \ldots, q_n \rangle;\)
      9. \(q_2 = \langle q_1, \ldots, q_{k-1}, ![\phi \land g]m, q_{k+1}, \ldots, q_n \rangle;\)
      10. \(E \leftarrow E \cup \{q_1, q_2\} \setminus \{q\};\)
5. for \(p \in P\) do
   11. Update \(T\) by \(Q_m(p \cdot q_1)\) and \(Q_m(p \cdot q_2)\);

\[
\begin{array}{l}
\langle\rangle \\
\langle\text{[true]push}\rangle \\
\quad \langle![\text{elementCount} \geq 1]\rangle\text{pop} \\
\quad \langle![\text{elementCount} \geq 1]\rangle\text{pop} \\
\quad \langle![\text{elementCount} \geq 1]\rangle\text{pop}\cdot\langle\text{[true]push}\rangle \\
\quad \langle![\text{elementCount} \geq 1]\rangle\text{pop}\cdot\langle\text{[true]push}\rangle \\
\quad \langle![\text{elementCount} \geq 1]\rangle\text{pop}\cdot\langle\text{[true]push}\rangle \\
\quad \langle![\text{elementCount} \geq 1]\rangle\text{pop}\cdot\langle\text{[true]push}\rangle \\
\quad \langle![\text{elementCount} \geq 1]\rangle\text{pop}\cdot\langle\text{[true]push}\rangle \\
\end{array}
\]

1
1
0
1
0
0
0
0
0
0

Figure 3.9: The observation table generated by the lazy \(L^*\) algorithm.

the \(L^*\) learning process is restarted from the scratch with the new alphabet \(\Sigma' = \{[\text{true}]\text{push}, [\text{elementCount} \geq 1]\text{pop}, ![\text{elementCount} \geq 1]\text{pop}\}\). Without lazy alphabet refinement, all the membership queries over the new alphabet \(\Sigma'\) have to be queried, as shown in the observation table in Figure 3.9. However, with lazy alphabet refinement, only the membership queries marked with the asterisk symbol (*) have to be queried. In this small example, only two membership queries are reduced due to the small alphabet size. In real-world examples, the size of alphabet is usually big, and the number of reduced membership queries is significant. The final model learned by Algorithm 1 is the same as the one shown in Figure 3.6 (b).
3.7 Implementation

In this section, we introduce the tool TzuYu which is a Java implementation of the proposed approach. TzuYu has over 20K lines of source code. In the following, we also discuss the challenges in implementing the proposed approaches and how they are addressed in TzuYu.

At first, we use Java Reflection to do class hierarchy analysis of each class referenced by the target class so as to collect relevant information like fields and methods of referenced classes. The numerical type graph for each referenced class is built by following its class hierarchy. The type graph uses the referenced types and hence it may reference many other types, but not all referenced types are useful for generating dividers. Therefore, we filter classes such as `java.lang.Thread`, `java.lang.Exception` and high level interfaces such as `java.io.Serializable`. The public methods defined in the target class identify the initial alphabet for the learner. Afterwards, the learner starts to generate membership queries and candidate queries according to Algorithm 1.

Given a membership query, the tester checks whether its trace is feasible or not by generating a number (which is configurable) of tests and uses reflection again to run the tests. During execution, the tester saves the runtime states of the arguments of each method. For argument generation, we develop a just-in-time approach, i.e., generate the required arguments just before executing a method. First we try to choose from existing parameters which pass the guard condition. If no existing argument can pass the guard condition, we generate a new argument in the hope that it can pass the guard condition. If the new argument does not satisfy the guard condition, we generate another set of arguments until the guard condition evaluates to true or a maximum number of tryout (configurable) is reached. Informally, an argument can be obtained from three sources, i.e., randomly generated from a set of pre-defined type compatible values; selected from existing executions that generate type compatible variables; or selected from type compatible out-referenced variables generated by the current execution. The above recursive argument generation procedure may not terminate for a recursive constructor which has
a parameter of the same class in which the constructor is defined. We set a maximum call depth for the recursive constructor as did by Lin et al. [118].

Before executing each method call, we store the object states of the receiver and the arguments as an instrumented state. We remark that using the Java standard clone mechanism to save object states is infeasible because the class may not implement java.io.Serializable or java.lang.Cloneable interface. We thus implement a mock-up mechanism similar to the standard clone mechanism in Java to save the runtime object into a mock-up object whose tree like class structure resembles the class structure of the original object. The mechanism differs from the standard clone mechanism in that only primitive type values of the object are saved. For reference type field we construct another mock-up object as its saved value. These mock-up objects can be used by the refiner. When the real object is needed, for instance, to generate a new test, we record the exact sequence of statements whose execution creates the object that can then be used to “clone” the arguments later by re-executing them.

Given a candidate query, the tester generates a number of tests from the candidate model. The default number (which is configurable) is twenty multiplied with the maximum length of traces generated in membership queries before this candidate query. Each testing trace is generated by depth first random walking on the model up to a fixed length, the length of the trace is set to two plus the maximum length of traces generated during membership queries. Due to randomness in random testing and random walking, a test case generated previously may not appear again later. To ensure the learning process is improving always (and hopefully converging), we store all the generated test cases so as to provide consistent answers. Notice that we do not store the instrumented states of the test case to reduce memory consumption and we re-execute the test case to create the states when they are needed (e.g., to evaluate the guard conditions).

One key step in our approach is to automatically generate a divider for alphabet refinement. We use the SVM techniques implemented in LIBSVM [50]. The first problem with using SVM is how to choose a good hyperplane as there are in theory an infinite set of hyperplanes which separate two sets of object states. The second problem is that the hyperplane discovered by LIBSVM often has float coefficients, which are often not as readable as integer values when we use them to build the model. Thus, we always
(if possible) choose integer coefficients which constitute a hyperplane which lies between the strongest and weakest hyperplane. Further, we implemented a few heuristics to preprocess the inputs to LIBSVM for generating a better divider. Firstly, we balance the positive and negative input data sets by duplicating data randomly chosen from the smaller set of the two, as SVM tends to build biased hyperplanes when the input data-set is imbalanced.

Secondly, because the arguments of method calls are generated randomly, LIBSVM may generate an incorrect divider. For instance, given a bounded stack with a size bound 5, if \( \text{push}(\text{element}) \) is invoked with \( \text{element} \) from \{1, 2, 3\} when the bounded stack is full, whereas it is invoked with \( \text{element} \) in \{5, 6, 7\} when the bounded stack is not full. LIBSVM may generate a divider \( \text{element} \geq 4 \) suggesting that calling \( \text{push}(\text{element}) \) with an input less than 4 will lead to failure. This is obviously incorrect. The problem is avoided with cross validation by checking whether the argument really affects the execution results. This is done by executing the successful (failed, respectively) traces whose arguments are substituted with arguments in the failed (successful, respectively) traces. For instance, in the above example, additional test cases are generated so that every invocation of \( \text{push}(\text{element}) \) is tested with the same set of input values, i.e., \{1, 2, 3, 5, 6, 7\}. As a result, if the argument is irrelevant to the execution result, it will be ruled out by cross validation.

3.8 Evaluation

In this section, we evaluate TzuYu on a set of Java library classes selected from the JDK and then compare TzuYu with existing tools. All the experiments were carried out on an Ubuntu 13.04 PC with 2.67 GHz Intel Core i7 Duo processors and 4 GB memory.

The set of target JDK classes used in this evaluation are shown in first column of Table 3.1 and these classes are also used by existing work [80, 179]. Column |M| shows the number of methods (exclusive of the constructors of the target class) which are used for alphabet of the learner. In this set of experiments, we generate two values for each parameter in each method. We set the maximum depth to five, which we found to be
Table 3.1: The runtime statistics for TzuYu. The entries above the dotted horizontal line are for small example programs and entries below it are for JDK classes.

| Target Class   | |M| |T_{total}| |Q_m| |Q_c| |T_r| |T^+| |I_R| |T_r| |Σ| |l| |
|----------------|---|---|---------|---|---|---|---|---|---|---|---|---|---|
| BoundedStack   | 2 | 764 | 21 | 4  | 98 | 69 | 4  | 138 | 4  | 2  |
| PipedOutputStream | 4 | 1548 | 48  | 5  | 160 | 71 | 5  | 59  | 7  | 2  |
| Signature      | 5 | 3227 | 75  | 6  | 200 | 102 | 8  | 156 | 9  | 2  |
| Stack          | 5 | 1177 | 39  | 4  | 120 | 83  | 4  | 59  | 7  | 2  |
| PipedOutputStream | 5 | 8343 | 75  | 6  | 200 | 48  | 8  | 5069 | 9  | 2  |

sufficient, for traversing the numerical value graphs in order to generate numerical value vectors for SVMs.

3.8.1 Results

Table 3.1 shows the various runtime statistics of TzuYu for these target classes. Column \( T_{total} \) shows the total time in milliseconds for learning stateful models. The next three columns show statistics about the learner. Column \(|Q_m|\) and \(|Q_c|\) shows the number of membership queries and candidate queries, respectively. Column \(|T_r|\) shows the number of traces generated by random walking. Column \(|T^+|\) shows the number of positive tests generated by the tester. Column \(|I_R|\) and \(T_r\) shows the number of SVM calls and the time in milliseconds taken by SVM to generate dividers, respectively. The last two columns show the size of alphabets and the number of states in the generated model, respectively.

The following observations are made based on the experimental results. Firstly, TzuYu successfully learned models in all cases in seconds. Furthermore, in most cases, the time taken by SVM is less than 20% of total time except for \texttt{java.io.PipedOutputStream} where the cross validation (in order to determine whether a method parameter is relevant) in a SVM call consumes a few seconds. Secondly, all learned models are sound and complete, which we confirm by comparing the learned one with the manually constructed actual one. Thirdly, the number of states in the learned model is minimum, i.e., two as we are differentiating two states only: failure or non-failure. This implies that for every method, whether invoking the method leads to failure or not can be determined by looking at the value of the data variables, and further, SVM is able to identify a suitable proposition every time. Lastly, we did not record
Table 3.2: Program invariants generated by Daikon, Psyco and TzuYu

<table>
<thead>
<tr>
<th>Method</th>
<th>Daikon</th>
<th>Psyco</th>
<th>TzuYu</th>
</tr>
</thead>
<tbody>
<tr>
<td>BoundedStack.push(Integer)</td>
<td>size one of [0, 1, 2]</td>
<td>-</td>
<td>size ≤ 2</td>
</tr>
<tr>
<td>BoundedStack.pop()</td>
<td>size one of [1, 2, 3]</td>
<td>-</td>
<td>size ≥ 1</td>
</tr>
<tr>
<td>PipedOutputStream.connect(snk)</td>
<td>sink = null ∧ snk ≠ null ∧ snk.connected = false</td>
<td>-</td>
<td>sink = null ∧ snk ≠ null</td>
</tr>
<tr>
<td>PipedOutputStream.write()</td>
<td>Signature.VERIFY = state</td>
<td>-</td>
<td>state ≥ 1 ∧ state ≤ 1</td>
</tr>
<tr>
<td>Signature.sign()</td>
<td>Signature.SIGN = state</td>
<td>-</td>
<td>state ≥ 1</td>
</tr>
<tr>
<td>Signature.update()</td>
<td>Signature.SIGN ≤ state</td>
<td>-</td>
<td>state ≥ 1</td>
</tr>
<tr>
<td>Stack.pop()</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Stack.peek()</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>PipedOutputStream.connect(snk)</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>PipedOutputStream.write(int)</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

the memory consumption due to the garbage collection feature of JVM. However, the memory consumption is relatively small since we did not store the instrumented states with the test cases and the number of test cases is relatively small which is linear in the number of candidate queries.

3.8.2 Comparison with Related Tools

We identified three closely related tools. PSYCO [80] is a symbolic execution based stateful behavior model learning tool; ADABU [66] is a dynamic behavior model mining framework and Daikon [75] is a dynamic invariant generator. We compare TzuYu with them in terms of time and the quality of the generated models. Table 3.2 shows the results of the invariants generated by the three tools and TzuYu. Notice that PSYCO is not publicly available; we thus only obtain the learned models documented in their paper [80].

We first compare the learned models as shown in Table 3.2. The invariants generated by ADABU are state invariants and they are omitted from Table 3.2. Methods with the trivial true invariant (e.g., size() in Stack) are also omitted. Both ADABU and Daikon need test cases as input to mine models and therefore we use the test cases generated by TzuYu as their input for a fair comparison. The number of generated test cases for each class is shown in the #TC column of Table 3.1. Neither ADABU nor Daikon is able to learn models for all of the classes. For instance, neither mined models for the java.io.PipedOutputStream class. ADABU often generates multiple (e.g.,
dozens of) models for one class, which means ADABU’s state abstraction techniques failed to generate a good invariant. The reason is that ADABU employs a set of pre-defined templates to generate invariants. If a mined state invariant contains irrelevant variables, ADABU’s state abstraction and model merging technique fails and therefore no unified model is generated. Daikon failed to mine models for `java.util.Stack` class. Both ADABU and Daikon use pre-defined invariant templates. In comparison, the invariants generated by TzuYu are better because TzuYu does not rely on templates but rather uses SVM techniques to discover propositions dynamically based on the object states. Furthermore, Daikon uses only successful executions whereas TzuYu uses both successful and failed executions, thus the model learned by TzuYu is more accurate than the one generated by Daikon.

For `example.PipedOutputStream` and `example.Signature`, PSYCO [80] can learn accurate transition guards due to the fact that it encodes all path conditions in the source code and uses an SMT solver to exactly find out whether failure happens. However, PSYCO is limited by the capability of the SMT solver.

Next, we compare the execution time of each tool on mining the models briefly. The time taken by each tool to mine the models is plotted in Figure 3.10. PSYCO is
not available for running the target classes, we cannot get the time for it. Both ADABU
and Daikon need test cases while TzuYu generates the test cases, so we only include the
time consumed by SVM for TzuYu. The figure shows that TzuYu often uses less time in
generating the models. An exception is the java.io.PipedOutputStream class
for the reason mentioned above.

### 3.8.3 Limitations and Discussions

Firstly, we use feed-back directed random testing to answer queries for active learn-
ing and therefore the learned models are not guaranteed to be neither sound nor com-
plete. An SMT solver can be used to make the learned model sound and complete
(to a certain extent in accordance with the capabilities of the SMT solver). For in-
stance, the learned model for java.uti.Stack in Figure 3.6 (b) can be verified
by showing that each transition is sound and complete, e.g., the self-looping transi-
tion at state 1 labeled with \([\text{eleCount} \geq 1] \text{pop}\) can be verified by proving two Hoare
triples: \(\{\text{elementCount} \geq 1\} \text{pop}() \{\text{noerror}\}\) (executing pop with a pre-condition
\text{elementCount} \geq 1 will not lead to error) and \(\{\text{elementCount} \leq 0\} \text{pop}() \{\text{error}\}\).

Further, if the SMT solver identifies a counterexample, the counterexample can be used
to refine the model.

Secondly, because our approach is based on random testing, there is no guarantee
that a good divider can be discovered in general—though it should emerge in theory
after sufficient testing. This can be partially fixed if we can obtain “better” test cases
through different means, e.g., from real execution history of the given class, or through
more sophisticated test case generation methods like concolic testing [168] and combi-
national testing [107].

Thirdly, our approach will not terminate if the model for the class under analysis is
beyond the expressiveness of finite-state machines with linear guard conditions. If the
refiner fails to find a divider for a membership query with conflicting results (i.e., the
same sequence of events leads to failure and success), a counterexample (i.e., a path
which is predicated to fail by the stateful behavior models but succeeds in real testing
execution, or the other way round) is returned so that \(L^*\) may introduce a new state. In
the worst case, TzuYu will keep generating models with ever growing number of states (and eventually times out). This is due to the limitation of SVM that could be overcome using advanced learning techniques.

Lastly, our approach distinguishes only two states: the normal state in which no exception is thrown and the exceptional state where exceptions are thrown. This is because we only use the criterion whether exception is thrown to distinguish the outcome of testing. Thus our approach works better for defensive programs. In practice, the programmers of the library often use defensive programming to make sure that the library works in an environment as it is supposed to.

Although in the experiment section we only learn behavior models for single classes in Java, the approach is applicable to single classes (or equivalent constructs) in object-oriented program languages such as C#. The only used third party library LIBSVM also provides C# interfaces and C# also supports reflection which is used to instrument the target class. Thus the proposed approach and its implementation can be easily adoptable to other object-oriented programs such as C# programs.

### 3.9 Summary

In this chapter, we propose a fully automated stateful behavior models learning approach from the source code of a single class in object-oriented programs. To fully automate the generation of test cases which are the required inputs for many automata learning tools, we combine the active learning algorithm $L^*$ with a random argument generation technique. We then use a supervised machine learning algorithm (i.e., the SVM algorithm) to abstract data into propositions. The contribution of our approach is that both guard conditions and the behavior models are learned automatically from the testing results.

The idea of using testing as the teacher for $L^*$ algorithm is also found in the AMC approach [83] which uses $L^*$ to handle counterexamples returned by the model
checker. The $L^*$ algorithm is also used for learning assumptions in compositional verification [21, 29, 97, 115] in formal methods community. TzuYu differs from these work in that it uses $L^*$ algorithm to learn the model from source code.

Our testing strategy is related to Randoop [147]. We extend Randoop to the context of learning in which the receiver object must be the same in order to learn a better model and we also add a new source for reference arguments which can be chosen from out-reference variables to improve data coverage. Tester in TzuYu is also related to TAUTOKO [65] which generates more test cases by mutating existing traces in the mined model (by using ADABU) to augment the model learning process as well as finding bugs.

ADABU [66] can be classified as a passive learner which requires a set of test cases as input; it abstracts the concrete states with simple templates to abstract states thus to get the abstract traces and then it merges models from abstract traces to generate a model. The combination of an active learning algorithm with automatic argument generation techniques enables TzuYu to learn stateful behavior models automatically.

The refiner in TzuYu is inspired by Sharma et al. [170] who use SVM and SMT solver to generate interpolants for counterexamples produced by model checkers. The goal of the refiner is in line with that of the dynamic invariant generator Daikon [75] and Axiom Meister [179]. Daikon uses a set of pre-defined invariant templates over data from the set of given runtime traces. Daikon may find some irrelevant invariants at a program point. Axiom Meister uses symbolic execution to collect all the path conditions which are then abstracted into preconditions. TzuYu’s refiner is based on SVM which enables TzuYu to find relevant linear arithmetic propositions over a large number of variables.
Chapter 4

Learning Sound and Complete Stateful Models

The efficiency of the approach in last chapter is due to the simultaneous learning of DFAs and guard conditions based on program testing. One of the major disadvantages of testing is the low coverage problem. This problem is essentially described by Dijkstra’s well-known comment in his 1972 Turing award lecture as “Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.”. In certain scenarios only complete behaviors models are of practical use. In this chapter we propose to learn sound and complete behaviors models for program libraries such that the learned models are useful in these scenarios such as model checking. The main contribution of this chapter is that symbolic execution is employed but in a restricted way to help verify and refine the partial behavior models which can be learned with the approach in last chapter. Although symbolic execution is much more expensive than testing. We demonstrate that the new approach can learn complete behavior models by not suffering too much performance overhead.
4.1 Introduction

In this chapter, we propose an automatic approach called TLV which combines testing, learning, and validation to generate an abstraction of a given class. The abstraction characterizes behaviors of any object of the class. In a way, TLV is designed to mimic programmers so as to combat the complexity of program analysis and verification. When experienced programmers are asked to analyze a given program, they often execute the program with various inputs, from which (among other artifacts like documentations, program comments, and domain knowledge) they would form some initial idea on what the program does (and how) and then validate their guess with more test cases or through code review. They may guess a number of times until they build a correct abstraction (in the mind) on what the program does. Depending on their objective, they would stop the process once the abstraction allows them to accomplish their analysis goal.

The architecture of TLV is inspired by the above process, as shown in Figure 4.1. The inputs are the source code of a program and optionally an artifact which TLV could use to determine the proper level of abstraction. TLV has three phases: learning, validation and refinement. In the learning phase, we apply automatic testing techniques to generate, inexpensively, sample behavior of the class, which consists of sequences of method calls. The hope is that the test cases would cover a large portion of the complete behavior. Furthermore, we adopt techniques from the machine learning community and design a learning algorithm based on the $L^*$ algorithm [24] (cf. Section 2.2.1.1) to not only guide the test case generation but also generate candidate abstractions systematically based on the testing results. In the validation phase, we apply more heavy-weight techniques like symbolic execution to validate the abstraction so that the abstraction is
guaranteed to be correct and accurate. After validation, the abstraction is checked to see whether it is at a proper level of abstraction. If it is too abstract, we refine the abstraction and restart from the testing phase. The iterative process ends when a correct and accurate abstraction is constructed.

However, a correct abstraction could be completely trivial and thus useless. In order to make sure the abstraction is useful, we need to answer two questions. The first question is: what is the right model for the abstraction? The answer decides what kind of behaviors the abstraction is capable of capturing, which in turn defines what purposes the abstraction could serve. One form of program abstraction is predicate abstraction [28] which is particularly useful for analyzing programs with non-trivial data states. Given a program and a set of predicates, predicate abstraction constructs an abstraction of the program by focusing only on the truth values of the set of predicates. In our setting, predicate abstraction means to construct an abstraction of the class in the form of a labeled Kripke structure [49], i.e., a finite state automaton whose transitions are labeled with method names and whose states are labeled with predicates. An example is shown in Figure 4.5(b). Compared with other models like finite state automata, this model is more expressive (for instance, using a predicate on the number of elements in a stack, it can express languages like the number of \texttt{pop} operations must be less than or equal to the number of \texttt{push} operations) and more catered for classes with rich data states. Furthermore, such models can be readily fed into a model checker for verification.

The underlying idea of TLV is to use testing to discover as many behaviors of the target class as possible. We first apply testing techniques with the hope to discover a large part of the behavior inexpensively. However, simply relying on random testing is limited (e.g., for predicate coverage [184]) and thus active learning techniques are adopted to not only guide the testing process but also to construct concise candidate abstractions automatically. Only when a likely abstraction has been obtained, program validation techniques are used to validate the abstraction. Furthermore, through learning, we are able to automatically discover predicates which can be used to refine the abstraction. The idea of learning from traces of a program is not new [39, 80, 91, 191]. Neither is the idea of verifying the learned model against programs [25, 197]. Rather, TLV combines a number of techniques for effective abstraction.
4.2 Approach Overview

In this section, we illustrate how TLV works using a simple example. Case studies on more realistic programs can be found in Section 4.7. The only input to TLV is the bounded stack class shown in Figure 4.2. For simplicity, we focus on two methods: push and pop. Recall that we need a usage context in order to determine the right level of abstraction. For now, assume that the abstraction is to be used for human comprehension and the user chooses not to provide any predicate initially. Based on
The assumption above, the initial set of predicates is \( \{\top, \bot\} \) where \( \bot \) is a special default predicate which denotes whether a failure (i.e., assertion violation or un-handled exception) has occurred and \( \top \) denotes no failure.

**The Learning Phase** In this phase, TLV applies a learning algorithm similar to the \( L^* \) algorithm [24, 163] to learn a candidate abstraction, relying on automatic testing techniques [147]. TLV drives the learning process by generating two kinds of queries (both of which are slightly different from those in the \( L^* \) algorithm). One is membership queries, i.e., whether a sequence of method calls would result in a particular abstract state. The other is candidate queries, i.e., whether a candidate abstraction is correct and accurate (formally defined in Section 4.3). The queries and testing results are summarized in an observation table, as shown in Figure 4.3 (a) where \( \langle \rangle \) is an empty sequence of method calls; \( \langle \text{pop}, \text{push} \rangle \) denotes the sequence of calling \( \text{push} \) after \( \text{pop} \). The result column shows the abstract state after the corresponding method calls. For instance, after an empty sequence of method calls, \( \top \) is true and calling \( \text{pop} \) right after initialization results in exception, i.e., \( \bot \). Notice that because methods may take parameters, the same sequence of method calls may result in different abstract states, as we shall see later. Based on the observation table, TLV generates the first candidate abstraction, as presented in Figure 4.3 (b).
Next, TLV asks a candidate query: is the abstraction in Figure 4.3 (b) correct? To answer the candidate query, TLV performs random walking, i.e., randomly generates a set of test cases which correspond to traces in the abstraction. Through the random walking, one inconsistency between the abstraction and the class under analysis is identified. That is, the abstraction predicts that calling \texttt{pop} from state \( \top \) always results in \( \bot \), whereas it is not the case. For instance, calling method \texttt{push} first and then \texttt{pop} results in no failure. The inconsistency suggests that the abstraction must be modified. In this case, the observation table is updated, as shown in Figure 4.3 (c), which includes the sequence \( \langle \texttt{push}, \texttt{pop} \rangle \) and its testing result. After more membership queries, TLV constructs the candidate abstraction shown in Figure 4.3 (d). The answer to the candidate query is positive and thus the learning phase terminates.

\textbf{The Validation Phase} The candidate abstraction may not be correct due to limitations of random testing. For instance, the abstraction in Figure 4.3 (d) is not correct as invoking method \texttt{push} at state \( \top \) may result in \( \bot \) when the size of the stack equals \texttt{MAX\_SIZE}. This behavior is missing because there is no test case which invokes \texttt{push} more than 1024 times. In general, cases like this are hard to generate through random testing. Thus, the learned abstraction must be validated and refined if necessary. For the candidate abstraction shown in Figure 4.3 (d), two proof obligations are generated. One is \( \{ \top \} \texttt{push} \{ \top \} \) (a Hoare triple), which denotes that invoking \texttt{push} when there is no failure always results in no failure. The other is \( \{ \top \} \texttt{pop} \{ \top \lor \bot \} \), i.e., invoking \texttt{pop} at \( \top \) may or may not result in failure. We adopt the assertion checking feature in Symbolic Execution (\textit{cf.} Section 2.2.1.2), in particular, Symbolic PathFinder (SPF) \cite{156} to discharge proof obligations. First, TLV modifies the \texttt{push} method by enclosing its method body with a try block, adds \texttt{assert(false)} to the catch block (i.e., to assert that there is no failure), adds \texttt{assert(true)} to the finally-block (i.e., to assert the post-condition), and adds the pre-condition to an if-conditional. The modified \texttt{push} method is shown in Figure 4.4. Then TLV symbolically executes the modified \texttt{push} with both parameters \texttt{element} and \texttt{size} as symbolic inputs. An assertion violation is found with concrete values for the symbolic inputs: \( \texttt{element} = 3 \) and \( \texttt{size} = 1024 \). Using the concrete values as parameters for \texttt{push}, TLV constructs a test case and executes \texttt{push}, which results in an exception (i.e., \( \bot \)). Thus a transition from state \( \top \) to \( \bot \) is added to the abstraction.
```java
public void push(int element, int size) {
    if (true) {//true encodes the pre-condition
        try {
            if (size >= MAX_SIZE) {
                throw new Exception("Full Stack");
            }
            elements[size] = element;
            size++;
        } catch (Exception e) {
            assert(false);
        } finally { assert(true); }//post-condition
    }
}
```

**Figure 4.4:** Modified `push` method

After the proof obligation \{⊤\}\(pop\{⊤ ∨ ⊥\}\) is also discharged, the abstraction shown in Figure 4.3 (e) is guaranteed to be correct and accurate (see the proof in Section 4.3).

Learning a candidate abstraction helps to reduce the proving effort. If a naïve approach was used to abstract the class, we would need to check satisfiability of every combination \(\phi ∧ m ∧ δ\), i.e., whether invoking \(m\) with \(\phi\) results in a state satisfying \(δ\), where \(\phi\) and \(δ\) are constraints which can be formed using conjunction of the given predicates or their negations and \(m\) is a method. The number of such combinations is exponential to the number of predicates.

**The Refinement Phase** A nondeterministic abstraction like Figure 4.3(e) might be confusing if it is intended for humans. For instance, what does it mean to say that calling `pop` may or may not lead to failure? To resolve non-deterministic transitions on `pop`, TLV can be instructed to identify predicates which would explain, for instance, when exactly calling `pop` leads to failure. The standard approach (e.g., as in [58]) is to partition the state \(⊤\) based on \(wp(pop, \bot)\), i.e., the weakest precondition of \(pop\) resulting in exception. Computing weakest precondition is often expensive. Instead, TLV applies machine learning techniques, e.g., Supporting Vector Machines (SVMs) [166] (cf. Section 2.2.3.1), to identify a new predicate. In particular, TLV gathers two groups of object states based on the test cases at state \(⊤\). One group contains stack objects which would result in state \(\bot\) after invoking `pop`. The other group contains those which would result
Chapter 4. Learning Sound and Complete Stateful Models

### Chapter 4. Learning Sound and Complete Stateful Models

The level of abstraction can be determined for different usage contexts. For instance, if a temporal logic property is present (i.e., to be verified), TLV would generate and refine the abstraction based on interactions with the model checker. For instance, assume the property is $G(\text{push} \land X\text{pop} \Rightarrow X(\text{size} \geq 0))$ (written in state/event linear temporal logic [49]), i.e., after push and pop, size $\geq 0$ should be always true. The initial set of predicates is set to be $\{\text{size} \geq 0, \bot\}$, i.e., all predicates in the property plus the default one $\bot$. Through the learning and validation phase, we obtain the abstraction shown in Figure 4.5(a). Through model checking (taking the abstraction as a labeled Kripke structure [49]), we found a spurious counterexample: $\langle \text{size} \geq 0, \text{push, size} \geq 0, \text{pop, } \bot \rangle$, which is a run of the abstraction. To remove this spurious counterexample, again the standard approach is to partition the state $p$ based on $\wp(\text{pop}, \bot)$. TLV rather applies SVM to identify a new predicate for differentiating states from which invoking pop results in state $\bot$ from states resulting in $\text{size} \geq 0$. With the generated predicate $\text{size} > 0$, TLV generates a new abstraction shown in Figure 4.5(b). We remark that the spurious counterexample above is ruled out by the new abstraction. Model checking the abstraction against $G(\text{push} \land X\text{pop} \Rightarrow X(\text{size} \geq 0))$ is successful and thus this abstraction serves as a proof of the property at an abstraction

![Figure 4.5](image.png)

**Figure 4.5:** The refined abstraction, where $p_1$ stands for $\text{size} \geq 0$, $p_2$ stands for $\text{size} = 0$ and $p_3$ stands for $\text{size} > 0$
level which is more abstract than the code. In this section, we present the details on how TLV generates an abstraction. We start with defining the problem.

### 4.3 Problem Definition

We assume a class $C$ contains a finite set of instance variables $V$ and a finite set of methods $M$, each of which may update variables in $V$. The semantics of $C$ is a labeled transition system $(S_c, s_c, M, T_c)$ where $S_c$ is a set of states, each of which is a valuation of all variables in $V$; $s_c \in S_c$ is the initial state; $T_c : S_c \times M \times S_c$ is the transition relation such that $(s, m, s') \in T_c$ iff, given the variable valuation $s$, executing method $m$ may result in variable valuation $s'$. A run of the labeled transition system (a.k.a. a test of $C$) is a finite sequence of alternating states and transitions $\langle s_0, m_0, s_1, m_1, \cdots, m_k, s_{k+1} \rangle$ such that $s_0 = s_c$ and $(s_i, m_i, s_{i+1}) \in T_c$ for all $i \geq 0$. The sequence of method calls in the run $\langle m_0, m_1, \cdots, m_k \rangle$ is called a trace.

The problem is to construct an abstraction of $C$ automatically. Let $Prop$ be a set of propositions constituted by variables in $V$. We write $2^{Prop}$ to denote the set of all propositions in $Prop$ and the negation of the rest. For instance, if $Prop = \{ p, q \}$, $2^{Prop}$ is $\{ p \land q, p \land \bar{q}, \bar{p} \land q, \bar{p} \land \bar{q} \}$. We write the powerset of $2^{Prop}$ as $\wp(2^{Prop})$, i.e., the set of all subsets of $2^{Prop}$. A member of $\wp(2^{Prop})$ can be represented succinctly. For instance, the set $\{ p \land \bar{q}, p \land q \}$ can be represented as $\{ p \}$, i.e., their disjunction. We write $s \models \phi$ to denote that $\phi$ evaluates to true given the variable valuation $s$. Given a set of concrete states $X$, we write $abs_{Prop}(X)$ to denote the disjunction of all members $\phi$ of $2^{Prop}$ such that $s \models \phi$ for some $s \in X$. For instance, if $Prop = \{ \text{size} \geq 0, \text{size} \geq 1024 \}$, $abs_{Prop}(\{ \text{size} \mapsto 5, \text{size} \mapsto 1034 \})$ is $\text{size} \geq 0$.

An abstraction of $C$ w.r.t. $Prop$, denoted as $A$, is a labeled transition system $(S_a, s_a, M, T_a)$ where $S_a \subseteq \wp(2^{Prop}) \cup \{ \bot \}$ is a set of abstract states, each of which is a subset of $2^{Prop}$ or $\{ \bot \}$ (a special state denoting exception); $s_a \in S_a$ satisfies $s_c \models s_a$; $T_a \subseteq S_a \times M \times S_a$ is an abstract transition relation. The abstraction is correct if there
exists \((s, m, s') \in T_c\) such that \(s \models \phi\) and \(s' \models \phi'\) imply \((\phi, m, \phi') \in T_a\). The abstraction is accurate if each \((\phi, m, \phi') \in T_a\) implies there exists \((s, m, s') \in T_c\) such that \(s \models \phi\) and \(s' \models \phi'\). However, a correct and accurate abstraction may still contain spurious runs, due to broken traces \([84]\) (i.e., an abstract transition is feasible locally but not globally). We use abstract states and predicates interchangeably hereafter.

A naïve approach to obtaining \(A\) is to check whether every possible transition \((\phi, m, \phi')\) where \(\{\phi, \phi'\} \subseteq S_a\) and \(m \in M\) is contained in \(A\). This approach is infeasible as in the worst case there are \(2^{|Prop|} \times M \times (2^{|Prop|} + 1)\) checks (as, for simplicity, we assume the behaviors after exception is un-interesting and thus there is no need to check cases where \(\phi = \bot\)), where \(|Prop|\) is the number of propositions and \(|M|\) is the number of methods. Thus, we propose the process shown in Figure 4.1 to learn \(A\). In the following, we explain the algorithms in details. In particular, testing and learning are combined in Algorithm 3 and validation is presented in Algorithm 4. Both algorithms refer to two global variables. One is an observation table \(obs\) (for storing abstract testing results) and the other is a table \(visited\) (for storing validation results).

### 4.4 Testing and Learning

In this phase, we apply a learning algorithm to not only guide the testing process but also generate candidate abstractions automatically. TLV starts with a testing and learning phase to obtain a candidate abstraction inexpensively. In this phase, TLV can be viewed as a ‘game’ between two players. One is a learner who, in order to learn, asks a series of membership queries and candidate queries. A member query asks which abstract states can be reached after a trace. For instance, in the stack example, a membership query would be: \(\langle \text{push}, \text{pop}\rangle\). After multiple membership queries, the learner makes a guess on what the abstraction is by generalizing what it has learned so far and asks a candidate query. A candidate query asks whether a candidate abstraction is correct and accurate. The other player is a teacher. The teacher’s job is to answer both kinds of queries. Ideally, a teacher would answer a membership query with all abstract states that can be reached with the given trace. The teacher answers positively to a candidate query iff the candidate abstraction is correct and accurate; if the answer


Algorithm 3: The learning algorithm

**input**: a program and a set of propositions $Prop$

**output**: an abstraction

1. let $obs$ be an empty observation table; $visited = \emptyset$
2. while true do
3.     while $obs$ is not closed and the time is not up do
4.         let trace $tr$ s.t. $T(tr) \neq T(tr') \forall$ prefix $tr'$ of $tr$
5.         for $m \in M$ do
6.             generate a membership query $tr \cdot \langle m \rangle$
7.             let $X := \text{Randoo}(tr \cdot \langle m \rangle)$
8.             $obs := obs + (tr \cdot \langle m \rangle \mapsto \text{abs}_{prop}(X))$
9.         generate a candidate query $A$ from $obs$
10.        apply random walking to check $A$
11.        if no inconsistency found then
12.            if Algorithm 2($A$, $obs$, $visited$) returns true then
13.                return $A$
14.            else
15.                let $(tr, s)$ be a counterexample to the candidate $A$
16.                $obs := obs + (tr \mapsto \text{abs}_{prop}({s}))$

To a candidate query is negative, the teacher should provide a counterexample in the form of a concrete test case, which shows the candidate abstraction is problematic. In practice, having a perfect teacher is expensive. For instance, answering a membership query would require checking whether it is feasible to satisfy any proposition in $Prop$ after a sequence of method calls, which is a non-trivial reachability analysis problem. Even worse, answering a candidate query would require solving the abstraction problem itself. Thus, TLV employs a tester (i.e., an imperfect teacher) to answer the queries.

TLV’s algorithm is presented as Algorithm 3. The inputs are a program and a set of propositions $Prop$ and the output is an abstraction. TLV maintains two data structures. One is an observation table $obs$ for storing (abstract) testing results and the other is a set $visited$ for storing validation results. The observation table $obs$ is a tuple $(P, E, T)$ where $P \subseteq M^*$ is a set of traces; $E \subseteq S_a$ is a set of abstract states; $T : P \rightarrow E$ is a mapping function such that $T(tr) = \phi$ indicates that after the trace $tr$, the abstract state $\phi$ can be reached. Initially, $P$, $E$, $T$, and $visited$ are all empty (line 1). We write $obs := obs + (tr \mapsto \phi)$ to denote the operation of adding the mapping $tr \mapsto \phi$ into the table, i.e., replacing $P$ with $P \cup \{tr\}$; replacing $E$ with $E \cup \{\phi\}$; $T$ is updated
with $T(tr) := \phi$ if $tr$ was not in the domain of $T$; otherwise, $T(tr) := T(tr) \lor \phi$. Intuitively, the latter states that if we know that after $tr$, we can reach an abstract state $T(tr)$, with the new mapping $tr \mapsto \phi$, we now know that after $tr$, we can reach either $T(tr)$ or $\phi$.

Within a certain time limit, TLV tries to make the observation table closed by asking multiple membership queries and adding mappings into $obs$ (line 3–8). Note that the concept of consistency in the $L*$ algorithm is irrelevant in our setting. An observation table is closed if the set $P$ is prefix-closed and for all $tr \in P$ such that $tr$ is not a prefix of some other trace in $P$ (i.e., $tr$ is maximum), there always exists a prefix of $tr$ say $tr' \in P$ such that $T(tr') = T(tr)$. Intuitively, the latter means that $tr$ can be represented by its prefix; therefore, TLV does not need to test further. Since there are only finitely many abstract states, $tr$ would eventually visit a state which is visited by its prefix. We remark that this definition is justified because our goal is to discover as many abstract states and transitions as possible. If the observation table is not closed, there must be a trace $tr$ such that $T(tr)$ is not equivalent to $T(tr')$ for every prefix $tr'$ of $tr$. In such a case, a membership query (i.e., $tr \cdot \langle m \rangle$) is generated for each method (line 6). In order to answer the query inexpensively, TLV generates multiple test cases using random testing (line 7). Function $\text{Randoop}(tr)$ is similar to the Randoop algorithm [147]. Given a membership query $tr$, TLV generates multiple test cases calling the methods in the query one-by-one (from the initial concrete state). In general, the methods would have multiple parameters and TLV generates arguments for every method call. Given a typed parameter, TLV randomly generates a value from a pool of type-compatible values. This pool composes of a set of pre-defined values (e.g., a random integer for an integer type, $null$ or an object with the default object state for a user-defined class) and type-compatible objects that have been generated during the testing process. In order to re-create the same object, we store the test case which produces the object. The details about test case generation can be found in Section 3.4 of Chapter 3.

After generating and executing multiple test cases according to $tr \cdot \langle m \rangle$, TLV collects the concrete data states reached by the test cases (say $X$) and updates the observation table with the mapping $T(tr \cdot \langle m \rangle) = abs_{prop}(X)$ (line 8). Ideally, after multiple membership queries, once the observation table $(P, E, T)$ is closed, TLV constructs a
candidate abstraction \( \mathcal{A} = (S_a, s_a, M, T_a) \) such that \( S_a = E; \) \( s_a \) is the state corresponding to the empty trace \( T(\langle \rangle) \); \( (\phi, m, \phi') \in T_a \) if there exists \( tr \in P \) and \( m \in M \) such that \( T(tr) = \phi \) and \( T(tr \cdot \langle m \rangle) = \phi' \). In practice, with many methods in the class, it might take a long time before the observation table is closed. Nonetheless, with the validation phase, we can construct the candidate abstraction even if the observation table is not closed. In fact, the goal is to discover every abstract behavior of the class and it is guaranteed that every behavior is discovered either during testing or validation. Thus, if closing the observation table takes a long time, TLV times out and constructs \( \mathcal{A} \) based on \( obs \).

Once the observation table is closed or the testing or learning process timeouts, TLV raises a candidate query on whether \( \mathcal{A} \) is correct and accurate with respect to \( Prop \) (line 9). TLV then employs a slightly different testing technique to answer candidate queries. We associate each abstract state \( \phi \) in \( \mathcal{A} \) with a set of concrete states which have been generated through testing so far and satisfy \( \phi \). Based on these concrete states, TLV uses random walking to construct test cases from each abstract state in \( \mathcal{A} \) to further explore behaviors of \( C \) (line 10). The testing result is then compared with \( \mathcal{A} \) to see whether they are consistent. \( \mathcal{A} \) is consistent with the testing result iff for any sequence of method calls \( tr' \) from a concrete state (associated with an abstract state \( \phi \)), the resultant concrete states \( X \) are consistent with the corresponding abstract state \( \phi' \) reached by the same sequence of methods in \( \mathcal{A} \), i.e., \( abs_{Prop}(X) \) logically implies \( \phi' \). There is an inconsistency iff there exists a concrete state \( s \in X \) such that \( s \not\equiv \phi' \) (line 11). In such a case, TLV constructs a pair \((tr, s)\), where \( tr = tr_1 \cdot tr' \) and \( tr_1 \) is the shortest trace reaching \( \phi \) in \( \mathcal{A} \), as a counterexample to the candidate query (line 15), which is then used to update the observation table (line 16). For instance, assume \( Prop = \{size \geq 0\} \) and the abstract state after \( tr \) in the observation table is \( size \geq 0 \), i.e., \( T(tr) = size \geq 0 \). If after calling the methods in \( tr \) in sequence, the concrete states are \( \{size \mapsto 2, size \mapsto 3, size \mapsto 4\} \), then it is consistent. However, a testing result \( size \mapsto -2 \) would be an inconsistency and the observation table would be updated so that \( T(tr) = size \geq 0 \lor size < 0 \).

Once the observation table is updated, TLV again checks whether it is closed and raises membership queries if it is not, until the next candidate query is generated. Once
the tester answers positively to a candidate abstraction (at line 11), TLV obtains an abstraction which is “correct” modulo the limitation of random testing. Then, Algorithm 4 is invoked to validate $A$ (line 12). If it returns true, $A$ is returned (line 13); otherwise, the process repeats. The details of Algorithm 4 is presented in the subsequent subsection.

Given that the number of states in $A$ (and the size of $E$ in the observation table) is bounded by $3|\text{Prop}| + 1$, the learning algorithm is always terminating. Furthermore, we argue that $A$ may be much smaller than this bound in practice. Firstly, variables in a class are often co-related, which is equivalent to say that there are hidden class invariants. Due to those class invariants, often not every abstract state is reachable. For instance, if a hidden class invariant is $v_1 \geq v_2$ and $\text{Prop} = \{ v_1 \geq 0, v_2 \geq 0 \}$, the abstract state $v_1 < 0 \land v_2 \geq 0$ is infeasible. Because $A$ is constructed based on concrete testing results, those hidden class invariants are embedded in $A$ naturally and hence $A$ would not contain those infeasible abstract states. Secondly, as mentioned, given a set of concrete states $X$ (reached by the same trace), the abstract state constructed is $\text{absProp}(X)$, which would effectively collapse many abstract transitions into one. Furthermore, unlike the $L^*$ algorithm, TLV may learn a non-deterministic abstraction, which could be exponentially smaller than its deterministic equivalent. Nonetheless, we admit that the effectiveness of the testing technique may affect the size of the abstraction. We skip the discussion on the complexity of the algorithm as it depends on the effectiveness of the testing techniques. Rather, we show empirically in Section 4.7 that the learning phase is usually efficient and the generated candidate abstraction usually covers a large portion of the behavior of $C$.

4.5 Validation

The abstraction $A$ learned through random testing might not be correct as some behaviors of $C$ may never be tested (e.g., TLV is unlikely to generate a test case which pushes more than 1024 times and thus the transition $(\top, \text{push}, \bot)$ would be missing). However, $A$ is guaranteed to be accurate (but may not be correct).
Algorithm 4: The validation algorithm

```
input: abstraction \( \mathcal{A} = (\mathcal{S}_a, s_a, M, T_a) \); table \( \text{obs} = (P, E, T) \); set visited
output: true iff \( \mathcal{A} \) is validated

1 for \( \phi \in \mathcal{S}_a \setminus \{\bot\} \) and \( m \in M \) do
   2 if \( (\phi, m) \notin \text{visited} \) then
      3 let \( \psi = \bigvee_{\phi', (\phi, m, \phi') \in T_a} \phi' \)
      4 let \( s = \text{SPF}(\{\phi\} m \{\psi\}) \) ▷ \( s \) is the concrete state such that \( s \models \phi \)
      5 if \( s \neq \text{NULL} \) then
         6 let \( s' = \text{Execute}(s, m) \)
         7 if \( \text{absProp}(\{s'\}) \notin E \) then
            8 let \( tr = \arg \min_{tr \in P. T(tr) = \phi} |tr| \) ▷ \( |tr| \) is the length of \( tr \)
            9 \( \text{obs} = \text{obs} + (tr \cdot (m) \mapsto \text{absProp}(\{s'\})) \)
            10 return false
         else
            11 \( S_a = S_a \cup \{\text{absProp}(\{s'\})\} \)
            12 \( T_a = T_a \cup \{(\phi, m, \text{absProp}(\{s'\}))\} \)
      else
         14 visited = visited \cup \{(\phi, m)\}
   15 return true
```

Lemma 4.1. Algorithm 3 returns an accurate abstraction \( \mathcal{A} \).

Proof (sketch): To prove that \( \mathcal{A} \) is accurate, we need to prove that for every transition \( (\phi, m, \phi') \) in \( \mathcal{A} \), there exists a concrete state \( s \) such that \( s \models \phi \) and invoking \( m \) at \( s \) would result in a concrete state \( s' \) such that \( s' \models \phi' \). This is guaranteed by line 8 and 16 in Algorithm 3 which adds a mapping into the observation table such that if \( T(tr) = \phi \) and \( T(tr \cdot (m)) = \phi' \), then there must be a concrete transition from a state satisfying \( \phi \) to a state satisfying \( \phi' \) through invoking \( m \), in both cases. Afterwards, we can prove the lemma based on the construction of \( \mathcal{A} \). □

The lemma above states that every transition in \( \mathcal{A} \) corresponds to at least one concrete transition. Next, TLV checks if there are missing transitions and if there is none, \( \mathcal{A} \) is guaranteed to be an over-approximation at the same time. In the following, we illustrate how the validation algorithm (Algorithm 4) works.

The inputs are the observation table \( \text{obs} \) and the corresponding abstraction \( \mathcal{A} \) as well as the set \( \text{visited} \) which contains pairs of the form \( (\phi, m) \) where \( \phi \) is an abstract state and
if (φ) {
  try { body of method \textit{m}; }
  catch (Exception e) {
    assert false if exception is not in \textit{ψ};
  }
  finally { assert(\textit{ψ}); }
}

\textbf{Figure 4.6:} The instrumentation of original code for symbolic proof

\textit{m} is a method name. The set \textit{visited} stores the successfully discharged proof obligations so far. Every time the algorithm is invoked, for every pair \((φ, m)\) of abstract states (exclusive of \(⊥\)) and methods, TLV checks whether it is in \textit{visited} (line 2). Intuitively, it is in \textit{visited} iff TLV has obtained all abstract states which are reachable from \(φ\) by invoking \(m\). If it is not in \textit{visited}, TLV generates a proof obligation \{\(φ\}\} \textit{m}\{\textit{ψ}\} where \(ψ\) is the disjunction of all abstract states which are reachable from \(φ\) through \(m\) in \(A\) (line 3). The proof obligation is discharged using symbolic execution, i.e., with the help of Symbolic PathFinder (SPF [156]), as explained in the following.

In a nutshell, given a Java program, SPF executes the code symbolically so as to see whether there is an assertion violation. If an assertion violation is possible, SPF generates a counterexample, which consists of the valuation of input variables and a path condition that lead to the assertion violation. We refer interested readers to work [156] for details on SPF. We instead present how the proof obligation is encoded as an assertion violation checking problem. The first step of the encoding is to syntactically transform the method \(m\) such that all relevant instance variables become parameters of the method. Next, TLV instruments the modified method with the required pre-condition \(φ\) and post-condition \(ψ\). Figure 4.6 illustrates how the instrumentation is done systematically.

TLV first encloses the original method body with a try-catch-finally block to catch all exceptions. The try block contains the method body of \(m\). If \(⊥\) logically implies \(ψ\) (i.e., \(A\) suggests that exception might be the result when we invoke method \(m\) with pre-condition \(φ\)), the try block contains no assertion; otherwise, it contains the assertion \texttt{assert(false)}. Thus, if an exception is not supposed to occur, then the occurrence of an exception would lead to an assertion failure. The finally block contains the assertion \texttt{assert(ψ)} which asserts the post-condition. Next, TLV encloses the try-catch-finally
block with an if-conditional block. The condition is set to be the pre-condition $\phi$ so that SPF checks only symbolic inputs which satisfy the pre-condition. The modified program is then fed to SPF for assertion violation checking.

If no assertion violation is found, the pair $(\phi, m)$ is added into \textit{visited} (line 12). Otherwise, using the information returned by SPF, TLV constructs a test case which starts from a concrete state satisfying $\phi$ and results in a concrete state violating $\psi$ (line 5). Note that in the actual implementation SPF is configured to generate multiple counterexamples at once to reduce the number of SPF invocations. For the stack example, when SPF is used to prove $\{\text{size} \geq 0\} \text{push}\{\text{size} \geq 0\}$, a counterexample is generated which allows TLV to construct a concrete state with $\text{element} = 3$ and $\text{size} = 1024$. Invoking method \textit{push} at this concrete state results in state $\bot$ which violates $\text{size} \geq 0$.

If $\text{abs} \text{Prop}(\{s'\})$ is in $S_a$ (not a newly discovered abstract state), at line 10, TLV adds a new transition from $\phi$ to $\text{abs} \text{Prop}(\{s'\})$. If the abstract state $\text{abs} \text{Prop}(\{s'\})$ was unreachable previously, at line 7, TLV updates the observation table with a new mapping: $tr \cdot \langle m \rangle \mapsto \text{abs} \text{Prop}(\{s'\})$ where $tr$ (i.e., the shortest trace which reaches $\phi$) is a representative of all traces reaching $\phi$. With the new abstract state, the observation table is no longer closed and therefore Algorithm 4 returns false (line 8) and TLV will execute the learning algorithm again to obtain another candidate. The idea is that we always first rely on testing to discover some of the states and transitions inexpensively. \textit{Note that executing the learning algorithm again does not invalidate Lemma 3.1 as we show in the following that $A$ remains accurate during the validation algorithm.} The validation algorithm returns true when every pair $(\phi, m)$ is in \textit{visited} (line 13). The following theorem establishes the correctness of TLV.

**Theorem 4.2.** When the validation algorithm (Algorithm 4) terminates, $A$ is a correct and accurate abstraction of $C$.

\textit{Proof (sketch):} According to Lemma 3.1, $A$ is accurate before the validation algorithm starts, i.e., for every abstract transition $(\phi, m, \phi')$ in $A$, there is a concrete transition $(s, m, s')$ such that $s \models \phi$ and $s' \models \phi'$. We need to prove that (1) during the validation algorithm, an abstract transition $(\phi, m, \phi')$ is added to $A$ if there is a concrete transition $(s, m, s')$ such that $s \models \phi$ and $s' \models \phi'$; (2) if there is a concrete transition $(s, m, s')$ such
that \( s \models \phi \) and \( s' \models \phi' \), the abstract transition \((\phi, m, \phi')\) is in \( A \). (1) is true because new transitions are only introduced at line 10 and (indirectly) at line 7 in Algorithm 4. In both cases, (1) is true as \( s \) is obtained from line 5 with a concrete transition. (2) can be proved by contradiction. Assume \((s, m, s')\) is a concrete transition such that \( s \models \phi \) and \( s' \models \phi' \) and \((\phi, m, \phi')\) is not a transition in \( A \). Then there is a proof obligation \( \{\phi\} m \{\psi\} \) such that \( \phi' \) does not imply \( \psi \) generated at line 3. Assume that SPF works correctly, then the proof must fail, which contradicts the fact all proof obligations must be discharged before the validation algorithm terminates. Thus, we conclude the above theorem is correct. □

**Complexity** Assume that proving with SPF is terminating, the validation algorithm always terminates. The number of proof obligations is determined by the number of abstract states in \( A \). In the worst case, it is exponential in the number of propositions in \( Prop \) and the number of propositions is fixed. In practice, it is often much less as we show empirically in Section 4.7. The transitions in \( A \) are discovered through either testing or symbolic execution. The more testing discovers, the less symbolic execution is needed. Because testing is more scalable than symbolic execution, thus by design, TLV minimizes symbolic execution as much as possible. Although \( A \) is correct and accurate, it does not imply that all runs in \( A \) are feasible. For instance, the run \( \langle \top, \text{push}, \top, \text{push}, \bot \rangle \) of the abstraction shown in Figure 4.3(e) is infeasible. This situation is essentially due to the phenomenon known as broken traces [84]. We use abstraction refinement to remove such infeasible runs.

### 4.6 Abstraction Refinement

There are two cases where an abstraction refinement is necessary. One is that the user requires to resolve some non-determinism in the abstraction. The other is to refine the abstraction so as to prune a particular spurious counterexample identified by a model checker. In the following, we explain the latter first and show that the two cases can be solved in the same way.
Chapter 4. *Learning Sound and Complete Stateful Models*

75

The abstraction generated after the validation phase is subject to verification techniques like model checking. Assume that the property to be verified is a safety property (e.g., a bounded LTL formula constituted by propositions on instance variables in $C$). Because the abstraction is guaranteed to be correct, if model checking $A$ concludes there is no counterexample, then the same property is satisfied by $C$. If a counterexample is identified, we need to check whether it is spurious. If it is spurious, $A$ must be refined to exclude the spurious counterexample. In the following, we show that a new predicate can be generated based on the information TLV gathered during the learning and validation process. We remark that finding the optimal refinement is known to be hard [58] and is not our goal.

Recall that by assumption, in the setting of verifying a temporal logic formula, $Prop$ contains all propositions in the formula. Let $\langle \phi_0, m_0, \phi_1, m_1, \cdots, \phi_k, m_k, \phi_{k+1} \rangle$ be the spurious counterexample, which is a finite run of $A$ (as this property is a safety property). Because this run is spurious, it must be broken at some abstract state $\phi_i$ where $i \leq k$, i.e., invoking $m_i$ at a reachable (from the concrete initial state) state satisfying $\phi_i$ never results in a state satisfying certain required constraint $\phi_{i+1}$ [84]. The idea is that if we are able to find a new predicate which could separate those concrete states (abstracted as $\phi_i$) which, after invoking $m_i$, would result in a state satisfying $\phi_{i+1}$ from those would result in a state violating $\phi_{i+1}$, then we can construct a new abstraction (with the new predicate) to rule out this spurious counterexample. For instance, in the stack example shown in Section 4.2, the spurious counterexample is: $\langle size \geq 0, push, size \geq 0, pop, \bot \rangle$. It is sufficient to rule out the run if we could find a predicate separating those concrete states associated with abstract state $size \geq 0$ into two groups: one resulting in $\bot$ after $pop$ and the other resulting in $size \geq 0$ after $pop$. Thus, the problem is to find a classifier for two sets of states, which can be solved using a machine learning based approach [170, 191]. We remark that in theory [58], this iterative process of abstraction and refinement would always terminate, assuming the program semantics is finite-state (e.g., considering an integer to be a bit-vector rather than a mathematical integer). In practice, it may run many abstraction/refinement iterations before termination.

In the case of resolving a non-determinism (as requested by the user), by definition,
we have one abstract state, at which calling the same method would result in two different abstract states. Thus, the task of resolving the non-determinism is similarly to find a classifier for two sets of states at the abstract state. In the following, we briefly explain how Support Vector Machines (SVMs) [166] is used to find the classifier.

During the process of generating the abstraction, TLV associates a set of concrete states for each abstract state, which can be partitioned into two groups accordingly. For instance, in the stack example above, one group contains stack objects with \( size \geq 1 \) (for which there is no exception after \textit{pop}) and the other contains a stack object with \( size = 0 \) (for which an exception occurs after \textit{pop}). With these two groups (say \( X \) and \( Y \)), TLV tries to identify a classifier. Formally, a classifier for \( X \) and \( Y \) is a proposition \( \omega \) such that for all \( x \in X \), \( x \) satisfies \( \omega \) and for all \( y \in Y \), and \( y \) does not satisfy \( \omega \). TLV finds the classifier automatically based on techniques developed by machine learning community, e.g., SVM (\textit{cf.} Section 2.2.3.1 as well as Section 3.5 of Chapter 3).

As long as \( X \) and \( Y \) are linearly separable, SVM is guaranteed to find a classifier (i.e., a hyperplane) separating \( X \) and \( Y \). Furthermore, there are usually more than one classifiers. In this work, TLV favors the \textit{optimal margin classifier} [170] if possible. This separating hyperplane could be seen as the strongest witness why the two groups are different.

In order to use SVM to generate classifiers, each element in \( X \) or \( Y \) must be casted into a vector of numerical types. In general, there are both numerical type (e.g., \textit{int}) and categorical type (e.g., \textit{String}) variables in Java programs. Thus, we need a systematic way of mapping arbitrary object states to numerical values so as to apply SVM techniques. Furthermore, the inverse mapping is also important to feedback the SVM results to the original program. We leverage the \textit{numerical type graph} in Chapter 3 to generate a \textit{numerical value graph} from each object type and apply SVM techniques to values associated with nodes in the graph level-by-level. We illustrate our approach using an example in the following.

Recall that one group contains stack objects with \( size = 1 \) and the other contains a stack object with \( size = 0 \). TLV first extracts two sets of feature vectors from the two groups using the first level features (i.e., features which can be accessed using the stack object and no other references) in the graph, i.e., \textit{isNull} and \textit{size}. The first set of feature
vectors is \{⟨0, 1⟩\} where ⟨0, 1⟩ denotes the stack object is not null (i.e., 0 means that \texttt{isNull} is false) and its variable \texttt{size} is of value 1. The second set is \{⟨0, 0⟩\}. Next, SVM finds a classifier \(2 \times \text{size} \geq 1\). Notice that if SVM fails to find a linear classifier based on the two sets of feature vectors, TLV constructs two new sets by using numerical values from next level in the graph (i.e., \texttt{isNull} for \texttt{elements} and \texttt{length} of \texttt{elements}, and the actual data in the array) and tries SVM again. The heuristic that we look for a classifier level-by-level is based on the belief that calling the same method leads to different results is more likely related to the values of variables directly defined in the class and less likely nested in its referenced data variables.

### 4.7 Evaluation

TLV (available at [190]) is implemented with about 35K lines of Java code. We use Eclipse JDT to analyze and instrument Java source code, e.g., for generating modified programs for symbolic execution. We use SPF [150, 156] for symbolic execution because it supports features (such as assertion checking) which are necessary in our approach. The experimental results are collected on a 64-bit Ubuntu 14.04 PC with a 3.10GHz Intel Core i3 processor and 4GB memory. For the learning phase, we generate 4 concrete values for each argument of the method in an abstract trace and each learning iteration is set to timeout in 1 minute. In the abstraction refinement component, we set the maximum depths to find the features for SVM to 5. For test case generation, we set the maximum length of generated array objects to 3 and the maximum length of automatically generated strings to 3 and we do no generate \texttt{nulls} for reference type parameter. We evaluated TLV to answer three research questions.

#### 4.7.1 Performance of TLV

The answer to this question depends not only on the capability of TLV but also on SPF. Thus, we answer the question by applying TLV to two groups of Java classes, one containing relatively simple classes which we could get useful results from SPF and the other containing real-world classes which are beyond the capability of SPF. The idea is
### Table 4.1: Statistics on TLV abstracting the classes, where N.A. stands for not available

<table>
<thead>
<tr>
<th>Class</th>
<th>LOC</th>
<th>#M</th>
<th>#Pred</th>
<th>Inputs</th>
<th>Learning</th>
<th>Initial Abs</th>
<th>Validation</th>
<th>Final Abs</th>
<th>Memory (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALTBit</td>
<td>60</td>
<td>2</td>
<td>3</td>
<td>22</td>
<td>1</td>
<td>0.4</td>
<td>5</td>
<td>11</td>
<td>7.0</td>
</tr>
<tr>
<td>BANKAccount</td>
<td>45</td>
<td>2</td>
<td>3</td>
<td>8</td>
<td>2</td>
<td>0.1</td>
<td>3</td>
<td>8</td>
<td>4.7</td>
</tr>
<tr>
<td>BOUNDEDStack</td>
<td>50</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0.2</td>
<td>3</td>
<td>3</td>
<td>2.7</td>
</tr>
<tr>
<td>FLIGHTRule</td>
<td>500</td>
<td>8</td>
<td>1</td>
<td>5209</td>
<td>1</td>
<td>60.3</td>
<td>7</td>
<td>16</td>
<td>5.0</td>
</tr>
<tr>
<td>INTMath</td>
<td>40</td>
<td>5</td>
<td>4</td>
<td>320</td>
<td>1</td>
<td>1.3</td>
<td>6</td>
<td>36</td>
<td>25.7</td>
</tr>
<tr>
<td>LISTItr</td>
<td>90</td>
<td>6</td>
<td>6</td>
<td>5485</td>
<td>1</td>
<td>60.2</td>
<td>6</td>
<td>42</td>
<td>29.3</td>
</tr>
<tr>
<td>SOCKET</td>
<td>200</td>
<td>7</td>
<td>10</td>
<td>6203</td>
<td>1</td>
<td>60.2</td>
<td>13</td>
<td>102</td>
<td>167</td>
</tr>
<tr>
<td>STREAM</td>
<td>180</td>
<td>4</td>
<td>3</td>
<td>41</td>
<td>1</td>
<td>0.6</td>
<td>3</td>
<td>9</td>
<td>7.0</td>
</tr>
<tr>
<td>SIGNATURE</td>
<td>50</td>
<td>5</td>
<td>4</td>
<td>61</td>
<td>1</td>
<td>0.4</td>
<td>4</td>
<td>15</td>
<td>13.1</td>
</tr>
<tr>
<td>eWallet</td>
<td>90</td>
<td>4</td>
<td>5</td>
<td>33</td>
<td>1</td>
<td>0.4</td>
<td>2</td>
<td>5</td>
<td>7.5</td>
</tr>
<tr>
<td>PAYApplet</td>
<td>100</td>
<td>5</td>
<td>6</td>
<td>31</td>
<td>1</td>
<td>0.3</td>
<td>2</td>
<td>5</td>
<td>29.8</td>
</tr>
<tr>
<td>SOCKETReal</td>
<td>1660</td>
<td>7</td>
<td>6</td>
<td>1807</td>
<td>1</td>
<td>60.6</td>
<td>13</td>
<td>15</td>
<td>N.A.</td>
</tr>
<tr>
<td>JAVAMailReal</td>
<td>2000</td>
<td>5</td>
<td>2</td>
<td>21</td>
<td>1</td>
<td>14.6</td>
<td>3</td>
<td>10</td>
<td>N.A.</td>
</tr>
<tr>
<td>STREAMReal</td>
<td>180</td>
<td>4</td>
<td>4</td>
<td>2725</td>
<td>1</td>
<td>60.2</td>
<td>4</td>
<td>14</td>
<td>N.A.</td>
</tr>
</tbody>
</table>
to show that TLV is able to generate correct and accurate abstraction efficiently if the symbolic execution engine is working as hoped, and TLV is able to generate meaningful abstractions (without soundness guarantee) for large programs even when SPF fails to provide any support.

The first group contains 12 Java classes. In particular, ALTBit, FlightRule, INTMath, Signature, Socket, and Stream were used in the evaluation of X-Psyco tool [91]; ServerTable and ListItR are from the work [22]; BankAccount is from the work [197] and eWallet and PayApplet are adopted from Java Card applets [93]. To determine the level of abstraction, for each class, we set the initial predicates to be those collected from the source code, e.g., conditions in “if” and “for” statements. In our experience, those predicates would often allow us to quickly get some idea of the class behavior. The set of predicates for each of the above target classes can be accessed at our website [190]. We acknowledge that these classes are relatively small because most of the above-mentioned approaches (like TLV) are limited to the capabilities of symbolic execution.

To show that TLV is useful even without the validation phase, we collect a second group of classes, containing real-world programs, and apply TLV to generate abstraction. In particular, SocketReal is the java.net.Socket class defined in JDK 7 (with >1.6K LOC in the class and >20K LOC in the referenced library); JavaMailReal is the com.sun.mail.smtp.SMTPTransport class defined in the JavaMail library (version 1.5.2, with 2K LOC in the class and >45K LOC in the referenced library). StreamReal is the JDK 7 class java.io.PipedOutputStream class (with 180 LOC in the class and >3K LOC in the referenced library). These programs are un-modified other than that we set the first two programs to connect to a local socket server and mail server for testing purpose (as did in TAUTOKO [64, 65]). They either use Java Native Interface or contain reference type fields and parameters, which are not supported by SPF. To determine the level of abstraction, we manually inspect the classes and collect predicates which we believe are associated with the class invariants. Based on the abstractions learned by TLV we confirm that those are indeed class invariants.

The experimental results for these two groups of classes are shown in Table 4.1. Column LOC shows the lines of code in the class without comments; Column #M shows
the number of methods used for abstraction; Column $\#Pred$ is the number of propositions for each class (excluding the one on whether the state is $\bot$). We collect the number of membership queries (column $\#MQ$), and the number of candidate queries (column $\#CQ$), and the total time (column $T_L$) to learn the initial abstraction. The statistics for the validation phase are the number of proof obligations (column $\#Prof$, i.e., the size of visited in the validation algorithm) and the time used in the validation phase (column $T_V$). A closer look shows that $T_L$ is dominated by the time spent on maintaining the observation table (merging abstract states/transition). On the contrary, running the test cases only takes a small portion of the time. $T_V$ includes the interprocess communication time and each invocation of SPF often takes less than one second. The instrumentation and compilation time in the validation phase are negligible. For all classes, we manually confirm the correctness and accuracy of the generated abstractions. It shows that for all classes, TLV generates the abstraction in minutes. Furthermore, the overall time is dominated mostly by the validation algorithm (for 10 out of 12 cases) for the first group of classes. For all classes, the peak memory consumption for TLV is 712 MB and thus TLV is reasonably memory efficient.

### 4.7.1.1 Comparison with Related Tools

There are two existing tools on predicate abstraction of Java programs: J2BP [149] and X-PSYCO [91]. J2BP generates abstractions for a Java program with a “main” method and not for Java classes. Furthermore, it does not support random numbers or symbolic inputs, we are unable to write a driver program so that J2BP can be used to abstract a Java class. X-PSYCO is designed for generating an interface specification. It discovers predicates (through symbolic execution) on method parameters to specify constraints which must be satisfied in order to invoke the method. X-PSYCO assumes that only propositions on method parameters and return values are relevant. This implies X-PSYCO and TLV target at completely different programs. Thus, we conclude that X-PSYCO and TLV are complementary but incomparable. In addition, there is one ongoing effort by the JPF team [67] and a previously reported tool [22], which is not available any more. Besides tools for predicate abstraction, there are tools of generating models for Java classes, among which we identify the TAUTOKO tool [64,65] to be bearing a
similar goal as TLV. Thus, in the following, we compare TLV with TAUTOKO\(^3\) in the context of answering the above research question.

TAUTOKO first uses ADABU [66] to generate an initial model for each test case in the user-provided test suite. It then mutates existing test cases to generate more tests cases from the initial model and combines models for new test cases with the initial model to generate an enriched model. For fairness, we use the test cases generated by TLV in the learning phase as the input test suite for TAUTOKO. For the predicates, TAUTOKO is limited to predicates generated with a set of abstraction templates over instance variables of the given class, while TLV is more flexible. Thus, we set the predicates used in TLV to be those used in TAUTOKO. We compare TLV and TAUTOKO by applying them to the first group of classes. Notice that TAUTOKO has trouble in obtaining models for the second group of classes for various reasons: TAUTOKO cannot handle \texttt{SocketReal} as TAUTOKO does not support Java 7; TAUTOKO does not generate models for \texttt{StreamReal} because it does not instrument classes in \texttt{java.io} package; TAUTOKO fails to generate the trace file from the test suite for \texttt{JavaMail-Real} because some test cases do not close the connection to the server and do not terminate; TLV is able to get the states of the object even if the connection does not terminate.

The statistics for the models generated by TAUTOKO and TLV are shown in Table 4.2, which shows the number of states (column \#S) and transitions (column \#T)

\begin{table}[h]
\centering
\caption{Comparing TLV with TAUTOKO on abstraction}
\begin{tabular}{|c|c|c|c|c|c|c|c|c|}
\hline
Class & TLV & & & & & TAUTOKO & & \\
& \#S & \#T & \#T\(_s\) & T\(_s\) & \#S & \#T & \#T\(_s\) & T\(_s\) \\
\hline
\texttt{AltBit} & 5 & 11 & 8 & 7.6 & 4 & 5 & 2 & 37.2 \\
\texttt{BankAccount} & 5 & 19 & 8 & 141.2 & 4 & 12 & 0 & 1817 \\
\texttt{BoundedStack} & 3 & 7 & 2 & 4.4 & 3 & 5 & 1 & 14.2 \\
\texttt{FlightRule} & 3 & 8 & 3 & 6.5 & 2 & 3 & 2 & 12.6 \\
\texttt{IntMath} & 2 & 16 & 8 & 5.9 & 1 & 6 & 0 & 6.4 \\
\texttt{ListTr} & 6 & 47 & 19 & 27.0 & 3 & 10 & 2 & 20.2 \\
\texttt{ServerTable} & 12 & 118 & 52 & 68.5 & 7 & 27 & 10 & 244.2 \\
\texttt{Socket} & 25 & 219 & 146 & 174.0 & 2 & 8 & 1 & 475.8 \\
\texttt{Stream} & 3 & 9 & 2 & 7.9 & 3 & 7 & 1 & 9.7 \\
\texttt{Signature} & 4 & 15 & 5 & 14.4 & 4 & 15 & 5 & 18.4 \\
\texttt{eWallet} & 3 & 13 & 8 & 8.2 & 2 & 2 & 1 & 2.4 \\
\texttt{PayApplet} & 6 & 29 & 20 & 15.3 & 2 & 3 & 2 & 12.5 \\
\hline
\end{tabular}
\end{table}

\(^3\)We use the version of TAUTOKO reported in [65] as the implementation reported in their later work [64] is not available.
discovered by TLV and TAUTOKO, respectively. In addition, \( #T_v \) denotes the numbers of transitions to state \( ⊥ \), which is a useful metric [65]. The results show that TLV generates more accurate models (more valid states and transitions) than TAUTOKO. The time statistics (column \( T \)) show that TLV is often more efficient than TAUTOKO, especially when the number of test cases is large.

4.7.2 Effectiveness of Testing and Learning

This question evaluates TLV’s underlying assumption, i.e., testing and learning could effectively reduce the effort on symbolic execution. For the second group of classes, the answer to the question would determine how much behavior the abstraction contains and thus how useful it is, since the symbolic execution engine is helpless for these classes.

For the first group of classes, this question is answered by measuring the percentage of abstract states/transitions which are discovered in the learning phase. In particular, we compare the initial candidate abstraction (column Initial Abs) which is generated based on testing and learning only and the final abstraction (column Final Abs). We collected the respective number of states (column \( #S \)) and number of transitions (column \( #T \)).

It can be observed that for most of the classes in the first group (11 out of 12), most of the states (96%) and transitions (94%) are discovered during learning based on the test cases, which suggests that our underlying assumption is often valid. On the other hand, there is only one class (PAYAPPLET) where testing is shown to be ineffective in discovering the behavior (only 17% of the transitions are discovered by testing), which evidences that a validation phase is indeed necessary. A closer look at the class shows that only the method “setKey” leads to a state (which is then connected to a number of other states) from the initial state. Furthermore, this transition can only happen when a particular integer parameter value is passed in (there is an “if” statement with condition \( size = (DES\_KEY\_SIZE + ID\_SIZE) \)). TLV did not generate the particular integer value for \( size \) satisfying this condition and thus missed many states. We expect this situation happens more frequently with larger and more complicated programs, which might pose a threat to TLV. On the other hand, by comparing the \( T_L \) with \( T_V \) and contrasting \( T_L \) with the number of transitions discovered during testing, it is evident
that testing discovers abstract states/ transitions much cheaper than symbolic execution and therefore it is wise to start with testing and learning.

For the second groups of classes, we compare the generated abstractions with those constructed manually using the same set of predicates. The results in Table 4.3 show that TLV can learn all the states and transitions of these classes with regard to the given set of predicates. In general, learning-guided testing is unlikely to cover all behaviors of a class. However, for these classes whose methods have no or few parameters, learning-guided testing, as implemented in TLV, is effective at discovering most of the behaviors systematically.

In the following, we present some details on the SOCKETREAL case. The abstraction generated by TLV is shown in Figure 4.7 with the 6 predicates used for abstraction.

The predicates are obtained from suspected class invariants, e.g., connected = \( T \) implies bound = \( T \) and bound = \( T \) implies created = \( T \). Thus, part of the goal of using

---

**Table 4.3: Comparing TLV’s abstraction and manual abstraction**

<table>
<thead>
<tr>
<th>Class</th>
<th>Testing and Learning</th>
<th>Manually Constructed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#S</td>
<td>#T</td>
</tr>
<tr>
<td>JAVAMailREAL</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>STREAMREAL</td>
<td>4</td>
<td>14</td>
</tr>
<tr>
<td>SOCKETREAL</td>
<td>13</td>
<td>85</td>
</tr>
</tbody>
</table>

---

**Figure 4.7:** A behavior model for java.net.Socket with the predicates: closed = \( T \), created = \( T \), bound = \( T \), shutIn = \( T \), shutOut = \( T \), connected = \( T \), where the error state \( S_1 \) and all transitions to it are omitted for brevity.
### Table 4.4: Statistics for automatic predicate generation

<table>
<thead>
<tr>
<th>Class</th>
<th>Time (s)</th>
<th>Mem (MB)</th>
<th>Predicates</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOUNDEDSTACK</td>
<td>77.5</td>
<td>361</td>
<td>$size \geq 0$, $size \geq 1024$</td>
</tr>
<tr>
<td>SIGNATURE</td>
<td>43.0</td>
<td>124</td>
<td>$state \leq 0$, $state \leq 1$, $state \leq 2$</td>
</tr>
<tr>
<td>PAYAPPLET</td>
<td>349.4</td>
<td>379</td>
<td>$state \geq 0$, $state \leq 0$, $state \leq 2$, $size \geq 16$, $size \leq 16$, $value + state &gt; 1$</td>
</tr>
</tbody>
</table>

TLV to generate the abstraction is to check whether these are indeed class invariants. The abstraction learned by TLV contains (only) 12 valid states plus the error state. In addition to those transitions shown in Figure 4.7, there is a transition from $S_0$ to $S_1$ labeled with bind. This transition occurs when the port number (e.g., 3) used for binding is reserved and thus the method call results in permission violation exception. Note that other transitions to the error state are omitted for brevity. We confirm that the abstraction is correct and accurate by manually discharging all the proof obligations. Based on the abstraction, we can easily confirm that the suspected class invariants are indeed invariants since all states in the abstraction satisfy them.

### 4.7.3 Capability of Predicates Generation

We remark that this question is best answered with specific properties to be verified and specific spurious counterexamples returned by a verification engine. We have integrated a model checker [175] into TLV, which makes TLV a fully automatic Java model checker based on abstraction and refinement (by learning new predicates).

In order to answer this question, we assume TLV is being used to verify a Java class against a precise deterministic specification on when an exception occurs in the class, i.e., a stateful behavior model or typestate [173]. Note that such a specification often involves predicates on instance variables. Thus, TLV is first used to construct an abstraction with one proposition $true$. The result is an abstraction containing two states: $true$ and $\bot$. Next, TLV refines the model by discovering new predicates which would show exactly when an exception occurs (and thus rules out spurious counterexamples.
found by verifying the abstraction against the given specification). We show that TLV eventually finds the right predicates based on testing results and learning.

The results are shown in Table 4.4 (note that not all of the classes have a nontrivial stateful behavior model). Note that for class PAYAPPLET, SVM generates two predicates $state \geq 0$ and $state \leq 0$ and uses their conjunction to obtain the same result as the one ($state = 0$) shown in the table. We remark that in this setting, TLV solves the same problem of synthesizing a stateful behavior model as that addressed by the TzuYu [191] tool. However, different from TzuYu, the stateful behavior models generated by TLV are guaranteed to be correct (as well as accurate).

### 4.7.4 Limitations and Discussions

TLV has two main limitations. First, because TLV employs symbolic execution for abstraction validation, it inherits the limitation from symbolic execution engines, e.g., limitations in handling programs with loops [94] or complex data structures [40, 101]. We believe that TLV (like previous approaches) would handle larger programs with advancements in program verification techniques. Second, because TLV relies on random testing to discover behaviors, the performance of TLV would suffer if the program contains many behaviors which are hard to explore with random testing (in this case, TLV constructs the abstraction mainly based on symbolic execution).

### 4.8 Summary

In this chapter, we propose an approach which combines testing, learning, and validation in order to automatically and efficiently generate accurate and correct stateful behavior models of single classes. The contributions are the following: (1) we develop an approach on combining testing, active learning, and validation to construct stateful behavior models with given set of Boolean formula; (2) we propose a way of generating new predicates to refine the stateful behavior models. We evaluate our approach using a number of programs, including benchmark programs as well as three real-world classes,
and show that our approach generates abstraction efficiently for program analysis and verification.

Our approach is the first to combine testing, learning, and validation for program abstraction. TLV is a generalization of the approach in last chapter which learns stateful behavior models for a single class. Both TzuYu and TLV rely on random testing and learning to generalize models from concrete tests. However, TzuYu learns only stateful behavior model which has only two states (⊤ and ⊥), whereas TLV can handle more predicates; TzuYu’s learning algorithm is a direct adoption of the \( L^* \) algorithm, whereas TLV’s learning algorithm is designed to maximize predicate-coverage [27]. More importantly, TzuYu provides neither correctness nor accuracy guarantee of the learned stateful behavior models, whereas TLV does.

As mentioned earlier, TLV bears a similar goal as the TAUTOKO tool [64, 65]. Different from TAUTOKO, TLV uses symbolic execution to discover more states and transitions and to provide correctness guarantee. Furthermore, TLV’s abstraction is catered for specific usages and has a targeted level of abstraction. Alrajeh et al. [19, 20] integrate inductive learning and model checking to elicit operational requirements from goal models. Their approach differs from TLV in their goals and underlying techniques.

Existing approaches on building finite state models [34, 66, 106, 127, 129] use similar state abstraction strategies as used in TLV. The STRAWBERRY tool [34] mines behavior protocols concerning usage of a web service. ADABU [66] generates invariants from concrete execution traces of Java classes with a set of fixed invariant templates. ReAjax [127] uses a similar way as ADABU to generate the abstract model for the Document Object Model of an Ajax web pages. KrKa et al. [106] use DAikon to generate a set of possible state invariants and then use SMT solvers to decide the feasibility of possible states (combinations of invariants) and transitions in the abstract model. Revolution [129] mines state based behavior model for systems whose behaviors may evolve with time.

TLV uses automata learning and testing to construct the initial abstraction. Similar ideas have been used to generate models for legacy systems [92, 128] and bug detection [154]. They use \( L^* \) to learn Deterministic Finite Automata (DFA) or Mealy
Machines, whereas the learning algorithm in TLV learns a Non-deterministic Finite Automaton (NFA). TLV requires the source code of the target class to generate abstractions while some techniques [34, 79] treat the System Under Test (SUT) as black-box, thus they use only the externally visible values for state abstraction. Ghezzi et al. [79] use behavior equivalent model for finite data container to generate an abstract model for a given Java class with a test suite.

TLV learns NFAs, whereas $L^*$ learns only DFA which is limited for programs with data variables. TLV is related to work on extending the $L^*$ algorithm to learn NFAs [38] or other finite state models [33, 48, 125, 169]. Bollig et al. [38] extend the $L^*$ to learn a non-deterministic Residual Finite State Automata (RFSA) for a deterministic SUT. Berg et al. [33] extend $L^*$ to learn a symbolic Mealy Machine. Shahbaz and Groz [169] introduce a direct Mealy Machine learning algorithm which handles counterexamples returned by the teacher much more efficiently. Lorenzoli et al. [125] propose the $GK$-tail algorithm to generate an Extended Finite State Automata. Cassel et al. [48] extend $L^*$ to learn symbolic register automata with tree queries.
Chapter 5

Learning Models via Passive and Active Learning

The two approaches in last two chapters take the source code of a program library as the only input to learn the behavior model using combination of testing and symbolic execution. Popular programs such as Java standard classes have many existing client programs which use them. The ways of invoking the program’s APIs in these existing client programs provide valuable information for learning behavior models. In this chapter, an approach which leverages existing client programs to learn behavior models is proposed. Besides learning from existing client programs passively, active learning is also used to generate human interpretable behavior models.

5.1 Introduction

In this chapter, we propose a scalable approach for synthesizing behavior models by combining statistical learning and active learning. The key observation behind our approach is that statistical learning and active learning can complement each other to produce human-interpretable models in a scalable and efficient way. Our approach first applies statistical learning to generate a probabilistic model from static call sequences extracted from existing client programs and then uses the probabilistic model as the teacher of “experiences” to provide guidance on actively learning a finite state machine.
(FSM) model. Due to noise in static call sequences, the experience teacher may not be confident about some call sequences. Dynamic testing is then employed to confirm the validity of such call sequences. Since such call sequences are a small portion of all call sequences, our approach is much more efficient than symbolic execution or testing based active learning techniques. The learned FSM model is easier for human interpretation than the statistical model. In some cases an FSM model is easier to implement and efficient to run than a probabilistic model, for example, in hardware based malware detection [69], it is easier to implement and more efficient to check whether a call sequence belongs to an FSM model than that of a probabilistic model.

5.2 Approach Overview

This section illustrates how HiSTAR works on the class `java.util.Stack` to learn its behavior model by following the workflow in Figure 5.1. For simplicity, we consider five main public methods in this class, i.e., `init`, `push`, `pop`, `empty`, and `size`, where `init` denotes the class constructor.

**Call Sequences Extraction** HiSTAR statically extracts all sequences of calls of these methods from many client programs. Figure 5.2 (a) shows an example client program that HiSTAR uses to extract the call sequences for `java.util.Stack`. HiSTAR first builds the Control Flow Graph (CFG) shown in Figure 5.2 (b) and extracts loop-free maximum paths from the CFG (i.e., paths which start from the entry and end with a return statement or an exception). For example, we can get two such paths: \( l_2, l_3, l_7, l_{15} \) and \( l_2, l_3, l_7, l_8, l_9, l_{10}, l_{15} \). Each path is then augmented with paths inside an inner loop in a way that if any node in the path is the header of an inner loop,
**public int clientMethod()**{
    Stack<Integer> stack = new Stack<Integer>();
    for(int i=0; i<10; i++){
        stack.push(i);
    }
    while(!stack.empty()){
        if(stack.size() < 2){
            stack.push(1);
            break;
        }
        stack.pop();
    }
    return stack.size();
}

**Figure 5.2:** A client program which uses the JDK class `java.util.Stack` (a) and its control flow graph (b)

The path is extended with the paths inside the loop right after the node. The path inside the first loop (Line 3–Line 5) is $l_4, l_3$ and the path in the second loop (Line 7–Line 13) is $l_8, l_{12}, l_7$. As a result, the extended paths are $l_2, l_3, (l_4, l_3)^*, l_7, (l_8, l_{12}, l_7)^*, l_{15}$ and $l_2, l_3, (l_4, l_3)^*, l_7, (l_8, l_{12}, l_7)^*, l_9, l_{10}, l_{15}$.

HiSTAR extracts the usage scenarios from these two paths by concatenating the method call sequences, and gets `init, (push)*, empty, (size, pop, empty)*, size and init, (push)*, empty, (size, pop, empty)*, size, push, size`. Finite method call sequences are then generated by unfolding the loops zero, one, and two times. For example, for the first usage scenario, we generate the following sequences: $(init, (push)^0, empty, (size, pop, empty)^0, size)$, i.e., $(init, empty, size)$.

**Passive Learning** We apply Hidden Markov Models (HMMs) [157] learning algorithm to learn an HMM from these finite call sequences. Figure 5.3 shows a simplified version of the learned HMM whose transitions with transition probabilities less than 0.01, and methods in each state with emission probabilities less than 0.01 are filtered out to reduce clutter. The simplified HMM still contains 14 states and 65 transitions and the learned

---

4Here we use the notation $(s)^*$ to denote zero or more occurrences of the sequence $s$. 
HMM contains 14 states and 103 transitions. Learning HMMs from call sequences has been applied in previous work [143].

HMMs are not easy for human interpretation because there are exponential number of paths (state sequences) which can generate a given call sequence due to probabilistic transitions and probabilistic emissions of methods. The other problem is that the method call sequences used for HMM learning contain both false positives and false negatives. For example, the call sequence init, empty, size (generated by repeating the both loops \((l_4, l_3)^*\) and \((l_8, l_{12}, l_7)^*\) zero times) is a false positive because in reality no programmer would call empty immediately after the constructor without pushing some elements into a stack; the call sequence init, push, empty, size, pop, empty, size, pop (generated by repeating loop \((l_4, l_3)^*\) and \((l_8, l_{12}, l_7)^*\) once and twice, respectively) is an invalid call sequence because the stack is pushed once and the second pop throws an exception. Note that this problem is pervasive for all techniques (including our approach) which train models from statically collected call sequences.
Chapter 5. Learning Models via Passive and Active Learning

**Active Learning** In order to generate interpretable models and tackle the problem in the collected call sequences, HIStar uses the $L^*$ algorithm [24, 163] and testing to learn a Deterministic Finite Automaton (DFA). The $L^*$ algorithm assumes there is a teacher and works by learning from two kinds of queries for the teacher: membership queries and candidate queries. We use the HMM as a teacher of experience to build a DFA. Furthermore, when the experience teacher is not confident enough, we apply testing to gather more information for learning. $L^*$ maintains an observation table (Figure 5.4...
(a) shows a snapshot of the table) to store the answers for the queries. The rows of an observation table are composed of base rows and extended rows (separated by a dashed line). Table contents (in white background) are indexed by row headers and column headers (gray background) each of which contains a call sequence. At first $L^*$ adds the empty sequence $\langle \rangle$ to the base row headers and the column headers. It then adds five call sequences $\langle$empty$\rangle$, $\langle$pop$\rangle$, $\langle$push$\rangle$, $\langle$size$\rangle$, and $\langle$init$\rangle$ to the extended row headers, which are the extensions of the empty call sequence in base rows with one of the methods.

Membership Query $L^*$ concatenates each row header with each column header and asks a membership query whether the concatenated sequence is valid (denoted by 1) or invalid (denoted by No). For example, for the table cell indexed by row header $\langle$push$\rangle$ and column header $\langle \rangle$, a membership query with the call sequence $\langle$push$\rangle$ is asked.

In an HMM, each call sequence is associated with a generation probability. The generation probability of the call sequence $\langle$push$\rangle$ is 0.00001. HiSTAR compares this generation probability to an upper threshold probability (e.g., 0.2) and a lower threshold probability (e.g., 0.03, refer to Section 5.4.3 for the details on how the two threshold probabilities are generated) to decide its validity. We answer 1 if the generation probability is greater than the upper threshold probability and 0 if the generation probability is less than the lower threshold probability. Intuitively, a call sequence with a high probability means that such call sequences are often obtained in client programs and thus they are highly likely valid, even if some sequences extracted from the client programs are invalid. On the other hand, if a call sequence is rarely observed in client programs, we believe it is not valid. The answer for $\langle$push$\rangle$ is 0 because its generation probability (0.00001) is less than the lower threshold probability (0.03) and $\langle$init$\rangle$ is valid because its generation probability is much larger than the upper threshold probability. If the generation probability falls between the lower threshold and upper threshold, we employ testing to resolve its validity. For example, the generation probability of $\langle$init, size$\rangle$ falls between 0.03 and 0.2 and HiSTAR executes the call sequence several times with randomly generated method parameters. If no execution of the call sequence throws exception, the call sequence is treated as valid; otherwise invalid.
In this example, the membership query answers 1 for \( \langle \text{init} \rangle \) and \( \text{No} \) for the other four. To make the table closed, \( L^* \) moves the row \( \langle \text{pop} \rangle \) to the base rows and asks queries for sequences \( \langle \text{pop,empty} \rangle \), \( \langle \text{pop,push} \rangle \), \( \langle \text{pop,init} \rangle \), and \( \langle \text{pop,init} \rangle \). The answers for these sequences are all \( \text{No} \). Now the table becomes closed, i.e., all contents of extended rows are included in the base rows, as shown in Figure 5.4 (b).

**Candidate Query** When the table is closed, \( L^* \) constructs a candidate DFA shown in Figure 5.4 (c), and asks a candidate query whether the candidate DFA is a good approximation of the HMM. We answer candidate query by generating a set of membership queries. A sequence is returned as a counterexample if its membership answer does not match its last state in the DFA. For the candidate DFA shown in Figure 5.4 (c), the sequence \( \langle \text{init,push} \rangle \) is returned as a counterexample because DFA says it is invalid (because its last state \( B \) is not an accepting state) but its membership query answers 1.

**Refinement** \( L^* \) updates its observation table with the counterexample by adding \( \langle \text{push} \rangle \) to the column headers and asking membership queries for empty cells created by the newly added column. Afterwards, the table becomes not closed because the row content 1,1 of the extended row \( \langle \text{init} \rangle \) is not included in the base rows. It then moves...
this row to the base rows and extends (init) with each method. It fills the cells for the extended sequences by asking more membership queries. Now the table as shown in Figure 5.5 (a) becomes closed again. L* constructs the second candidate DFA shown in Figure 5.5 (b) and asks the second candidate query. After several other candidate queries when no counterexample can be found, HiSTAR learns the DFA shown in Figure 5.6.

The number of call sequences which need to be executed dynamically in our approach is a small portion of all call sequences asked by L* and thus is much smaller than those in existing active learning techniques. All the call sequences in existing active learning approaches are answered through symbolic execution [22, 80], testing [191], or using both [91, 177] and L* asks $O(|\Sigma| n^2 + n \log m)$ membership queries [24, 163], where $|\Sigma|$ is the number of public methods, $m$ is the length of the longest counterexample returned by the teacher and $n$ is the number of states of the learned DFA.

The remaining sections are organized as follows: we explain the algorithms used to extract method call sequences from existing client programs in Section 5.3; Details of training HMMs from collected method call sequences are explained in Section 5.4; Section 5.5 explains how to actively learn behavior models from HMMs and testing. We apply our approach to generate a benchmark which covers the classes in JDK comprehensively and evaluate the performance of our approach in section 5.6. We discuss the related work and conclude this chapter in section 5.7.

5.3 Call Sequences Extraction

We choose the Maven repository [178] as the source of our training data, since it consists of thousands of Java client programs, and manifests abundant interface usage scenarios. We follow these three steps to collect data: (1) We crawl all JAR files with their latest versions in the Maven repository by their URLs; (2) We use Soot [183] to convert the Bytecode of each class in the downloaded jar files to the Jimple intermediate representation; (3) We use static analysis techniques to extract all call sequences of JDK classes from each method in the client programs.
A usage scenario for a target class is a sequence of method calls of public methods in this class with possible repetitions of some methods. Algorithm 5 gives the algorithm to extract usage scenarios. The basic idea behind this algorithm is to find the *simple paths inside each loop* in the bottom-up manner from the inner-most loops and then combine the simple paths in a loop with all the possible usage scenarios inside its immediate nested loops. Here, a simple path is defined as a path without repeated vertices, i.e., a simple path is a path without loops; a simple path inside a loop is a simple path which starts at one of the loop header’s successors and ends at the loop header.

First, we use static analysis [17] (page 658) to find the loops in the client method (implemented by function `findLoops`). Loops may be nested inside other loops and loops inside a client method form a loop tree [132], in which each node denotes a loop, and is immediately nested inside the loop denoted as its parent node (except the root node which does not have a parent loop). Next, we perform a simple alias analysis (implemented by the function `mustAliasAnalysis`) to precisely track the method calls on a target object instance. In particular, we apply intra-procedural analysis with local must-alias analysis [37] for the client method under analysis to find all the must-aliases for each receiver of a target method at the current statement before our call sequence analysis. To handle the case that fields are instances of the target class and are accessed in different methods in the client program, alias analysis needs to track the field accesses as well and thus requires the construction of the call graph of the client method under analysis.

A depth-first search based algorithm (function `RecursiveDFS`) is used to find all simple paths from the entry to an exit of a client method (Line 5). After finding a simple path (Line 11) we recursively extend (function `CombinePath`) the simple path with possible simple paths from its inner loops (by calling the function `FindPathsInLoop` at Line 24); if any node in a simple path is the header of an immediate inner loop (Line 23), we compute all sub-usage scenarios which correspond to the paths within the inner loop starting from the header’s successor which is also inside the loop and ending at the header (Line 24 and function `FindPathsInLoop`). Then we insert each found usage scenario inside the inner loop with the repetition annotation after current node (Line 25–Line 28). Nested loops are handled by recursively calling the function `RecursiveDFS`. 
Algorithm 5: Extracts usage scenarios in a client method

**input**: Interface set $I$ of the target class $C$, the CFG $G = (\text{entry}, \text{exits}, B)$ of a client method $M$, $B$ is the set of blocks in $M$.

**output**: A set of usage scenarios $ups$

1. let $\text{loops} \leftarrow \text{findLoops}(G)$
2. let $\text{aliases} \leftarrow \text{mustAliasAnalysis}(C)$
3. let $bb2seq \leftarrow \text{findSeqsInBlock}(G, \text{aliases}, I)$
4. let $\text{path} \leftarrow \langle \text{entry} \rangle$ \hspace{1cm} ▷ path is a stack
5. let $ups \leftarrow \text{RecursiveDFS}(\text{path}, \text{exits}, B)$
6. $\text{path.pop}()$
7. return $ups$

**Function** RecursiveDFS($path, T, N$)

1. let $ups \leftarrow \emptyset, curr \leftarrow path.\text{top}()$
2. if $curr \in T$ then
3.   $ups \leftarrow ups \cup \text{CombinePaths}(path, N)$
4. for $succ \in curr.\text{successors} \land succ \in N$ do
5.   if $succ \notin path$ then
6.     $\text{path.push}(succ)$
7.     RecursiveDFS($path, T, N$)
8. $\text{path.pop}()$
9. return $ups$

**Function** CombinePaths($path, blks$)

1. let $seq \leftarrow \langle \rangle$ \hspace{1cm} ▷ list of sets of sequences
2. for $curr \in path$ do
3.   $seq+ = \{bb2seq[curr]\}$
4.   for $inner \in \text{loops.immediateInnerLoop}(blks)$ do
5.     if $curr = inner.\text{header}$ then
6.       let $subs \leftarrow \text{FindPathsInLoop}(inner, \{curr\})$
7.       let $comps \leftarrow \emptyset$
8.       for $sub \in subs$ do
9.         $comps \leftarrow comps \cup \{(sub)^*\}$
10.     $seq+ = comps$
11. return $\text{combineComponents}(seq, \text{aliases})$

**Function** FindPathsInLoop($loop, header$)

1. let $ups \leftarrow \emptyset, path \leftarrow \langle \rangle$
2. for $succ \in header.\text{successors} \land succ \in loop$ do
3.   $\text{path.push}(succ)$
4.   $ups \leftarrow ups \cup \text{RecursiveDFS}(path, \{header\}, loop)$
5. $\text{path.pop}()$
6. return $ups$
For each outer loop we use function RecursiveDFS, which is used to find simple paths in the client method (Line 5) and simple paths in a loop (Line 34), by passing the header as the exit node. If the node is also the header of an immediate inner loop, we insert the sub-usage scenario in the inner loop with repetition marks after the node. Note that we only find the simple paths inside the loop from the loop header’s successor to the loop header and do not consider the simple paths which exit the loop not from the loop header (the third parameter loop in Line 14 and the condition succ ∈ N at Line 12). For example, the simple paths inside the second loop in Figure 5.2 only include the simple path \( l_8, l_{12}, l_7 \). The path \( l_8, l_9, l_{10} \) is not a simple path inside the loop because it exits from the loop at Line 10 instead of the loop header (Line 7) and it has been covered as the sub-path of the simple path \( l_2, l_3, l_7, l_8, l_9, l_{10}, l_{15} \) during traversal of its enclosing blocks (the main method).

After extracting the usage scenarios, we use a simple heuristic to generate all possible call sequences by repeating each sub-sequence scenario. According to an empirical study [74], structured Java programs have an average loop depth of less than 3, and the number of loops inside a single method is on average less than 2. Therefore, we extend these sub-sequences qualified by the * mark for zero, one, and two times, i.e., we traverse each loop in the client program zero, once, twice. In addition, we insert the class constructor method before each call sequence.

This simple heuristic can generate typical usage scenarios and avoid the exponential number of generated call sequences, but noises (both false positives and false negatives) may exist in the extracted call sequences due to limitations of static path-sensitive analysis (and limitations in alias analysis) and the exponential number of paths. The false negatives are caused by extracting a finite number of call sequences from exponential number of paths: some valid call sequences which can only be collected by the loops more than 2 times are not collected by this heuristic. The false positives call sequences are caused by the false repetitions (combinations) of different paths in the same loop or different loops. We remark that the problem that the statically extracted call sequences contain both false positives and negatives are universal in all techniques (e.g., the work by Nguyen et al. [143] and the work by Raychev et al. [161]) based on learning from
static call sequences and not unique in H1STAR. This problem motivates us to use active learning algorithm to learn from testing results as well.

5.4 Passive Learning

We adopt an Hidden Markov Models (HMMs) \[157\] training algorithm to train an HMM for each used JDK class from collected call sequences.

5.4.1 Hidden Markov Models

A Hidden Markov Model (HMM) is a statistical Markov model in which the system being modelled is assumed to be a Markov process with unobserved (hidden) states. The Markov process satisfies the Markov property that the future events depend only on current state and not on previous states. An HMM \( M = (S, \Sigma, \pi, A, B) \) consists of a set of \( n \) hidden states \( S \), i.e., \( |S| = n \), an initial state probability distribution \( \pi = \{\pi_1, \pi_2, \cdots, \pi_n\} \) for all states in \( S \), the set of \( n \) emission probability distributions \( A = \{a_1, a_2, \cdots, a_n\} \), one probability distribution for each state to emit a symbol in \( \Sigma \), and a set of \( n \) transition probability distributions \( B = \{b_1, b_2, \cdots, b_n\} \), one for each state to transit to other states. A probability distribution is a finite set of nonnegative real values \( \{p_1, p_2, \cdots, p_n\} \) such that \( \forall 1 \leq i \leq n \cdot 0 \leq p_i \leq 1 \) and \( \sum_{i=1}^{n} p_i = 1 \). \( M \) satisfies the Markov property in the sense that each state’s initial state probability, emission probability distribution, and transition probability distribution only depend on this state.

Example 5.1. We use the “biased coins” example \[157\] to illustrate how HMMs work and the problems with HMMs. Suppose there are 3 biased coins each of which has different probabilities of giving a head (H) or tail (T) for a toss. Thus the HMM for this example has 3 states denoted as \( S = \{c1, c2, c3\} \) and \( \Sigma = \{H, T\} \). The initial state probability and emission probability of each state are shown in the lower table in Figure 5.7, i.e., \( \pi = \{\pi_1 = 0.2, \pi_2 = 0.4, \pi_3 = 0.4\} \) and \( A = \{a_1 = \{a_1(H) = 0.6, a_1(T) = 0.4\}, a_2 = \{a_2(H) = 0.25, a_2(T) = 0.75\}, a_3 = \{a_3(H) = 0.45, a_3(T) = 0.55\}\} \). The transition probabilities for each state is shown on each transition of the upper diagram
in Figure 5.7. An observation sequence is a sequence of coin tossing results (either H or T) which can be generated by tossing the 3 coins in the following process:

(1) Randomly choose a coin to toss according to the initial state probability distribution.

(2) Toss the current coin and the probability of giving a tail or head is in accordance with the emission probability of the current coin and then record the result of this coin tossing.

(3) Randomly choose a new coin according to the transition probability distribution of the current coin and set it as the current coin and goes to step (2).

5.4.2 Training HMMs

We train the single class HMM using a modified Baum-Welch algorithm [143] which follows the iterative procedure of the Baum-Welch algorithm [31]. The modified algorithm stores the call sequences as a map and combines multiple occurrences of a call sequence in training dataset into one map entry which maps the call sequence to its number of occurrences. The number of hidden states in the HMM should also be specified as input before training. At each iteration the forward-backward computation method, which reduces the computational cost, is used to compute the parameters \( \pi, A, \) and \( B \) of the new model based on current model parameters. A scaled version of the forward-backward method is used to solve the numerical underflow problem.
Chapter 5. Learning Models via Passive and Active Learning

5.4.2.1 Convergence Criteria

The iterative training procedure in the Baum-Welch algorithm needs a convergence criterion to terminate. Besides setting a maximum number of iterations, we use a distance measure \( D(\mathcal{M}_1, \mathcal{M}_2) \), which measures the distance of \( \mathcal{M}_1 \) generated in current iteration and the HMM \( \mathcal{M}_2 \) generated in previous iteration, to calculate the convergence criteria. \( D(\mathcal{M}_1, \mathcal{M}_2) \) is defined as \( \frac{1}{N} \sum_{i=1}^{N} \left( \frac{\ln \Pr(s^i | \mathcal{M}_1) - \ln \Pr(s^i | \mathcal{M}_2)}{|s^i|} \right) \), where \( N \) is the number of training call sequences \( \{s^1, s^2, \ldots, s^N\} \) and \(|s^i|\) is the length of the call sequence \( s^i \). If the distance \( D(\mathcal{M}_1, \mathcal{M}_2) \) is less than a threshold value, we terminate the iterative procedure and return \( \mathcal{M}_1 \) as the trained HMM.

5.4.2.2 Initial Model

The Baum-Welch algorithm tends to find a local minimum with a given initial model. To avoid these local minimums, we perform the training algorithm several times with different initial HMMs. The parameters \((\pi, A, B)\) of each initial HMM are initialized randomly. Among the HMMs generated with different initial models, the one which maximizes the generation probabilities of the training call sequences is selected as the output HMM for a specific number of hidden states in the HMM.

5.4.2.3 Optimal HMMs

The number of hidden states in the target HMM is also an input for the Baum-Welch algorithm and thus should be specified before training. However, there is no systematic way of setting the best number of hidden states in the HMM. We use a simple heuristic to choose the HMM which maximizes the generation probabilities of sequences in a validation dataset. For the collected call sequences, we divide them into two datasets: the training dataset and the validation dataset. We set the number of hidden states to be within a maximum number. For each candidate number of hidden states, we use the training dataset to train an HMM and use the trained HMM to calculate the sum of generation probability for each sequence in the validation dataset. Finally, we
choose the HMM that has the highest sum of generation probability for sequences in the validation dataset.

### 5.4.3 Validity Thresholds

The trained HMM will be used as the teacher of experiences in the active learning phase to calculate the generation probability of a call sequence and the generation probability is compared with a threshold probability to decide the validity of the call sequence. The threshold probability is set as the minimum generation probability of all sequences in the training dataset with the same length as the given call sequence, plus a delta value $\delta > 0$. In case that the call sequence for a query is longer than the longest call sequence(s) in the training dataset (this is possible because the call sequences are generated by active learning algorithm), the minimal generation probability of the longest call sequence(s) in the training dataset is used. The rationale behind this threshold is that the probabilities in the trained HMMs are generated based on the training dataset and HMM training algorithm also generalizes from the training dataset. The effect of generalization is that some call sequences which are not in the training dataset may be treated as valid call sequences according to their generation probabilities. Thus the delta value is used to flag call sequences whose generation probabilities fall in the range of the minimum probabilities (with errors $\delta$) as suspicious. A call sequence whose generation probability is larger than the minimum generation probability plus the delta value is treated as a valid call sequence. However, if its generation probability is less than the threshold probability, there are two possible explanations for its relative small generation probability: the call sequence is indeed an invalid call sequence or it is a valid call sequence but is not contained in our training dataset.

The basic idea is to use dynamic testing to confirm the validity of a suspicious call sequence. However, an HMM describes valid call sequences only (because HMM is trained from valid call sequences extracted from client programs) and $L^*$ asks both valid and invalid call sequences and the number of valid call sequences is a small portion of the number of all possible call sequences (in theory there could be exponential number of call sequences in terms of the number of methods in the target class). Thus the
number of call sequences with small generation probabilities is much larger than the number of valid call sequences. In order to reduce the number of running random testing, we set another threshold probability which is the lower value of the threshold probability, i.e., the minimum generation probability of all call sequences in the training dataset with the same length as the call sequence in the query, minus $\delta$. We only need to run random testing for call sequences whose generation probabilities fall between the two thresholds. If a call sequence whose generation probability is smaller than the smaller threshold probability, it is treated as an invalid call sequence without testing it.

## 5.5 Active Learning

HMMs are not good for human interpretation and we adopt the active learning algorithm $L^*$ [24, 163] and testing to learn a DFA $U$ for a target class from its HMM.

### 5.5.1 Membership Query

We use the HMM to calculate a generation probability of the call sequence and compare the generation probability with the higher threshold probability. If it is larger than the higher threshold probability, the call sequence for the query is treated as a valid call sequence; if it is smaller than the lower threshold probability, it is treated as an invalid call sequence. If the generation probability falls between the lower and high threshold probabilities, we use dynamic testing to double-check whether call sequence is valid or not. Compared with symbolic execution or pure testing based active learning techniques [22, 191], the number of call sequences which need to be executed dynamically is much smaller.

### 5.5.2 Candidate Query

The teacher needs to check whether a candidate DFA is equivalent to the unknown DFA. However, the oracle in our setting (HMM) is not a DFA and thus they are incomparable. We generate a set of call sequences from the candidate DFA and use the HMM to
calculate their generation probabilities. Then we check whether the validity of a call sequence given by the HMM is consistent with its validity in the DFA. A sequence is inconsistent if it is valid according to the HMM (the validity of the call sequence is decided the same way as call sequences in membership queries) but it is invalid according to the candidate DFA (the last state of the call sequence is a rejecting state) or it is invalid according to the HMM but it is valid according to the candidate DFA (the last state of the call sequence is an accepting state). We return an inconsistent call sequence as the counter-example for the candidate query. If no inconsistent call sequence exists, we say that the candidate DFA is a good approximation of the HMM and return the candidate DFA as the final DFA learned by the $L^*$ algorithm from HMM and testing.

There are an exponential number of call sequences that can be accepted by a DFA, thus to cover all accepting sequences of a DFA is infeasible. We use random walking [54, 111] on the DFA to generate a set of call sequences of the same length larger than the maximum length of the membership queries for consistency checking. Since the HMM captures valid call sequences only, thus it can only decide whether a call sequence is valid, but cannot decide whether a sequence is invalid (this is also the reason we introduce dynamic testing to handle invalid call sequences). According to the semantics of program language, a call sequence is treated as invalid if any of its prefix call sequences is invalid. Thus $L^*$ at each iteration tries to split one of the accepting states if the candidate query returns a counterexample. Thus we only extend accepting state in the candidate DFA. This approach also reduces the number of running dynamic testing.

5.5.3 Random Testing

For a call sequence whose generation probability falls between the lower and upper threshold values, we cannot decide whether the call sequence is really invalid or due to the lack of such call sequences in the training data. Therefore, in order to confirm the validity of such call sequences, we adopt the testing technique, i.e., to run the call sequence, to confirm its validity. However, given a call sequence, we have to generate the concrete parameters before we can actually run it. The details of how to generate and execute tests for a call sequence are explained in Section 3.4 of Chapter 3.
construction method (either a call to the constructor or a static method returning an
object of this class), otherwise the sequence cannot be executed. If no concrete call
sequence for the call sequence throws any exception it is valid; otherwise, it is invalid.

5.6 Evaluation

We implemented HiSTAR in Java. The call sequence extraction component uses Soot
to analyze the Java bytecode in each downloaded jar file. We use the open source
library jahmm [77] for training HMMs and calculating the generation probability of
each call sequence for membership queries and candidate queries. We implement the
$L^*$ algorithm [24, 163] and the testing component by ourselves.

In this evaluation we run HiSTAR on a set of Java Bytecode programs extracted
from Maven repositories to synthesize behavior models for classes defined in JDK. We
aim to investigate the following research questions:

RQ1. Can HiSTAR be applied to real world programs in large scale.

RQ2. What is the quality of the synthesized behavior models.

RQ3. How does HiSTAR perform in comparison with other related tools, active learn-
ing based approaches, ad-hoc automata inference.

5.6.1 Settings

We download more than 60$K$ unique jar files (only the latest version of jars from
the same product are used) as the client programs for all JDK classes defined inside
java.* packages. The call sequences for all these classes are collected from client
programs using our call sequences extraction component. Call sequences for a class are
filtered out from the mixed outputs of all sequences for classes in the same package.
For each target class we train an HMM and then use $L^*$ and testing to learn from the
HMM a DFA as the behavior model for the target class.
TABLE 5.1: Statistics of artifacts used and generated by HiSTAR

<table>
<thead>
<tr>
<th>Client Jars</th>
<th>Number of client jars</th>
<th>60,932</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Number of classes</td>
<td>5,468,875</td>
</tr>
<tr>
<td></td>
<td>Number of methods</td>
<td>50,474,219</td>
</tr>
<tr>
<td></td>
<td>Number of unique classes</td>
<td>3,320,202</td>
</tr>
<tr>
<td></td>
<td>Number of unique methods</td>
<td>29,919,821</td>
</tr>
<tr>
<td></td>
<td>Disk space for client jars</td>
<td>50.88 GB</td>
</tr>
<tr>
<td>Call Seqs</td>
<td>Number of classes in java.*</td>
<td>1,051</td>
</tr>
<tr>
<td></td>
<td>Number of call sequences</td>
<td>444,553K</td>
</tr>
<tr>
<td></td>
<td>Avg. number of call seqs. per class</td>
<td>422,981</td>
</tr>
<tr>
<td></td>
<td>Avg. length of call seqs.</td>
<td>11.62</td>
</tr>
<tr>
<td></td>
<td>Disk space for call seqs.</td>
<td>13 GB</td>
</tr>
<tr>
<td></td>
<td>Number of used methods</td>
<td>11,094</td>
</tr>
<tr>
<td></td>
<td>Disk space for API names</td>
<td>4.3 MB</td>
</tr>
<tr>
<td>HMMs</td>
<td>Number of trained models</td>
<td>931</td>
</tr>
<tr>
<td></td>
<td>Avg. number of states</td>
<td>17.3</td>
</tr>
<tr>
<td></td>
<td>Avg. number of transitions</td>
<td>84.4</td>
</tr>
<tr>
<td></td>
<td>Avg. number of training iterations</td>
<td>41.3</td>
</tr>
<tr>
<td></td>
<td>Total training time</td>
<td>19,765s</td>
</tr>
<tr>
<td></td>
<td>Total space for trained models</td>
<td>16 MB</td>
</tr>
<tr>
<td>DFAs</td>
<td>Number of learned models</td>
<td>858</td>
</tr>
<tr>
<td></td>
<td>Avg. number of states</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>Avg. number of transitions</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>Avg. learning time</td>
<td>97 ms</td>
</tr>
<tr>
<td></td>
<td>Avg. number of membership queries</td>
<td>665.3</td>
</tr>
<tr>
<td></td>
<td>Avg. number of candidate queries</td>
<td>2.1</td>
</tr>
<tr>
<td></td>
<td>Avg. number running testing</td>
<td>721</td>
</tr>
<tr>
<td></td>
<td>Avg. time for testing</td>
<td>88.4 ms</td>
</tr>
<tr>
<td></td>
<td>Avg. time for HMMs to generate prob.</td>
<td>3.8 ms</td>
</tr>
<tr>
<td></td>
<td>Total space for learned models</td>
<td>16MB</td>
</tr>
</tbody>
</table>

During HMMs training for a target class we set the maximum number of 30 hidden states. The threshold used in the convergence criteria is empirically set to 0.3. We generate 3 different initial HMMs for training the 3 candidate HMMs for each of the given number of hidden states. The delta used for computing upper and lower threshold probabilities for answering queries is $0.02/\text{len}^2$, where \text{len} is the length of the call sequence in the query. For the random testing component, we generate a maximum of 2 successful concrete sequences for a call sequence with a maximum try-out of 100 parameters combinations. We allow a ratio of 0.1 of \text{null} values assigned to reference type parameters. All the data are collected on a 64-bit Ubuntu/Linux 16.04 physical machine with 12 cores (3.50GHz Intel Xeon(R) CPU) and 32GB RAM.
5.6.2 Performance of HiSTAR

Table 5.1 shows the statistics of HiSTAR on extracting call sequences (rows in “Call Seqs”), training HMMs (rows in “HMMs”), and generating DFAs (rows in “DFAs”).

5.6.2.1 Performance of Call Sequences Extraction

The total time taken for statically extracting call sequences from the 60K maven jar files is 8,087,695 seconds (about 2,246.6 hours or 93 days). The average time taken for processing each jar file is about 132.7 seconds including the time taken by Soot for loading the jar file. We observed that on average Soot takes about 1 second to load a jar file. The time taken by Soot to load jar files are in total 49,608 seconds (about 13.8 hours). Although the total time taken for extracting call sequences is long, this step can be parallelized. We used three PCs with the same configuration to extract the call sequences and it took about 1 month to collect the call sequences.

We could alternatively use the call sequence extraction algorithm in DroidAssistant [143] to extract call sequences. We re-implemented its algorithm (following the description in that paper) by using the Soot framework (for CFG construction) to extract call sequences from maven jar files. Our implementation of the algorithm took 2 days to process about 1,000 jar files and then we stopped using it. A further investigation shows that its algorithm extracts the call sequences at the end of a program path and the number of (static) program paths in a client method (in Bytecode) can easily exceed 1 million even by traversing each loop at most once due to “switch case”, many consecutive “if”, and nested loop in Java source programs. This phenomenon of large number of program paths is also observed by Harris et al. [87]. The other problem hinders its algorithm is that many path segments (basic blocks) in the client program do not call a method in the target classes; the algorithm has to traverse all the path segments before it knows that the path does not contain any interesting method calls, and thus the time spent on traversing the path is wasted.

Our call sequence extraction algorithm traverses each simple path inside a loop to check whether the path segments in the simple path contain any method calls to target
classes and caches them if any. Then we instantiate the loops with simple paths which contain interesting API calls. Although the path explosion problem also exists in our algorithm, our algorithm is able to calculate how many paths inside a client method (by multiplying the number of simple paths inside each loop by traversing the nested loops outwards) and extract a maximum number of call sequences.

5.6.2.2 Performance of HMMs Training

Although 1051 classes and interfaces inside the java.* packages have been used by some client programs. For some classes, all its call sequences contain only one method call, i.e., the lengths of the call sequences for the target class are 1. These call sequences are too short to train a meaningful HMM and we discard such classes. For example, some subclasses (e.g., java.util.IllegalArgumentException) of the java.lang.Exception are often called by creating an object of such exception and throws the exception object in the normal code (HiSTAR ignores the statements in the “catch block” due to limitation of Soot on generating too many spurious control edges to exception handlers [99]). HiSTAR trains 931 HMMs out of 1051 target JDK classes. It takes in total 19,765 seconds (or 5.5 hours) to train the HMMs and on average it takes 21.3 seconds to train 3 HMMs (with 3 initial models). The average number of states of the trained 931 HMMs is 17.3 and the average number of transitions whose transition probabilities greater than or equal to 0.001 is 84.4.

5.6.2.3 Performance of DFAs Learning

HiSTAR generates DFAs from the HMMs in combination with random testing. In total HiSTAR generates 858 DFAs from 931 HMMs. Only one method of some classes is called by some client programs (note the difference from the case in above section: each call sequence may contain multiple calls to the only used method in the target classes). This is often due to the fact the method is called on a parameter of the client method or an instance field which has already been initialized. We also ignore such classes. For example, the only called method of the class java.beans.ExceptionListener is exceptionThrown(Exception).
The statistics starting with “Avg.” for learning DFAs in Table 5.1 are averaged over all generated DFAs. The number of states and valid transitions (both source and target states of a valid transition are accepting states) is 5 and 21, respectively. Compared with HMMs, the numbers of states and valid transitions are much smaller, which makes it easier for human interpretation. Answering queries with HMMs is much faster than testing (3.8 ms for all call sequences vs. 88.4 ms for tested call sequences).

5.6.3 Quality of Generated Models

We use the benchmark created by Pradel et al. [152] as the reference behavior models for evaluating the quality of the DFAs generated by HiSTAR. The benchmark contains 32 DFA models for 32 Java standard classes under java.* packages. These models are generated from their API documentation using an algorithm and thus the models can be considered to be both accurate and complete. Given a reference model $R$ and our model $M$. The precision (recall) of $M$ with respect to $R$ is calculated based on the following steps [152]:

1. Compute the deterministic and minimal union automaton $U$ of $M$ and $R$ and mark all transitions in $U$ that come from $M$ (from $R$).

2. Compute $U'$ by applying a modified k-tails algorithm [36] to $U$ while propagating which transitions are marked. Make $U'$ deterministic and minimize it. Let $m$ be the number of marked transitions in $U'$.

3. Compute the intersection of $U'$ and $R$ (the intersection of $U'$ and $M$) and let $m_t$ be the number of marked transitions in $U'$ while computing the intersection.

4. The precision (recall) of $M$ with respect to $R$ is $m_t / m$.

The modified k-tails algorithm in step (2) is used to merge states of $U$. The set of call sequences (of length $k$) which are enabled at a state is called the $k$-tails of the state. The modified k-tails algorithm merges two states if one state’s $k$-tails is a subset of the other state’s $k$-tails. Thus, the precision and recall are influenced by the parameter $k$ of the
Figure 5.8: The precision and recall plots of 30 DFAs generated by HiSTAR with respect to the benchmark DFAs. The cyan bar stands for the median, the upper whisker and the lower whisker is the maximum and minimum value, respectively.

k-tails algorithm. The final precision and recall are the average of the precisions and recalls for \( k = 1, k = 2, k = 3 \), and the exact comparison between \( M \) and \( R \) [152].

We plot the precision and recall for each of the DFAs with respect to the corresponding reference model in Figure 5.8. In total we are able to generate 30 DFAs for 32 classes in the benchmark, the DFAs for the class `java.net.Socket` and `java.net.MulitcastSocket` cannot be generated because the method `receive` waits for incoming messages but there is no message and thus blocks our tester. For the 30 DFAs generated by HiSTAR the precision is all high because methods that are not used in the reference model are filtered out. The highest recall is 100% for the class `java.util.StringTokenizer`, because it has many valid call sequences (after preprocessing) and the objects of this class are often used as local variables in a client method. We found that models with low precision are caused by insufficient good training data. For example, the precision of the DFA for the class `java.net.URLConnection` is low, this is because we require that all valid call sequences for training HMMs should start with a constructor call, but this class is an abstract class and has no callable constructor. The conclusion is that good and diverse training data is the key for learning good HMMs (and DFAs).
5.6.4 Comparison with Related Tools

We identified the following work which also generates behavior models by learning from either statically extracted call sequences or dynamical call sequences. The tool TAUTOKO [64] requires a test suite for the target program but our client programs are not runnable without proper configurations. Thus we do not compare with this tool.

DroidAssistant learns HMMs from call sequences extracted from Android Apps and uses HMMs for code recommendation. We do not compare the HMMs generated by HiSTAR with those generated by DroidAssistant for code commendation based on the following two reasons: (1) different inputs are used for training HMMs in the two approaches: the call sequences in HiSTAR are extracted from Java client programs and those in DroidAssistant are extracted from Android Apps. Furthermore, the detailed settings (e.g., convergence criteria, initial model generation) for training HMMs in DroidAssistant are not available, thus the comparison would not be fair, and (2) we filtered out call sequences which do not start with the method call to a constructor and DroidAssistant does not filter out such call sequences. This filtering may have impact on the accuracy for code recommendation because the call sequences extracted from client programs without this filtering are more close to reality. The receiver of a method call may be passed in as parameter or be an instance filed and the constructor of the receiver has been called in other places. This filtering discards these traces. The experiments in DroidAssistant demonstrate that code recommendation based on HMMs has better accuracy than those approaches based on n-gram and RNNs, thus we do not compare with SLANG [161] which is based on n-gram and RNNs.

The tools TzuYu [191] and TLV [177] actively learn DFAs with guard conditions using testing and symbolic execution. We run TzuYu on the JDK classes in the benchmark and TzuYu can run the only class java.net.URLConnection. The generated DFA for this class is invalid because it has no transitions to an accepting state. TLV uses both testing and symbolic execution and it suffers the same problem as TzuYu. Although the testing component in our approach suffers from the same problem in which case the generated DFAs are answered by HMMs only. The tools Psyco [80] and X-Psyco [91] are based on static symbolic execution and dynamic
symbolic execution, respectively. They require the source code of JDK and we used
the source code for OpenJDK version 6 as the input and the two tools cannot generate
models for any class. Because the symbolic execution engines used in the two tools
cannot handle Java constructs like reference types and class inheritance which are used
by most of the classes in JDK.

5.6.5 Threat to Validity and Discussions

There are several threats to the validity of this approach. As in other data mining tech-
niques, there are many parameters to tune in order to generate good-quality HMMs. The
parameters used in our experiments may not be valid for other types of data. The testing
component uses random testing to generate parameters for methods in a call sequence
and cannot generate high-quality parameters for reference types. One possible solution
is to use program slicing [189] to automatically extract runnable code snippets related to
parameter construction from client programs. One external threat is that our approach
was only evaluated with JDK APIs, it is unclear how the approach would behave in
other types of APIs.

5.7 Summary

We combine statistical learning (passive learning from a dataset) and active learning
(which may generate new data for queries) in synergy to learn behavior models for sin-
gle Java standard classes. HMM training algorithm is used to learn models for all call
sequences which are statically extracted from client programs of the target class and $L^*$
algorithm is used to learn DFAs from HMMs and testing results. The two learning ap-
proaches complement each other. We apply the proposed approach to generate behavior
models for more than 800 Java standard classes. Our approach can be applied to learn
the behavior models for any library given that it has many client programs.

We made the following contributions: (1) we propose a scalable approach which
combines passive learning and active learning to automatically learn behavior models of
JDK classes, to our best knowledge, we are the first to combine statistical learning with active learning for learning behavior models. (2) We apply HiSTAR, an implementation of the proposed approach, to generate behavior models for about 800 Java standard classes inside `java.*` packages from more than 60K maven jar files. We evaluate the generated behavior models with respect to a small reference benchmark models and the results show that the generated behavior models are rather precise.

Robillard et al. [164] have done a comprehensive survey on various API properties inference techniques. In this section we only survey the related work in behavior models synthesis and focus on three aspects of each work: (1) input data, (2) synthesis technique, and (3) output model. There are two main categories of approaches for synthesizing behavior models: passive inference and active inference.

DroidAssistant [143] generates HMMs from static call sequences for code recommendation; our work generates human interpretable models from HMMs and testing results. The call sequences extraction algorithm of HiSTAR differs from that of DroidAssistant in intermediate representations and efficiency. The detailed differences between HiStar and DroidAssistant are explained in the evaluation section. SLANG [161] uses static analysis to extract call sequences and build $n$-gram and Recurrent Neural Networks (RNNs) for code completion. ASTLan [141] uses a statistical graphical model to represent method calls in client programs together with the control structures (context) in which the methods are called and uses Bayesian inference to calculate the conditional probability of a graph given a partial graph. Compared with $n$-gram, RNNs, and graph models, training HMMs is more efficient than training $n$-grams, RNNs, or graph models because an HMM has less parameters to estimate from the training data than the other models.

The second group of techniques in this category generate finite state machines models from runtime call sequences. They assume there are test cases for a library and then instrument and run the test cases to collect runtime call sequences. The test cases are often found in test suites [65, 66, 121, 153] for the library or generated from models with model based testing techniques [64]. The sk-strings [159] algorithm uses a heuristic based inference technique to infer probabilistic finite state automata (PFSA) from a set of dynamic or static traces. The call sequences generated from the dynamic approaches
are accurate, but they are limited because test suites for a library are often unavailable or have lower coverage for many usage scenarios. Ammons et al. [23] generates DFAs from PFSAs.

The third group of techniques use other data mining techniques to mine other forms for behavior models from source code of client programs. Behavior models are represented as partial order among methods [16, 188], association rules [123], item sets [41], subgraphs [51, 144], code idioms [18], and topic modelling [136].

Active learning based techniques for synthesizing behavior models often use the $L^*$ algorithm or its variants to learn behavior models and use techniques such as symbolic execution [80], testing [191] or both [91, 177] to answer queries in the $L^*$ algorithm. Techniques in this category do not require many client programs of a library but assume the code of a library is available. HiSTAR answers most of the queries with HMMs and a small portion of the queries using testing. Due to limitations of symbolic execution in dealing with loops and third party libraries and the path explosion problem, the models generated by active learning with symbolic execution are accurate but not scalable. Active learning based on testing is plagued with poor capability of automatically generate meaningful test inputs for call sequences, thus many interesting API usages cannot be covered by testing based active learning approaches.
Chapter 6

Learning Models for Malicious JavaScript Detection

Starting from this chapter, the second part of the thesis is devoted to the applications for learning the attack behavior models of malicious software. The learned attack behavior models are then used to detect malware variants. This chapter focuses on learning attack behavior models to detect malicious JavaScripts.

6.1 Introduction

As a standardized dynamic language and one the three core languages for building interactive web contents, JavaScript (JS for short) gains native support from all major web browsers. Due to pervasive support by web browsers, malicious JavaScript programs are the ideal tools to attack browsers and their host systems. According to the security report [12] of Microsoft, malicious JavaScript programs account for the highest number of its detected malwares in the first half of 2013.

Commercial anti-virus software often uses signature [72] based techniques to detect malicious software efficiently. The signature based approaches generate a hash value as the signature for a known malware and apply the same signature generation process
for a suspicious software, if the signature of the suspicious software matches the signature of a known malicious sample, the suspicious software is detected as a malware. One major drawback of these approaches is that they can be easily evaded with code obfuscation techniques which cause the obfuscated version of the malware to have a different hash value [193] than the original malware. Furthermore, our own experience with VIRUS TOTAL [10], which aggregates the results from 56 anti-virus software, suggests that these anti-virus tools give different family name (or type) for the same type of JS malware, not to mention behaviors.

Beside signature based approaches, other approaches for detecting JS malware can be grouped into two categories. Approaches in the first group use machine learning techniques to capture malicious characteristics of malware and detect malware efficiently. They use syntactic information [63,100] or dynamic information (e.g., API used by the JSs [62,162]) as the features to classify a JS program as malicious or benign. These approaches have two main limitations due to representing the semantic information of a malware through its syntactic information: (1) like signature-based approaches, these approaches can be easily evaded through alternating the syntax of the malware; (2) these approaches cannot be used to detect new types of malwares nor infer the attack type of a malware.

Approaches in the second category detect malicious JS using dynamic analysis in sandboxes. They monitor the execution of a suspicious JS program in a sandbox to collect its runtime information. The collected information is then compared with runtime information of benign JSs as an outlier detection problem—the JS is detected as malicious if its runtime information is not consistent with the information for benign JSs. Several approaches are capable of detecting specific types of JS malware such as heap-spray attacks [73,160], worms [46,120] and JS shell code embedded in Adobe Reader [182]. The major drawback of these approaches is that they are not designed to be a general framework to detect other types of JS malware. Compared with other detection techniques, these approaches take longer time caused by dynamic execution. Besides, Hallaraker and Vigna [86] propose a general framework to detect malicious JSs in Firefox with defense policies. But their approach is not able to infer the family or attack type of a JS malware.
Chapter 6. Learning Models for Malicious JavaScript Detection

The above approaches do not use the attack behavior as a feature to detect JS malwares. As in previous chapters where behavior models for a program library are used to captures the behaviors of the library in a succinct way, the behavior model can also be used to characterize the common attack behaviors of JS malware in the same family. In this chapter, we propose to learn the attack behavior models of representative JS malwares in the same family automatically and then use the learned attack behavior model to detect and classify emerging JS malwares.

Automatic learning of attack behavior models of JS malware faces two challenges. Firstly, sometimes the attack behaviors of malwares in the same family are highly diverse due to too coarse criteria are used to categorize malware families, e.g., the family of JS malware named Trojan.js.iframe.* has the same behavior for code injection but has diverse attack behaviors. A meaningful way of classifying JS malwares according to their attack behavior is necessary for detecting new attacks. A good way of classifying JS malwares according to their attack behaviors involves domain expertise and is often done manually. Thus it is challenging to automate the processes of learning attack behavior models and of classifying JS malwares. Secondly, JS is a dynamic language which use dynamic typing and thus it makes static analysis of JS program very difficult if impossible, i.e., to learn an attack behavior model, we have to resort to dynamic analyses of the JS malwares.

To address the first challenge, we propose to use behavior models of the browser-level system calls which are used by the JS malwares (hereafter referred as attack behavior models) to represent their attack behaviors. A JS program uses these system calls provided by the host browser to interact with the browser. As demonstrated in previous chapter, it is effective to model the behaviors of a program with system calls or actions [32,45,103]. The attack behavior model contains possible malicious executions of this type of attack including variants in the same family.

To address the second challenge, we propose a dynamic approach to automatically learning an attack behavior model for each family of JS malwares. Given both benign and malicious execution traces of a family of JS malwares, we filter out security-irrelevant system calls. The processed system executions contain only security-relevant system calls and are grouped into high-level actions that represent a meaningful and
security-relevant behavior of JavaScript code. Subsequently, we extract the common actions that appear in all the malicious execution traces, and represent execution traces as action sequences.

Second, we develop a learning framework based on the $L^*$ algorithm [24, 163] to learn attack behavior models. The $L^*$ algorithm requires a teacher to answer membership queries and candidate queries. In this chapter, a membership query asks whether an action sequence is malicious or not; a candidate query asks whether the current attack behavior model is consistent with the common attack behaviors of malwares in the same family. We answer membership queries by combining data dependency analysis, defense rules and JS replay mechanism. To answer candidate queries, we adopt a random walk algorithm to generate a set of sequences from the learned DFA, and raise one membership query for each sequence. Lastly, the inferred attack behavior models are used to detect variants of modelled malwares and classify them accordingly.

The main contributions of this chapter are the following:

1. We propose to use attack behavior models to model JS malwares with the same attack types. The attack behavior models are effective for detecting existing types attacks and attack variants.

2. We propose an automatic dynamic approach to learn accurate attack behavior models from dozens of JS malwares. Our approach does not require many malicious traces for training. Instead, our approach uses data dependency analysis, defense rules, and JS replay mechanism to efficiently check whether a given trace is malicious.

3. We learn attack behavior models for 8 major families of JS malwares using 120 malicious JS samples. To evaluate the effectiveness of the learned models, we run more than 10K real-world JSs. The results show that our approach does not only outperform existing commercial malware detection tools, but also detects malware variants and even new attacks.

The rest of this chapter is organized as follows. Section 6.2 introduces the modelling of attack models of JS. Section 6.3 elaborates the overview of our approach. Section 6.4
and Section 6.5 explain more details of our approach. Section 6.6 explains the implementation details of our approach. Section 6.7 provides the evaluation of our approach. Section 6.8 reviews related work and concludes this chapter.

6.2 Attack Behavior Modelling

The prevalence of JS has attracted attackers to employ it for their malicious intentions. Malicious JS takes advantages of vulnerabilities or weakness of the client side with the aim to execute arbitrary instructions on the client’s machine. According to Kaspersky Security Bulletin [7], attacks targeting at JRE, Adobe Reader, browsers, Adobe Flash account more than 95% of all recent attacks launched by JS code. Following this trend, this study focuses on 8 infectious and hazardous JS attacks [186], i.e., attack targeting browser vulnerabilities (Type I), browser hijacking attack (Type II), attack targeting Adobe Flash (Type III), attack targeting JRE (Type IV), attack based on multimedia (e.g., images, videos) (Type V), attack targeting Adobe PDF reader (Type VI), malicious redirecting attack (Type VII) and attack based on Web attack toolkits (Type VIII). This categorization includes most of commonly seen JS attacks, e.g., in the recent list of 500 malicious samples reported by WEB INSPECTOR, we find that 411 (82.2%) fall into the above 8 types of attacks.

In dynamic approaches for security analysis of various attack types, system calls are normally used to model the malicious behaviors [45, 73]. The rationale is that malware triggers or payloads are often resource oriented activities, e.g., creating a process. In this work, we use Firefox as the targeted browser. In Firefox, browser-level system calls are system calls to the XPCOM [8] layer of Firefox (see Section 6.6.1). To better capture the malware behavior involving the interactions with XPCOM components, we use the lower level XPCOM method calls in this work rather than method calls of JS APIs. Here, we formally define a system call as a tuple \( sc = (I, M, N_p, S_p, T_r) \), where \( I \) is the interface name of \( sc \), \( M \) is the method name, \( N_p \) is the number of parameters, \( S_p \) is the list of arguments, and \( T_r \) is the return type. A JS execution trace is defined as a sequence of system calls \( \pi = (sc_1, sc_2, \ldots, sc_n) \), occurring in a chronological order.
<a href='javascript:

var file=Components.classes["@mozilla.org/file/local;1"].
createInstance(Components.interfaces.nsILocalFile);
var path = "/usr/bin/gnome-calculator";
file.initWithPath(path);
var proc=Components.classes["@mozilla.org/process/util;1"].
createInstance(Components.interfaces.nsIProcess);
proc.init(file);
proc.run(true,[path],1);
'></a>

**Figure 6.1:** The JS code of a web-based attack

| System calls $sc_i$: ($I$ | $M$ | $N_p$ | $S_p$ | $T_r$) | Actions $a_i$ |
|-----------------|-----------------|-----------------|-----------------|-----------------|
| nsIOService2 | newURI | 3 | data:text/html;base64,PHN...; | void | $a$ |
| nsIURI | scheme | 0 | | void | $b$ |
| nsPrefBranch | getComplexValue | 12 | intl.ellipsis;object; | void | $n_1$ |
| nsPrefLocalizedString | data | 0 | | void | $n_2$ |
| nsPrefBranch | getBoolPref | 11 | devtools.inspector.enabled; | void | $c$ |
| nsIOService | newURI | 3 | data:text/html;base64,PHN...; | void | $a$ |
| nsIURI | scheme | 1 | | void | $b$ |
| nsLocalFile | initWithPath | 1 | /usr/bin/gnome-calculator; | void | $d$ |
| nsProcess | init | 1 | object; | void | $e$ |
| nsProcess | run | 3 | true;object;1; | void | $f$ |
| nsSecureBrowserUI | init | 1 | object; | void | $n_3$ |

**Figure 6.2:** A concrete execution trace and the common actions \{a, b, c, d, e, f\} in the malicious traces of the attack in Figure 6.1

**Example 6.1.** We list the JS of an attack in Figure. 6.1. Due to the vulnerability of CoolPreviews (Mozilla Firefox Extension) [4], via a link pointing to a data URI which embeds the Cross-Site scripting payload, the malicious page can inject exploiting code that is rendered and executed in the Chrome privileged zone\(^5\). Although a calculator application is used in Figure 6.1, arbitrary code can be executed. Figure 6.2 shows the related system calls in a malicious trace of the attack in Figure 6.1. Each row represents a security relevant system call (see Section 6.4), where the five elements inside a system call are split by “|”. The trace shown on the right column only contains 12 security-relevant system calls after filtering out the security irrelevant ones from the original hundreds of system calls.

---

\(^5\)The Chrome privilege grants the JS code the permission to do everything in browser, which is similar to the root permission in OS. By default, JS is not allowed to create a file or a process outside the sandbox with Chrome privilege.
For end-host malware behavior modeling, existing studies use two types of behavior atoms [45], i.e., system call and action. The types of models include 4 basic structures (a $n$-gram of, a sequence of, a bag of, and a tuple of atoms), or even any combination of the basic structures [45].

Kolbitsch et al. [103] pinpointed that sequences of system calls are not suitable for attack behavior modelling, as a new variant can be easily crafted by reordering of the code to achieve the same goals. On contrary, bags of or tuples of atoms totally ignore orders or dependencies among atoms. Thus, Kolbitsch proposed to model program behavior in a behavior graph, which is essentially a data dependency graph of system calls at runtime. In behavior graph, two continuous system calls with no control or data dependency (no source-sink data flow) are considered isomorphic. Thus, the rationale is two-fold: system call sequence is not resilient to malware variants with reordered system calls, while bag of system calls may be too relaxed and introduce false positive cases. Behavior graph essentially models an attack using a bag of system calls together with their data dependency. As there exist many equivalent permutations of system call sequence for system calls in the same control block, behavior graph provides an effective data dependency model among system calls. However, graph based matching of behavior graphs [103] can be computationally costly for malware detection.

Note that this issue of JS behavior modelling has not been investigated in depth previously. Similar to using finite state automata (FSA) in modelling normal behavior via Linux system calls [167], we propose to use DFA to model different JS attack types based on browser-level actions. Different from [167], our contribution is to implement an online teacher to perform active learning.

### 6.3 Approach Overview

The workflow of our approach shown in Figure 6.3 contains several steps. Firstly, given the variants of a certain attack, we get both of benign and malicious execution traces by running these variants in the instrumented browser. Traces from malware variants of the same attack, not from benign software or various attacks, are used for training, as
Figure 6.3: The general workflow of our approach JS\textsuperscript{*} to modelling the behaviors of an attack type

JS\textsuperscript{*} targets at one type of attack each time when applied. Traces from general benign software may be irrelevant to the modelled attack, or even introduce noise for model learning. With the training traces from malware variants, the preprocessing step filters out security-irrelevant system calls and extracts actions from traces. Here, what we refer to as actions are high-level operations (e.g., creating a process) that can be implemented by different concrete system calls to achieve a meaningful goal [32, 45]. Subsequently, action sequences converted from execution traces serve as the training set for the JS\textsuperscript{*} learning (see Section 6.4).

In this study, the expected attack model is represented in form of a DFA $\mathcal{D}$ with a fixed alphabet $\Sigma$ (i.e., common actions extracted from malicious training traces). To infer $\mathcal{D}$, the JS\textsuperscript{*} learning module interacts with the polynomial time JS\textsuperscript{*} teacher via asking membership and candidate queries to make the observation table closed and consistent (see Section 6.5). The teacher answers membership queries effectively based on a combination of defense rules, data dependency analysis and replay mechanism (see Section 6.5.1). By regression testing and random sampling techniques, JS\textsuperscript{*} teacher can answer candidate queries efficiently (see Section 6.5.2).

Finally, the learned DFA serves as an abstract attack model, which can be used to identify the attack type of the captured traces from suspicious variants. However, there may exist some action sequences that are non-deterministic, i.e., the same action sequence may be malicious or benign, depending on its arguments. In Section 6.5.3,
we explain the inferred DFA and present a refinement for the DFA if a nondeterministic action sequence is found.

6.4 Trace Preprocessing

We start our approach by capturing browser-level system call traces during loading and rendering a web page through instrumenting the browser (see Section 6.6.1 for implementation details). Each recorded trace contains thousands of system calls, most of which are irrelevant to browser security. To precisely and concisely model attacks, removing security-irrelevant system calls and mapping the rest similar ones into high-level actions prelude further analysis.

**Irrelevant System Calls Filtering.** To the best of our knowledge, there is no official classification of XPCOM system calls in terms of security. In this work, we record all system calls generated during the process of launching the browser, loading and rendering a totally blank web page. We consider system calls to these 667 (out of the 1948 interfaces in XPCOM [8]) basic and very common interfaces as no security risks, i.e., *security-irrelevant system calls*, e.g., system call `nsIDOMWindow.GetscreenY` that gets the Y coordinate of a window. Most of these 667 interfaces are related to browser initialization and configuration. In our study, system calls uncommonly appearing in normal browser launching are *security-relevant system calls*, e.g., system call `nsIProcess.run` that executes a process.

**Action Abstraction.** Some system calls provide similar or identical functionalities, e.g., in Figure 6.2, `nsIPrefBranch.getBoolPref` and `nsIPrefBranch.getIntPref` get the Bool and Int type preference data; `nsIIOService1.newURI` and `nsIIOService2.newURI` both construct a new URI, respectively. Thus, we abstract system calls with similar functionalities as the same type of *action* for the purpose of variants detection. We name the actions alphabetically starting from ‘a’ in this work. After the abstraction, malicious traces based on system calls become abstracted action sequences. Due to multitasking of the browser, each execution of the same JS code may produce different action sequences but with the identical attacking route. Intersecting these action
Algorithm 6: extractCommonActions

input : $S_{\pi} = \{\pi_1, \pi_2, ..., \pi_n\}$, a set of malicious system call traces
output: $M$, the map whose key is an action and the value is the set of system calls abstracted by this action. Initially empty.
output: $\Sigma$, the set of common actions relevant to the attack

1 for each trace $\pi \in S_{\pi}$ do
   foreach system call $sc \in \pi$ do
      if $\exists a_k \in M$.keys $\bullet$ IsSameAction($a_k$, $sc$) then
         $M$.get($a_k$).add($sc$);
      else
         create a new action $a_n$ according to $sc$;
         $M \leftarrow M \cup \{(a_n, sc)\}$;
      end if
   end foreach
end for each

8 $\Sigma \leftarrow \text{GetAction}(\pi_1, M)$;
9 for each trace $\pi \in \{\pi_2, ..., \pi_n\}$ do
   $\Sigma \leftarrow \Sigma \cap \text{GetAction}(\pi, M)$;
end for each

sequences to extract common actions, making visible common behaviors of malicious traces.

Algorithm 6 illustrates how system calls are mapped into actions and then how common actions are extracted from the given set of malicious traces $S_{\pi}$. First, the map $M$ is built from lines 1 to 7. At line 3, each system call $sc$ in $\pi$ from $S_{\pi}$ is compared with each key $a_k$ in $M$ by calling $\text{IsSameAction}$ to check the functionality similarity, at Line 3. If the fully qualified name of $sc$ is similar to the fully qualified name of any system call in set $M$.get($a_k$), $sc$ belongs to $M$.get($a_k$). Being similar is satisfied if string similarity according to Levenshtein Distance [85] is above a threshold, 80% in our study [108]. We also refine the results of this step via the manual check, considering the limited size of system calls involved in each type of attack. Method $M$.get($a_k$) returns those system calls abstracted by $a_k$. If $\text{IsSameAction}$ returns true, i.e., $sc$ can be abstracted by action $a_k$, $sc$ is added into $M$.get($a_k$) (line 4). Otherwise, a new action $a_n$ is created according to next available alphabet (ASCII letter in our implementation) at line 6; and the pair $(a_n, sc)$ is added to $M$ at line 7. Practically, action abstraction step is manually verified, as the size of alphabet is small (see section 6.7).

After $M$ is built up, method $\text{GetAction}(\pi_1, M)$ at line 8 collects the actions in $\pi_1$ and assigns to $\Sigma$. From lines 8 to 10, all action sequences are iteratively used to build the set
Malicious Action Sequences $S_{\pi}$:  
- $\pi_{m1}: \langle c.a.b.d.e.f \rangle$
- $\pi_{m2}: \langle a.c.b.d.e.f \rangle$
- $\pi_{m3}: \langle a.b.c.d.e.f \rangle$
- $\pi_{m4}: \langle a.b.d.c.e.f \rangle$
- $\pi_{m5}: \langle a.b.d.e.e.f \rangle$
- $\pi_{m6}: \langle a.b.d.e.f.c \rangle$
- $\pi_{m7}: \langle c.a.c.b.e.c.d.e.c.f \rangle$
- $\pi_{m8}: \langle a.b.c.c.a.b.d.e.f \rangle$

Benign Action Sequences $S_{\pi}'$:  
- $\pi_{b1}: \langle c.b.d.e.f \rangle$
- $\pi_{b2}: \langle a.c.d.e.f \rangle$
- $\pi_{b3}: \langle a.b.c.e.f \rangle$
- $\pi_{b4}: \langle a.b.d.c.f \rangle$
- $\pi_{b5}: \langle a.b.d.e.c \rangle$
- $\pi_{b6}: \langle c.b.a.d.e.f \rangle$
- $\pi_{b7}: \langle c.a.b.e.d.f \rangle$
- $\pi_{b8}: \langle c.a.b.d.f.e \rangle$

Figure 6.4: Representing traces as sequences of common actions ($\pi_{m8}$ is the counterpart of the trace in Figure 6.2)

of common actions $\Sigma$—actions appearing in all traces in $S_{\pi}$.

Note that extracting common actions is only conducted on malicious traces. Only after that, the available benign traces are represented as sequences of common actions. Suppose we collected 8 malicious and 8 benign concrete system call traces, we apply action abstraction and intersection on these 8 malicious traces to build the set of common actions, i.e., $\{ a, b, c, d, e, f \}$ in Figure 6.2. Finally, in Figure 6.4 we represent these 16 traces in sequences of common actions, which serve as the input for the JS$^*$ learning approach.

6.5 JS$^*$ Learning Framework

This section is devoted to the explanation of the proposed JS$^*$ learning framework. Our approach is based on the $L^*$ algorithm, which learns a DFA from a set of regular strings [24, 163]. In our study, we combine domain knowledge (e.g., defense rules), program analysis (e.g., data dependency), and other techniques (e.g., replay mechanism, random sampling) to implement an effective teacher to answer these two queries (see Sections 6.5.1 and 6.5.2).

6.5.1 Membership Query

An ideal implementation of the online teacher for membership queries should be able to correctly answer if the system call sequence $\pi_\alpha \in \Sigma^*$ should be accepted (i.e., whether
πₐ represents an attack of the modelled type) in polynomial time. Practically, it is infeasible to extract all possible traces for a JS program and judge their maliciousness. Here, we propose a feasible and efficient way to utilize defense rules, data dependency analysis, and also JS replay mechanism to answer membership queries on the fly, as elaborated below.

### 6.5.1.1 Browser Defense Rule

Defense rules refer to security policies commonly used by the mainstream browsers to detect malicious attacks. Violation of such rules indicates possible security risks in existing studies [171]. Mozilla mainly adopts permission relevant policies: the same origin policy (used in [86] [98]), the Configurable Security Policies (CAPS) as Mozilla built-in zone-based rules [5], and the signed-script policy (used in [86, 196]). In our study, these defense rules are applied to the URL source, information and permission of executed JS code. By combining these rules in the teacher, we can test a trace—a violation of any defense rule indicates that the tested trace is malicious.

Note that defense rules are mainly used to check the trace that are tested by the JS* on the fly. Besides, using defense rules can only indicate whether a trace is benign or not [86], but fail to model the attack behaviors.

### 6.5.1.2 Data Dependency Analysis

Given the existing training traces, data dependency analysis is adopted to find equivalent permutations (EPs) of an action sequence. The assumption that the order of two mutually independent system calls (or actions) does not matter is reasonable [45] [103]. Thus, we adopt this assumption and infer EPs via data dependency analysis. For example, given malicious sequence π₁ = ⟨a₁, a₂, a₃⟩, where a₃ has data dependency⁶ on a₁ and a₂, denoted as {a₃ ← (a₁, a₂)}, we can infer π₂ = ⟨a₂, a₁, a₃⟩ is also malicious. So π₂ and π₁ are EPs.

---

⁶The data dependency is calculated based on the original system calls of the actions, we omit the implementation details for the interest of space.
Algorithm 7: isEqualPermutation

input : $S_\pi = \{\pi_1, \pi_2, \ldots, \pi_n\}$, a set of malicious action sequences
input : $\pi_a$, a given action sequence
output: true or false, if $\pi_a$ is an equivalent permutation

1 $S_{e\pi} \leftarrow \emptyset$
2 foreach sequence $\pi_i \in S_\pi$ do
3   $S_{dc} = \text{getDependencyClosure}(\pi_i)$
4   $S_{e\pi} \leftarrow S_{e\pi} \cup \text{getPermutations}(\pi_i, S_{dc})$
5 if $\pi_a \in S_{e\pi}$ then
6   return true
7 return false

Given a training set of malicious sequences $S_\pi$ and an unknown system call sequence $\pi_a$, Algorithm 7 shows how to test if $\pi_a$ is an EP according to the sequences in $S_\pi$. The basic idea is to get the data dependency closures among actions inside a sequence $\pi_i$ by invoking method $\text{getDependencyClosure}(\pi_i)$. In two steps, this method builds the dependency closure\(^7\). Given $\pi_i$, JS\(^*\) gets all direct dependencies among system calls inside this trace, from which actions are abstracted. If two system calls have a direct data dependency (see Section 6.6.1), the two relevant abstracted actions have a direct data dependency. Second, this method propagates the direct data dependency relationship among actions into a closed transitive indirect data dependency. In the running example, this method returns two closures $\{b \leftarrow a\}$ and $\{(e, f) \leftarrow d\}$. As $c$ is independent, actually, $\pi_{m1}$ to $\pi_{m6}$ (and other similar sequences only with different $c$ positions) are all EPs.

By $\text{getPermutations}$, the mutually independent relation between actions (e.g., $a_1$ and $a_2$ in the above case of $\pi_1$, $c$ and other actions in the case of $\pi_{m1}$) infers the equivalent permutations. All EPs should not equal to an existing benign sequence, and then be stored into a set for comparison with $\pi_a$. A right match indicates an EP. Practically, the calculation of $S_{e\pi}$ from lines 1 to 4 in Algorithm 7 is pre-built once as preprocessing. Due to the limited length of action sequence and the small size of training sequences, $\text{getPermutations}$ is scalable in reality, e.g., for the partial DFA in Figure 6.11, there is the dependency closures: $\{n \leftarrow m \leftarrow l \leftarrow k \leftarrow j \leftarrow i \leftarrow (g, h) \leftarrow f \leftarrow (e, d, e) \leftarrow b \leftarrow a\}$.\(^7\)

\(^7\)A dependency closure is a transitive relationship among actions, and each in this closure is directly or indirectly involved in data dependencies on others.
Table 6.5: The JEIS statements reverse-engineered from action sequence

<table>
<thead>
<tr>
<th>JEIS statement</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. nsIPrefBranch.getBoolPref(&quot;devtools.inspector.enabled&quot;);</td>
<td>$\Rightarrow c$</td>
</tr>
<tr>
<td>2. var nsIOService2=Components.classes[&quot;@mozilla.org/network/io-service;1&quot;].getService(Components.interfaces.nsIOService);</td>
<td>N.A.</td>
</tr>
<tr>
<td>3. var nsIURI=nsIOService2.newURI(&quot;data:text/html;base64,PHNjcmxlwd...&quot;,null,null);</td>
<td>$\Rightarrow a$</td>
</tr>
<tr>
<td>4. nsIURI.scheme();</td>
<td>$\Rightarrow b$</td>
</tr>
<tr>
<td>5. var nsILocalFile=Components.classes[&quot;@mozilla.org/file/local;1&quot;].createInstance(Components.interfaces.nsILocalFile);</td>
<td>N.A.</td>
</tr>
<tr>
<td>6. nsILocalFile.initWithPath(&quot;/usr/bin/gnome-calculator&quot;);</td>
<td>$\Rightarrow d$</td>
</tr>
<tr>
<td>7. var nsIProcess==Components.classes[&quot;@mozilla.org/process/util;1&quot;].createInstance(Components.interfaces.nsIProcess);</td>
<td>N.A.</td>
</tr>
<tr>
<td>8. nsIProcess.init(nsILocalFile);</td>
<td>$\Rightarrow e$</td>
</tr>
<tr>
<td>9. nsIProcess.run(true,[&quot;/usr/bin/gnome-calculator&quot;],1);</td>
<td>$\Rightarrow f$</td>
</tr>
</tbody>
</table>

Thus, $(g, h)$ and $(c, d, e)$ are two sets of independent actions that lead to 12 EPs—multiplied by 2 for $(g, h)$ and 6 for $(c, d, e)$.

Data dependency analysis is used to check the benignity of traces on the fly in our teacher implementation. Based on dependency closures inferred from malicious sequences, JS* identifies benign sequences not holding these closures, and find malicious EPs. Sequences $\langle a.b.d.e.f \rangle$ (removing prefix $c$ from $\pi_{m1}$) and $\langle c.a.b.d.f.e \rangle$ (partial reordering of $\pi_{m1}$), which satisfy dependency closures but are unknown for benignity, will be tested with replay mechanism as presented below.

6.5.1.3 Trace Replay

Replay mechanism is implemented to dynamically test maliciousness of an inquired trace during on-line learning. It is implemented using JS’s Equivalent Intermediate Script (JEIS), which is the intermediate code rather than the source code. The basic idea is to manually craft a JEIS with Chrome privileges, according to a given action sequence. We do not craft JEIS from scratch, but create a JEIS by adding, deleting or reordering JEIS statements from the reverse-engineered JEIS of existing training traces.
As explained at the beginning of this section, a new sequence usually has a prefix or suffix added to or deducted from a previous sequence that has been tested in a membership query. For example, during learning, a membership query checks sequence \(\langle a.b.d.e.f \rangle\) (removing prefix \(c\) from \(\pi_{m1}\)). To replay \(\langle a.b.d.e.f \rangle\), we get the reverse-engineered JEIS of \(\pi_{m1}\), then remove the intermediary script relevant to \(c\), and finally craft the expected JEIS that contains statements 2-9 in Figure 6.5. Executing this JEIS in JS engine realizes replay mechanism for sequence \(\langle a.b.d.e.f \rangle\). As replaying this JEIS produces the same result as that of running JEIS of \(\pi_{m1}\), we claim that \(\langle a.b.d.e.f \rangle\) is also malicious.

JEIS execution produces three types of outcomes. First, JEIS execution may trigger some defense rules, or some heuristic rules that detect malware in the sandbox of JS engine [86], e.g., a JEIS with system call \texttt{nsIProcess.init} whose origin is from an external website violates CAPS, as its interface \texttt{nsIProcess} cannot be executed without Chrome privilege. In case of running JEIS of \(\pi_{m1}\) and \(\langle a.b.d.e.f \rangle\), we consider they are malicious and cause potential risks. Second, JEIS execution may cause some runtime exceptions or crashes. Such exceptions or crashes indicate that the replayed sequence is infeasible rather than malicious, e.g., according to JEIS of \(\pi_{m1}\langle c.a.b.d.e.f \rangle\) in Figure 6.5, we want to craft the JEIS of \(\langle c.a.b.d.f.e \rangle\) that has a reversed order of \(e\) and \(f\). Inverting \(e\) and \(f\), and running the new crafted JEIS (i.e., the JEIS statements in the order of \(\langle 1,2,3,4,5,6,7,9,8 \rangle\)) causes an exception in the JS engine, as \texttt{nsIProcess.run} (statement 9) is executed before \texttt{nsIProcess.init} (statement 8). So \(\langle c.a.b.d.f.e \rangle\) is infeasible, and this infeasible sequence means that the outcome is benign. Lastly, if JEIS execution fails to produce any obvious resource oriented activities and violates no rules, it is assumed benign.

### 6.5.1.4 Membership Query Algorithm

Given malicious action sequences \(S_{\pi}\) and benign sequences \(S'_{\pi}\), e.g., the sequences in Figure 6.4, Algorithm 8 describes the process in answering the membership query regarding to the given action sequence \(\pi_a\).
Learning Models for Malicious JavaScript Detection

Algorithm 8: membershipQuery

Input: \( \pi_a \), a given action sequence, which cannot be null

Output: true or false, if \( \pi_a \) should be accepted by the expected DFA.

1. If \( \exists \pi_i \in S_{\pi} \cdot \pi_i \) is a prefix of \( \pi_a \) then
   - return true;
2. If \( \exists \pi'_i \in S'_{\pi} \cdot \pi_a == \pi'_i \) then
   - return false;
3. If isEqualPermutation(\( \pi_a, S_{\pi} \)) then
   - return true;
4. If \( \pi_a \) violates any defense rule then
   - return true;
5. Return Replay(\( \pi_a \));

First, if any \( \pi_i \in S_{\pi} \) is a prefix of \( \pi_a \) (line 1), i.e., \( \pi_a \) equals to or starts with \( \pi_i \), \( \pi_a \) must be malicious. If \( \pi_a \) equals any sequence in \( S'_{\pi} \) (line 3), \( \pi_a \) must be benign. Then isEqualPermutation(\( \pi_a, S_{\pi} \)) method at line 5 defined in Algorithm 7 is called to check whether \( \pi_a \) is an EP, and a true answer means that \( \pi_a \) is also a malicious EP. The next step is to check if \( \pi_a \) violates the defense rules in Section 6.5.1.1 at line 7. Finally, method Replay(\( \pi_a \)) at line 9 is called to concretely execute \( \pi_a \) using replay mechanism to verify its benignity.

In our running example, an example of satisfying the check at line 1 is action sequence \( \langle c.a.b.d.e.f.a \rangle \), which is considered as malicious as it starts with an existing malicious sequence \( \pi_{m1} \). An example for isEqualPermutation(\( \pi_a, S_{\pi} \)) at line 5 is to check action sequences \( \langle c.a.b.e.f.d \rangle \) and \( \langle a.b.c.c.d.e.f \rangle \). According to the collected traces like \( \pi_{m8} \) in Figure 6.2, we found that action \( b \) has data dependency on \( a \); \( c \) is independent; meanwhile \( e \) and \( f \) have a data dependency on \( d \). Thus, \( \langle c.a.b.e.f.d \rangle \) is not an equivalent permutation of existing malicious sequences like \( \langle c.a.b.d.e.f \rangle \). But \( \langle a.b.c.c.d.e.f \rangle \) is malicious as it satisfies the identified data dependency, similar to \( \pi_{m3} \) but with only one more independent \( c \).

At line 7, our example in Figure 6.1 does not violate any rule, as all the scripts are from the same origin, and no signed-script is used. Lastly, Replay(\( \pi_a \)) at line 9 is effective in running the left uncertain sequences ranging from simple ones \( \langle a.b \rangle \) to
those complicated ones like \( b.a.f.a.b.f.e.d.a.e.d.d.f \) for membership query. Actually, \( \text{Replay}(\pi_a) \) all returns \textit{false} for these two queries, as the two replayed sequences produce no malicious results (no resource oriented activities and no permission/rule violations).

### 6.5.2 Candidate Query

During learning, an intermediate DFA \( C \) is inferred after multiple membership queries. To judge if \( C \) is equivalent to the expected DFA \( D \) that accurately models the attack, an efficient algorithm is required for the validity check in polynomial time. When a candidate query is evaluated, two types of counterexamples can be found on \( C \)—false positives and false negatives. The former means that a sequence accepted by \( C \) should be rejected by \( D \), while the latter means that a sequence rejected by \( C \) should be accepted by \( D \).

Given \( S_\pi \) and \( S'_\pi \) in Figure 6.4, Algorithm 9 illustrates how the teacher answers the candidate query for the given \( C \), based on regression testing and random sampling testing. First, at line 1, each sequence \( \pi_i \) from the known sequence sets \( S_\pi, S'_\pi \) and the previously tested sequence set \( S_{o\pi} \) is input to \textit{membershipQuery} and also \textit{C.isAccepted}(). Method \textit{membershipQuery}(\( \pi_i \)) at line 2 returns true if \( \pi_i \) is accepted by \( D \). If \( C \) and \( D \) show different acceptance results for \( \pi_i \), a counterexample \( \pi_i \) is found and returned. Second, a random walk function \textit{randomWalks}() is used to generate \( S_{n\pi} \), a new set of random sequences that include both accepted and rejected ones on \( C \). The rationale of using random walk is that sequences on \( C \) cannot be enumerated due to potential loops. \textit{randomWalks}() at line 4 has three parameters, where \( C \) is the candidate DFA; \( C.\text{stateSize}*\text{times} \) denotes the number of generated sequences and \textit{times} is an input constant to multiply; \( C.\text{stateSize}+\text{extraLenLmt} \) is the maximum allowed length of generated sequences and \textit{extraLenLmt} is the extra length that can be larger than \( C.\text{stateSize} \). Here, \( C.\text{stateSize} \) denotes the size of total states in \( C \). Then, each sequence \( \pi_i \) in \( S_{n\pi} \) is also given to \textit{membershipQuery}() and \textit{C.isAccepted}() for acceptance check—an inconsistency indicates a counterexample. Note that any tested \( \pi_i \) from \( S_{n\pi} \) is added to \( S_{o\pi} \),
Algorithm 9: candidateQuery

input : $C \neq NULL$, the learned candidate DFA

input : $S_o$, the set of old sequences that have been tested in previous calls of
candidateQuery, initially being $\emptyset$ before any candidateQuery is called

output: $\pi_{ce}$, an counterexample sequence found

1 foreach trace $\pi_i \in (S_\pi \cup S'_\pi \cup S_o)$ do
2       if membershipQuery($\pi_i$) $\neq C$.isAccepted($\pi_i$) then
3           return $\pi_{ce}$ $\leftarrow$ $\pi_i$
4
5 $S_n$ $\leftarrow$ randomWalks($C$.stateSize $\times$ times,$C$.stateSize+$\text{extraLenLmt}$)
6 foreach trace $\pi_i \in S_n$ do
7       $S_o$ $\leftarrow$ $S_o$ $\cup \{\pi_i\}$
8       if membershipQuery($\pi_i$) $\neq C$.isAccepted($\pi_i$) then
9           return $\pi_{ce}$ $\leftarrow$ $\pi_i$

return $\pi_{ce}$ $\leftarrow$ NULL

(a)

(b)

Figure 6.6: The learned candidate DFA $C_1$ (a) and $C_2$ (b)

which is used for regression testing in answering the next candidate query. Finally, if
no counterexample is found, NULL is returned and the learning process stops.

In our running example, candidate queries are asked twice. The first candidate
DFA $C_1$ in Figure 6.6 (a) is inferred when $P$ in the observation table only contains
{$\lambda, a, b, c, d, e, f$} with one state 0. For $C_1$, Algorithm 9 returns a malicious sequence
$\pi_{m1} : \langle c.a.b.d.e.f \rangle$ as a counterexample for further learning. Afterwards, a candidate
DFA $C_2$ in Figure 6.6 (b) is learned. According to regression testing and random sam-
pling testing in Algorithm 9, no counterexample for $C_2$ is found—$C_2$ is equivalent to the
expected $D$. 

6.5.3 The Learned DFA and Refinement

Given the 16 sequences in Figure 6.4, JS* undergoes 622 membership queries and 2 candidate queries (with times=5 and extraLenLmt=5 for method randomWalks in Algorithm 9), and infers a DFA $D$ to model the attack—or equivalently a regular language $L = (c)^* \cdot a \cdot (c)^* \cdot b \cdot (c)^* \cdot d \cdot (c)^* \cdot e \cdot (c)^* \cdot f \cdot (c)^*$ over $\Sigma = \{a, b, c, d, e, f\}$. To apply this DFA to detect malicious variants of this attack, a trace from a suspicious variant is collected and preprocessed to be converted into an action sequence $\pi$ over $\Sigma$. An acceptance of $\pi$ on $D$ suggests that $\pi$ is from a malicious trace.

The sequences in $S_\pi$ in Figure 6.4 are all deterministic (being certainly malicious), and produced by the script with fixed arguments in Figure 6.1. In practice, the same sequence of actions might be sometimes malicious and sometimes benign, depending on the arguments of the calls. For instance, if the last argument of the statement `proc.run(true,[path],1)` in Figure 6.4 is changed from “1” to “0”. Executing the new code can produce the same sequences as those old ones in Figure 6.4, and all data dependencies inside these new sequences still hold. But the new sequence $\pi_m': \langle c.a.b.d.e.f \rangle$ is benign and creates no process, since inside $\pi_m'$ the last argument of action $f$ is “0”, which makes nsIProcess.run include zero argument from the argument list. One way to solve the problem is to model system calls with different arguments as different actions, for instance, we can represent the above as: $\pi_m: \langle c.a.b.d.e.f(1) \rangle$ and $\pi_m': \langle c.a.b.d.e.f(0) \rangle$. To refine the DFA for such case, we derive new actions $f_m$ and $f_b$ from current action $f$. Here, $f_m$ refers to the action $f(\{s_1,...s_n\})$ with the argument set $\{s_1,...s_n\}$ that produces malicious outcome like $f(1)$. Similarly, $f_b$ refers to the action $f$ with arguments that produce benign outcome. Then with the new $\Sigma' = \{a, b, c, d, e, f_m, f_b\}$, JS* is applied again on the training sequences to learn a refined DFA. Thus, JS* supports refinement for nondeterministic sequences in a reactive way.
6.6 Implementation

In this section, we introduce the infrastructure of JS* to capture dynamic execution traces of JS code, and the replay mechanism to execute an artificial trace in Firefox’s JS engine.

6.6.1 Capturing System Calls

Our implementation is an instrumentation to the Firefox kernel. There are three layers in Firefox’s JS execution environment (shown in Figure 6.7), i.e., the upper layer interpreter SpiderMonkey, the middle layer XPCOM and the lower layer libraries. At the upper layer, Firefox’s JS engine SpiderMonkey interprets input JS code and invokes the related lower layer libraries via XPConnect, which provides interaction between SpiderMonkey and XPCOM. Here, XPCOM (Cross Platform Component Object Model) is the middle layer and implemented by the lower layer libraries, e.g., Gecko library for page rendering, Necko library for network accessing.

As XPConnect delegates the lower layer implementation and bridges the upper and middle layer, we hook calls to methods in XPConnect interfaces as JS browser-layer system calls. The instrumented XPConnect layer intercepts the information of XPCOM
system calls at runtime. Intercepted information includes interface name, method name, argument information (e.g., type, value and memory address), return value, call mode, file name, execution time, etc.

The intercepted information of system such as call objects, arguments and return value facilitates data dependency analysis among system calls. By comparing the arguments of each system call with the return value or the related object of another system call in terms of object type and memory address, it can be checked if there exists a direct data dependency. In the implementation, we also propagate data dependency check to calculate the transitive dependency relationship for ease of data dependency analysis in Section 6.5.1.2. If the type and memory address of last system call’s return value equals that one of the next system call’s arguments, we can conclude that they have a data dependency relationship. The frequency of calling interface is high. According to one experiment, the frequency is average 879 times per minute. Therefore, interface calling information cannot be used as elementary unit of analysis. What is more, individual interface calling cannot reveal any intact function.

6.6.2 Trace Replay

As Section 6.5.1 explains, trace replay mechanism is implemented to dynamically test maliciousness of a given trace, which is similar but different (in the order, prefix or suffix of its system call sequence) from an existing or tested trace.

We implement replay mechanism as an instrumentation to Firefox to attain Chrome privilege. The artificial system call trace to be replayed is not conjectured up. Instead,
the required order of system calls is guided by the $L^*$ algorithm. A new sequence usually has a prefix or suffix added to or deducted from a previous trace that has been tested in a membership query. For example, during the $L^*$ learning, a membership query asks the maliciousness of an action sequence $\langle a.b.d.e.f \rangle$, which is similar to $\pi_{m1} : \langle c.a.b.d.e.f \rangle$ but without the prefix $c$. To replay this sequence and check its maliciousness, we use a reversed-engineered JS trace, called JS’s Equivalent Intermediate Script (JEIS). We get the JEIS of $\pi_{m1}$ that is shown in Figure 6.5, then remove the intermediary script irrelevant to $c$, and finally craft the expected JEIS shown in Figure 6.8. Executing this JEIS by the JS engine realizes the functionality of replaying the action sequence $\langle a.b.d.e.f \rangle$. Actually, as executing this JEIS leads to the same result of executing the JEIS of $\pi_{m1}$, $\langle a.b.d.e.f \rangle$ is also malicious.

6.6.2.1 Running JEIS with Arguments

An interface in Mozilla is a definition of a set of functionality that could be implemented by components. Each component implements the functionality as described by interfaces. They can be referred to using a string, e.g., ‘@mozilla.org/intl/texttosuburi;1’. This string is called a contract ID. To replay one single system call, we need to retrieve a component corresponding to its interface name. The contract ID of the component can be used to get the component; and we maintain a mapping table between contract ID and interface name. It is also can be attained from Mozilla Developer Network [9].

A challenge in replaying a trace is how to assign the proper arguments for system calls involved in the JEIS trace. Each system call instance must specify its interface name, method name, the number and values of arguments and the type of return value, which are mandatory elements for a system call and also prerequisites of replay. As the elements of a system call is stored in our infrastructure, the arguments can be reused when this system call is replayed. The primitive argument types can be reused without any difficulty, but for the object type argument our infrastructure has direct memory access to capture the address and the length of that object for reuse. Note that our replay mechanism generally supports the JEIS with arbitrary arguments.
F I G U R E 6.9: The JEIS for the benign action sequence \( \langle a.b.c.d.f.e \rangle \)

### 6.6.2.2 Execution Results of JEIS

The execution of JEIS outputs three types of outcomes. First, the execution of JEIS triggers some defense rules, or violates some heuristic rules that detect malware in the sandbox of JS engine [86], e.g., a JEIS with system call `nsIProcess.init` whose origin is from an external website violates CAPS, as its interface `nsIProcess` cannot be executed without Chrome privilege. In such case, we consider it as a malicious trace which causes potential risks. Second, the execution of JEIS may also cause a runtime exception or a crash. Rather than considering the replayed trace as malicious, the exception or crash actually indicates that the replayed trace is infeasible, e.g., according to \( \pi_{m3} : \langle a.b.c.d.e.f \rangle \), we want to craft the JEIS of \( \langle a.b.c.d.f.e \rangle \) that has a reversed order of \( e \) and \( f \), shown in Figure 6.9. Inverting \( e \) and \( f \) causes an unpredictable error for the JS engine, as `nsIProcess.run` is executed before `nsIProcess.init`. So \( \langle a.b.c.d.f.e \rangle \) is infeasible, and this infeasible sequence means that the outcome is benign. Lastly, if the execution of the JEIS trace fails to lead to the same result as that of malicious traces and violates no rules. We consider it is benign.

JEIS must be executed in Chrome privilege. One way to obtain Chrome privilege is to execute as an extension to browser. Therefore, to ensure the polynomial time complexity of \( L^* \) algorithm, we modify our installed extension each replay. Generally speaking, installation of new extension requires restart of browser.
6.7 Evaluation

In this section, we evaluate our approach with real-world JS malware samples to answer the following research questions:

RQ1 Can the learned attack behavior models correctly and effectively describe common attack behaviors?

RQ2 Is our approach efficient for learning attack behavior models?

RQ3 Is our approach efficient for detection of known malwares?

RQ4 Are the models useful for detection of emerging malicious variants and new attacks?

6.7.1 Settings

Eight popular types of JS attacks mentioned in Section 6.2 are used for evaluation. We collect 276 unique malicious JS programs from more than 10,000 real-world malwares by removing duplicated and outdated ones. These malwares come from the following sources: 40 from VXHEAVEN [3], 66 from OPENMALWARE [2], and 170 from the recent list of malicious websites reported by WEB INSPECTOR [11].

We manually inspect these 276 samples to verify their maliciousness by running these malicious JavaScripts together with their desired host software (e.g., browser, Adobe reader) in a sandbox. We confirm their maliciousness by check the results they took on the sandbox browser. Besides, we collect 10,000 benign JSs from the Alexa [1] top 100 web sites, none of which is reported as malicious by the 56 tools provided by VIRUSTOTAL.\(^8\)

120 out of 276 (40% and 15 for each attack type) malicious JSs are used as the training set to learn attack behavior models. We run each training example 10 times to generate 10 traces, most of which differ from each other due to different browser

---

\(^8\)To our best knowledge, JSAND is the only open service for JS malware detection, and VIRUSTOTAL is powered by updated versions of mainstream anti-virus products.
contexts at the time of execution. These traces are converted to action sequences with the filtering step. Among 8 different attack types, at least 37% (for Type V) to at most 60% (for Type VII) of traces are unique. We apply the proposed approach for each attack type to learn 8 attack behavior models in total. To evaluate the performance of the learned models, we use the remaining 156 malicious samples and the 10,000 benign samples as the validation set.

All the results are collected on a 64-bit Ubuntu/Linux 12.04 PC running Firefox 17.0 with Intel i7 2600 3.4GHz CPU and 8GB memory.

### 6.7.2 Attack Models Learning

The statistics about the learning process and the generated attack behavior models are collected in Table 6.1. Column $|S|$ shows the number of the states of learned attack behavior models; Column $|\Sigma|$ shows the number of common actions used for learning; Column $|Q_m|$ shows the number of membership queries; Column $|Q_c|$ shows the number of candidate queries; Column Time (s) shows the time in seconds used by the $L^*$ algorithm; Column Total Time (s) shows the total time in seconds of the whole learning process including trace generation and trace replay.

**RQ1: Correctness.** We validate the correctness of learned attack behavior models from two aspects: (1) identifying the high-level semantics by checking their alphabets, and (2) comparing the languages of attack behavior models with the descriptions of the CVEs used by these attacks.
First, checking the relatively small set of common actions $\Sigma$ briefly tells whether common essential behaviors of the same attack type are captured and modelled. The set of common actions ($\Sigma$) for each attack type is reasonable and relevant to the attack type. For instance, Type I attack can be generally divided into three steps: 1) put the shell code in a predictable memory location; 2) trigger an exploitable crash (modelled by common actions like `nsIAppStartup.trackStartupCrashEnd` and others in $\Sigma$ of Type I); 3) the shell code will be executed to perform the attack, invoking system calls related to file operations (e.g., `nsIFile.append`).

Second, we check accepted traces of Type VI attack DFA and identify five steps: downloading malicious pdf file, executing embedded JS to scan vulnerability, exploiting the vulnerability, executing payload, and actual sabotage.

Some behaviors (e.g., payloads executing) are shared by all eight attack types. Payloads include executing arbitrary command, binding shell or reversed shell using TCP. In Figure 6.10, we illustrate the common behaviors of binding shell or reversed shell using TCP, after the exploit for each corresponding attack is done.

The unique parts of an inferred DFA captures the essential attack behaviors of the corresponding attack type. In Figure 6.11, we show a fraction of the attack behavior model of Type I attack. This fraction exploits the CVE-2013-1710 vulnerability. Specifically, this attack invokes system call `crypto.generateCRMFRRequest` to enable...
remote attackers to execute arbitrary JavaScript code or conduct cross-site scripting (XSS) attacks via vectors related to Certificate Request Message Format (CRMF) request generation. The original trace contains 1,236 system calls and has 12 equivalent permutations with the same set of actions. However, with our JS* learning approach, the DFA in Figure 6.11 contains only 14 actions and 19 states. Thus, our inferred attack behavior models in form of DFA are concise yet accurate, without loss of the essence of attack.

A notable case is that the DFA of malicious redirecting attack (Type VII) has 466 states. A possible explanation is that samples of Type VII attack used for learning are less similar than samples of other attack types. Thus, 15 dissimilar samples infer a DFA with a small size of alphabet but a large number of states, which does not necessarily mean a bad modelling result. On contrary, it suggests that many traces are accepted by this DFA—more possible variants of this attack type. Such explanation is backed up by the fact that malicious redirecting attacks are simple with a small set of common behaviors (|Σ|=15), but flexible with possibly enormous variants: 986 out of totally
1,300 malicious JS samples provided by VXHEAVEN are drive-by-download attacks that generally relate to Type VII attack.

**Effectiveness.** Besides manual observation, we empirically validate the usefulness of the DFAs by conducting predication. In the validation set of 10,000 benign and 156 malicious samples, each sample is executed 10 times to get different traces. If any trace is accepted by one of eight learned DFAs, the corresponding sample is detected as one variant of the attack that is modelled by the matched DFA. Totally, JS\(^*\) correctly detects 149 out of 156 (95.51\%) malicious samples and 9,957 out of 10,000 (99.57\%) benign samples. The distributions of 7 false negative cases (4.49\%) and 54 false positive cases involving 43 distinct benign samples (0.43\%) are listed in column JS\(^*\) FN and JS\(^*\) FP of Table 6.2\(^9\), respectively. Among 43 distinct benign samples, 11 (54-43) are falsely accepted by two DFAs, e.g., 2 benign samples are accepted by DFAs of both Type VII and VIII, which share commonality—toolkits based attacks utilize abnormal redirection iteratively to evade detection. Thus, JS\(^*\) attains low overall false negative rate (≈5\%) and false positive rate (≈0.5\%) for all 8 attack types.

**RQ2: Performance.** The results reported in column Time (s) of Table 6.1 show that our approach is highly scalable in the core learning process. The required learning time is generally proportional to the state number and the alphabetical size of the learned DFA. Column Total Time (s) includes the time used for trace generation.

---

\(^9\)The reason to put Total at the first row is that there is no type-based detection in Jsand or the 56 tools on VirusTotal. We separately list the 8 type-specific rows to show how attack types affect detection results.
and replay, which is the major overheads for dynamic approaches. As reported by Rieck et al. [162], CUJO takes 500 ms in average to analyze a webpage for dynamic feature extraction. JS* in average takes 1 second to generate or replay one trace, for a given script snippet.

Owning to the step of preprocessing, a small alphabet can be built from the execution traces by filtering security-irrelevant system calls out and extracting common actions from traces. Alphabetical sizes (column $|\Sigma|$) of the 8 learned DFAs are all less than 30, and 5 out of 84 DFAs even have $|\Sigma|$ less than 16. Usually, a small value of $|\Sigma|$ leads to a small number of raised membership queries and candidate queries, e.g., DFAs with $|\Sigma| \leq 11$ have $|Q_m| \leq 42K$ and $|Q_c| \leq 12$. For these DFAs with $|\Sigma| \leq 11$, the core learning process can be accomplished in 5 seconds. For other types except Type VII, it takes only less than 9 seconds. The most time-consuming one is for Type VII. Considering large values of $|Q_m|$ and $|Q_c|$, it is fast to finish core learning in 22 seconds. It is also acceptable to finish all, including trace generation and replay, in 4597 seconds.

Another observation is that JS* requires a large value of $|Q_m|$ but a small value of $|Q_c|$. The explanation is that action sequences in the training set are quite different from each other in terms of length or substring. In contrast, sequences $\pi_{m1}$ to $\pi_{m6}$ and $\pi_{b1}$ to $\pi_{b6}$ in Figure 6.4 show high similarity in length or substring, with different positions of action $c$. These similar sequences in our running example quickly lead to a closed and consistent observation table, which makes $|Q_m| = 622$ and $|Q_c| = 2$. However, for these 8 real attack types, there are no such ideally similar sequences that lead to quick convergence to a closed and consistent observation table. Thus, it usually needs a large number of membership queries to reach a candidate DFA.

**RQ3: Tool comparison.** We compare JS* with the open JS malware detection service JSAND 2.3.6 [6,62] and VIRUSTOTAL [10], an online malware detection service powered by 56 mainstream anti-virus products. The comparison mainly focuses on false negative rates and average predication time. We do not compare false positive rates, as all 10,000 benign samples are verified by JSAND and 56 tools on VIRUSTOTAL, i.e., no false positive case in benign samples for JSAND and 56 tools on VIRUSTOTAL.
Table 6.3: Detection ratio of our approach and other tools on 156 malicious samples in the testing data set

<table>
<thead>
<tr>
<th>Tool</th>
<th>Detection %</th>
<th>Tool</th>
<th>Detection %</th>
</tr>
</thead>
<tbody>
<tr>
<td>JS*</td>
<td>95.51%</td>
<td>MCAFEE</td>
<td>57.05%</td>
</tr>
<tr>
<td>AVAST!</td>
<td>81.41%</td>
<td>JSAND</td>
<td>39.10%</td>
</tr>
<tr>
<td>GDATA</td>
<td>73.72%</td>
<td>TREND</td>
<td>30.77%</td>
</tr>
<tr>
<td>AVG</td>
<td>73.08%</td>
<td>SYMANTEC</td>
<td>28.21%</td>
</tr>
<tr>
<td>BITDEFENDER</td>
<td>71.15%</td>
<td>CLAMAV</td>
<td>12.82%</td>
</tr>
<tr>
<td>F-SECURE</td>
<td>69.87%</td>
<td>PANDA</td>
<td>1.92%</td>
</tr>
<tr>
<td>KASPERSKY</td>
<td>67.95%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In total, JSAND\(^{10}\) correctly detects 39.1% (61/156) malicious samples, and 95 false negative cases are not evenly distributed among the 8 types (see Table 6.2). Among the 8 attack types, JSAND yields the lowest false negative rate (8.7%) for Type VI, and the highest false negative rate (80%) for Type I and Type II attack. As JSAND is a dynamic detection tool, due to the limit of the used honeypot, it may miss samples from Type I and Type II, which are platform specific and not easy to trigger.

Column VT.\(^{11}\) in Table 6.2 denotes the average number of tools that detect each malicious sample, among 56 tools provided by VIRUS TOTAL. For each of 156 malicious samples, on average 21 (37.5%) of 56 tools can successfully detect it. We also observe that on average 34 (60.7%) of 56 tools can detect each of Type VI samples. This observation indicates that 56 tools on VIRUS TOTAL can generally better detect Type VI attacks than others. This observation is consistent with previous finding that JSAND has the lowest false negative rate (8.7%) for Type VI. Among the 8 types, in average only 15 tools (26.8%) can detect each of Type VII attacks. Thus, Type VII attack is difficult to detect for state-of-the-art tools, due to its flexible attack behaviors and enormous variants. This point is supported by the complexity of the inferred DFA of Type VII.

To see the capabilities of state-of-the-art tools, in Table 6.3, we test the detection ratio of the open-source anti-virus tool CLAMAV [13] and 2014 best reviewed anti-virus products [14]: AVG, AVAST!, BITDEFENDER, F-SECURE, GDATA, KASPERSKY, MCAFEE, PAN-DA, SYMANTEC, and TRENDMICRO. For the validation set of 156

\(^{10}\) As JSAND uses dynamic analysis, we submitted each sample ten times. If any submission reports that the sample is malicious or suspicious, we consider it malicious.

\(^{11}\) We also submitted each samples to VIRUS TOTAL five times in December 2014, and the results reported by VIRUS TOTAL were consistent for different submissions.
malicious samples, the best tool AVAST! achieves a detection ratio of 81.41%. Other tools perform even worse. We manually inspect false negative cases for JS* and other tools. One sample that belongs to Type VII is missed by both JS* and VIRUSTOTAL, as it targets at mobile platform and fails to launch the attack in our testing environment.

According to the JSAND’s report, we calculate the predication time by deducting the analysis starting time from the reported generation time. In average, it takes 4.7 seconds for JSAND to finish the execution and predication of one sample (see column JSAND T(s) in Table 6.2). In contrast, JS* takes 1.36 seconds in average to execute the tested sample and check the trace with 8 learned DFAs. Thus, the predication time was reduced by 71% in JS*. For predication time of the 56 tools, VIRUSTOTAL runs them in parallel and sets a timeout 1 minute to receive responses. As results from different tools are dynamically added to the result page, according to our observations, most tools can finish the predication in 5–10 seconds.

**RQ4: Detecting variants.** The predication results show the capability of JS* in detecting malicious variants of the same attack type, as the 120 training samples used for learning share quite low textual similarity with the 156 testing samples. We use code clone detection tool CloneMiner [30] and fail to detect sample pairs that have file-level textual similarity above 30% (due to different exploits and obfuscation). Thus, owing to the nature of dynamic analysis of JS*, syntactic obfuscation poses no challenge to JS*.

**Detecting new attacks.** Besides the DFAs, we also test the usefulness of the partial DFAs. Among the 156 testing samples, a new attack that exploits CVE-2014-1580 is missed by JSAND and other 56 tools. Our 8 inferred DFAs also fail to accept its traces. However, the partial DFA in Figure 6.10 can detect it, as the payload execution part of its traces is accepted by the DFA. Thus, the partial DFAs can also be used for fine-grained behavior identification. In this case, payload execution helps to detect the new attack, but cannot classify according to the given exploit types. Last, based on the partial DFAs, we manually craft 8 malicious samples for each attack types, by combining different exploits and payloads. These new crafted ones can fail existing tools, e.g., in average 23 tools on VIRUSTOTAL can detect the crafted ones, while 33 tools detect their original samples in average.
6.7.3 Discussion

Behavior model or stateful behavior model? To sum up, the experiments have shown the effectiveness of JS* in modelling attack behaviors and predicing variants. JS* generally works well for malicious attacks with clear and representative common attack actions, e.g., *Attack targeting JRE* and *Attack based on multimedia* (e.g., images, videos). According to our controlled experiments, the traces generated by a handful of unique and representative samples can help JS* to efficiently infer a DFA. Currently, we only use the arguments of system calls for data analysis, not for argument based behavior modelling. The rationale is twofold: (1) We model JS attack based on system calls, not JS APIs. System calls work at the lower level than JS APIs. The same code snippet (with the same APIs) leads to different low level system call traces. Thus, system call traces essentially are reflection of APIs together their arguments. (2) In reality, these 276 malicious samples mostly can launch the attack with the fixed built-in arguments, without the interaction or input from users. In chapter 3, we show that behavior modelling of APIs together with their arguments requires stateful behavior models (behavior models with guard conditions for transitions). A stateful behavior model is good at capturing behaviors of simple data-rich program. However, in this chapter, considering the size of alphabet, inferring stateful behavior models is not scalable but prone to path explosion.

DFA decomposition. Currently, we model major behaviors of an attack type in one DFA, including vulnerability exploiting and payloads execution. For each step, there might be several different behavior patterns, e.g., different payloads executions in Figure 6.1 (executing arbitrary code in victim computer) and Figure 6.10 (binding shell using TCP). We initially manually identify such patterns from one DFA. Then we convert patterns and other DFAs into directed cycle graphs, and then we apply GENERICDIFF [192] to do graph matching — checking if these patterns exist in other DFAs. We also extent GENERICDIFF for sub-graph isomorphism to extract possible patterns (similar partial DFAs) from 8 inferred DFAs. In this study, we treat attack behavior modelling in a top-down way — tracking the whole trace of an attack to model and
classify. In other study on mining API usage patterns [142], behavior modelling is done in a bottom up way.

6.7.4 Threats to Validity

There are several threats to the validity our approach. First, the inferred models are based on dozens of representative variant traces confirmed by VXHEAVEN, OPENMALWARE, and WEB INSPECTOR. The collected traces directly affect the learning results. To address this problem, we need to further investigate the impact of sample size and representativeness on the results. Second, the parameters used for the Levenshtein distance similarity in Section 6.4 are commonly used in code clone similarity analysis [108], i.e., a threshold of 80% similarity in our study. Parameters for randomWalks in Algorithm 9 are the same as those used in Chapter 3. Further investigation is needed to see the effects of parameters. Lastly, the external validity is that the results are observed from the 8 specific JS attack types. Due to the distinctive natures of different JS attacks, the results of this study may not be applicable to malware whose malicious behavior are not reflected by system call invocations. We remark that this is one assumption of this work.

6.8 Summary

In this chapter, we propose an automatic approach to detecting JS malwares via automatic learning of the attack behavior models. The learned attack behavior models are used to detect malware variants of the same attack type and even new attack types. The key contribution is to combine static analyses (defense rules) with dynamic analysis (data dependency analysis and replay mechanism) to check whether a sequence of browser-level system calls is malicious or not in an efficient way. Eight popular types of attacks are used to evaluate the proposed approach. The results show that our approach is scalable and effective for detecting JS malwares.

System-call based behavior modelling and malware detection. Existing studies [104] [165] [187] have utilized system calls in dynamic analysis to detect abnormal
behaviors, such as attacks or intrusions. However, these studies failed to model program behaviors, until Sekar et al. [167] modelled normal executions (system call sequences) of a process as a Finite State Automaton when most normal traces are available. [167] uses a heuristic algorithm to model normal behaviors rather than learning using the $L^*$ algorithm, and the modelling process is not in polynomial time. Kolbitsch et al. [103] modelled program behavior using a direct acyclic graph of system calls along with their data-dependent parameters, and applied graph matching algorithm on these graphs to identify abnormal ones. Christodorescu et al. [55] further defined a new graph representation of program behavior and applied a machine learning algorithm to mine malware specifications from dependence graphs of the malicious and benign programs. Besides, layered system call graph in [130] and tainted argument analysis for selected calls in [172] are proposed for malware detection.

A recent quantitative study [45] investigated how the choice of system-call based behavior modelling influences the quality of detection results. They found that $n$-gram is the best model for low-level system calls, whereas bags and tuples without order information yield best results when high-level actions are introduced. However, in [45], other models (e.g., graph or DFA) are not discussed and the authors acknowledged the impossibility of generalizing observed results in a closed form. $JS^*$ uses action sequences to model attack behaviors in the form of DFA. Although orders of actions matter in the DFA, the DFA learned by $JS^*$ accepts the equivalent action sequences derived from the corresponding tuples or bags of actions, owning to the analysis on EPs of action sequences.

**Behavior model inference with $L^*$.** Several studies adopt the $L^*$ to infer behavior models in the security community but not attack behaviors. Cho et al. [53] proposed to infer botnet command and control protocols from the sequence of messages sent over the network by using the $L^*$ algorithm. Cho et al. [52] also presented an approach to infer an abstract model of the analyzed application in the form of DFA, and then to apply symbolic execution for bounded state-space exploration by virtue of the guidance provided by the inferred DFA. Cho et al. [52] infers protocol. If a message exchange trace violates the protocol, the given trace is not accepted. In our study, if malicious results are produced or defense rules are violated, the tested action sequence is accepted.
Candidate queries can be answered in three ways. First, the expected DFA to be learned is available, e.g., the existing protocol in [52]. Second, random sampling is used to generate traces to test the equivalence between a candidate DFA and the expected DFA.

**JavaScript malware detection.** Existing studies on detection of malicious JS mainly adopt static analyses, or dynamic analyses, or hybrid analyses to identify the characteristics of malicious JS. JSand [62] extracted features from 4 aspects (redirection, de-obfuscation, environmental context, and exploitation) via dynamic analyses, and used Naïve Bayes to detect JS malware. Canali et al. [44] proposed to perform a large-scale static analysis to identify the malicious web pages by applying a fast and reliable filter PROPHILER. Curtsinger et al. [63] presented ZOZZLE, a tool that predicates the benignity or maliciousness of JS code by using features associated with Abstract Syntax Tree (AST) hierarchy information. REVOLVER [100] also heavily relies on static analysis to build the ASTs and to compute the similarity among ASTs. CUJO [162] uses hybrid analysis in an on-the-fly way to extract dynamic and static features from program information and execution traces of JS programs, respectively. All extracted features are processed by *n*-grams for SVM based classification. These studies focus on detection of general JS malware. Other existing studies rely on dynamic analysis to detect specific attacks [46,73,120,160,182]. Compared with these tools, our approach can model attack behaviors. It does not require a large-size training set and can be generally applicable to attacks with explicit browser-level system calls (actions).
Chapter 7

Learning Models for Hardware-assisted Malware Defense

This chapter continues the applications of learning attack behavior models for malware detection. In particular, it is devoted to learning behavior models of conventional malware in terms of the system calls the malware invoke. The learned behavior model is then encoded in FPGA to perform online detection of malware variants.

7.1 Introduction

In last chapter, we uses attack behavior models to detect and classify malicious JavaScript programs. Compared with other dynamic approaches [76, 78, 140], attack behavior models based approaches incur low performance and resource overhead and can be used to detect malware variants. The approach in last chapter is designed to be a general framework for detecting JavaScript malwares and is implemented in software. One major drawback of software based solutions is that they can be bypassed or disabled with advanced techniques. Integrating anti-debugger and anti-ptrace techniques in malware is a widely accepted practice to evade software based solutions. Moreover, kernel-level rootkits and bootkits can gain OS level (Ring 0) privilege to disable the detection software. Recently, hardware-assisted techniques have been recognized as a promising approach to defend against malware for two major reasons: (1) detection and
In this chapter, we propose a hybrid approach, as shown in Figure 7.1 for malware defense at low performance and resource overhead. Our approach consists of two phases: software based learning phase and hardware-assisted detection/defense phase. In the learning phase, we use a software approach to learn the attack behavior models of known malware in terms of the system calls used by the malware. Thus, the attack behavior model captures the common sequences of system calls used by malwares with the same attack type to launch attacks. We also adopt the $L^*$ algorithm [24,163] to learn the behavior models. Different from the approach in last chapter, this approach uses a static analysis instead of dynamic analysis to check whether a given sequence of system calls can achieve a malicious behavior. During hardware-assisted detection/defense phase, the host hardware system is augmented with a hardware based on Field-Programmable Gate Array (FPGA) which uses the learned behavior models to check whether the execution traces of a suspicious software are malicious or not. Our hardware-enhanced architecture is named as Malguard (malware guard). The proposed approach solves the inflexibility challenge of hardware-assisted implementations. When enough new malwares emerged, we only need to perform the learning phase to learn a new or an updated attack behavior model and load the learned model into FPGA. Both software and FPGA are amenable for changes, and therefore our approach is flexible for changes.
We evaluate our approach with 168 real-world Linux malwares and 370 benign executables. Our evaluation shows that: (1) Malguard can detect malware and its variants with high accuracy; (2) it is able to classify malware samples based on their attack behavior and detect zero-day malware with similar attack patterns; (3) it detects malicious behavior in real time (within 5 clock cycles), which implies that it can possibly prevent the system from major damages caused by the malware; and (4) it has a small footprint.

The rest of this chapter is organized as follows. Section 7.2 describes the overview of our approach. The attack behaviors modelling and learning is explained in Section 7.3 and Section 7.4, respectively. We discuss the hardware design in Section 7.5. We present our evaluation results in Section 7.6. In Section 7.7, we discuss the related work and conclude this chapter.

## 7.2 Approach Overview

In this section, first we describe our approach in detail. Then we discuss the performance of our approach. As shown in Figure 7.1, the inputs for our approach are a set of malwares (*.elf files) in a specific attack type, we first run these malwares in sandboxes to collect the runtime execution traces. The collected traces are then preprocessed and analyzed to extract precedence rules between various security-related system calls. These traces and precedence rules are used by the $L^*$ algorithm to check whether a sequence of system calls (generated by the $L^*$ algorithm) is malicious or not and to answer membership queries. The software based learner can learn an attack behavior model for malwares with this attack type. The learned attack model is in essence a DFA. Thus the transition table corresponding to the learned attack behavior model is encoded in the FPGA of our hardware-assisted detector. Given the attack behavior models of known malware types, we monitor the execution traces of a suspicious executable. The traces are processed in the same way as in the learning phase. Then the problem of checking whether an execution trace is malicious or not becomes a language recognition problem: if the processed execution trace is accepted by one of the attack behavior models, then the suspicious executable is classified as a malware.
7.3 Attack Behavior Models

We build DFAs for the following seven popular attack behaviors that are observed in our malware dataset:

I. Virus replication behavior – Virus replicates itself by first copying its binary into the temporary/executable file and finally executing the file.

II. Overloading system memory – Some malware repetitively executes system calls to increase the program memory to overload the main memory. This often results in OS crashing and performance degradation.

III. Scanning ports and executing shell commands – This malicious behavior includes scanning ports and receiving malicious payloads from internet (e.g., to execute shell commands).

IV. Data leakage on the network – This attack model captures the malicious behavior that leaks user information (e.g., browser cookies) from the host onto the network.

V. Spawning – A typical behavior of virus is to replicate itself by creating several clones (child processes) from its own binary.

VI. CPU resource overloading – This attack intends to control the CPU usage by performing anomalous activities, e.g., modifying system files, creating harmful files. This makes the system extremely slow leading to CPU-starvation attacks.

VII. Killing other running processes – It subsumes the malicious behavior that intends to kill other running processes. Typically, malware first changes the default actions for the given signal (e.g., interrupt) and then kills running processes, which is not noticed by OS.

Although DFA can accurately model the malicious behavior, DFA-based detection usually incurs false positives due to the fact that some malware executes similar system call patterns as the benign programs while their maliciousness is resulted from their repetitive pattern. For example, the `brk` system call is executed by benign program to
increase the program data segment size, while “Brk.c” repetitively uses \texttt{brk} to overload the system memory, which may lead to CPU starvation attacks. Another malware “Spy-eye collector” sends data from host system to the network repetitively. Hence, for such cases, DFA model alone can result in high false positives (as shown in Section 7.6.1). In order to consider such malicious behaviors (attack type II and VI), we combine DFA with frequency to model the repetitive pattern. The threshold frequency at which a behavior should be considered malicious is set empirically by observing the differences between the malicious and benign traces.

### 7.4 Behavior Models Learning

We use the $L^*$ algorithm [24, 163] to learn the DFA models for describing the attack behaviors of malware. For malware detection in our context, we use the set of system calls as the alphabet symbols $\Sigma$ in $L^*$. To answer a membership query, the teacher checks whether a given trace of system calls is malicious or not; to answer candidate query with a given candidate DFA $C$, the teacher checks whether $C$ describes the malicious behavior. The only inputs we have are a set of malicious traces and a set of benign traces collected by running the malware samples in a sandbox. However, it is highly possible that the strings (i.e., traces of system calls) asked by $L^*$ for membership queries are not in the input traces. Thus, the teacher cannot answer all membership queries by using the input traces only. There are two issues related to answering candidate queries: 1) the teacher only knows a set of traces for malware and benign programs and it also does not know the DFA model for attack behavior; 2) the teacher needs to provide a counterexample if the candidate DFA is not good enough.

We use pairs of precedence rules in the form $(a, b)$, where $a, b \in \Sigma$ and $a \neq b$, to solve above issues in answering membership queries and candidate queries. The set of rules are generated automatically from input traces for each attack type using Algorithm 10. For each attack type, we use its malicious traces as the $MSet$ and a set of benign traces as the $BSet$. For each malicious trace, we collect all pairs of consecutive system calls into $MPSet$ and collect all pairs of consecutive system calls from each benign trace into $BPSet$. The elements that are in $MPSet$ but not in $BPSet$ (i.e., the
Algorithm 10: Generate precedence rules

**Input:** MSet: a set of malicious traces; BSet: a set of benign traces
**Output:** PSet: a set of precedence rules

1. Set $MPSet := \varnothing$
2. Set $BPSet := \varnothing$
3. for $tr \in MSet$ do
   4. for $index \in [0 \cdots tr.size - 2]$ do
   5. String $a := tr.valueAt(index)$
   6. String $b := tr.valueAt(index + 1)$
   7. $MPSet := MPSet \cup \{(a, b)\}$
4. for $tr \in BSet$ do
   5. for $index \in [0 \cdots tr.size - 2]$ do
   6. String $a := tr.valueAt(index)$
   7. String $b := tr.valueAt(index + 1)$
   8. $BPSet := BPSet \cup \{(a, b)\}$
5. $PSet := MPSet - BPSet$

Set difference between $MPSet$ and $BPSet$ are considered as the precedence rules for a particular attack.

7.4.1 Membership Query

We use precedence rules to answer membership queries. For a membership query with trace $tr$, we use the set of precedence rules as a filter to check whether $tr$ is malicious or not and answer the membership query accordingly. If the trace violates any of the precedence rules for this attack type, we return $false$, i.e., the trace is not malicious. If $tr$ passes all the precedence rules, then $tr$ is considered as malicious and the answer is $true$. Note that a trace $tr$ passes a precedence rule $(a, b)$ only if $tr$ contains both $a$ and $b$, and $b$ comes after $a$ in $tr$.

7.4.2 Candidate Query

For answering candidate queries, we use the benign and malicious traces to check whether there is any inconsistency between the traces and the candidate DFA $C$. For a benign trace $tr$, we run the trace on the candidate DFA $C$ and check whether the final
Chapter 7. Learning Models for Hardware-assisted Malware Defense

state of running $tr$ on $C$ is an accepting state or not, i.e., $tr \in C$?. If $tr \in C$, then there is no inconsistency; however, if $tr \notin C$, then there is an inconsistency. This is because according to our available input traces, $tr$ is benign, but according to the candidate, $tr$ is malicious. In this case, we return $false$ with $tr$ as the counterexample. For a trace $tr'$ in the malicious input traces, we run it on DFA $C$ and check if $tr' \in C$. If $tr' \in C$, no inconsistency is found; however, if $tr' \notin C$, then we return $false$ with $tr'$ as the counterexample trace.

Since the number of collected malware samples for each attack type is small and thus the number the malicious traces for each attack type is also small. This small number of malicious traces makes it often inadequate to check the candidate DFA only against the input traces. We also generate test traces with different combinations of the APIs up to specific lengths (different for different attack types) and check them against the set of rules and the candidate DFA to find any inconsistency. The number of possible traces is exponential to the trace length. To alleviate this problem, we calculate the transitive closure of the set of precedence rules. During the generation of test traces, we discard those traces which violate any precedence rule in the transitive closure explicitly. A trace $tr$ violates a precedence rule $(a, b)$ explicitly if $tr$ contains both $a$ and $b$ but $b$ occurs before $a$ in $tr$. We check each test trace against the filters and also run the trace on the candidate DFA. If the results of running the trace on DFA and the filters are different, we return the trace as the counterexample. If no inconsistency is found in the above two steps, we return $true$ to the candidate query. Then the final candidate DFA is equivalent to the candidate DFA $C$.

7.5 Microarchitecture Design

In this section, we present the architectural design of Malguard and discuss its design considerations. As shown in Figure 7.2, Malguard runs in parallel to the processor. For the proof of concept, we implement the Malguard design in FPGA. It can also be implemented in ASIC platform. FPGA supports flexible update of the hardware verification logic by utilizing its reconfigurability feature. Malguard is integrated with the commit stage of the processor. We assume that the processor and OS are trustworthy but the user
applications may be malicious. Our platform consists of 32-bit x86 processor running Linux. Henceforth, some details are platform specific, however, our design can also be implemented on other platform with change in specification.

Our design consists of three main components: sysbin, buffer and DFA checker as shown in Figure 7.2.

### 7.5.1 System Call Bin (sysbin)

Sysbin is integrated with the commit stage of the processor pipeline. In the commit stage, sysbin traces the software trap instruction (`int 0x80` in x86) from the reorder buffer. At trap instruction, the sysbin reads the system call type from the `eax` register, as done by Das et al. [68]. The system call is pushed into the buffer (Figure 7.2), which is used by the DFA checker later. In order to differentiate the system call traces obtained from various processes (for multitasking support), we trace the `CR3` control register. Sysbin requires minor modification in the processor design, as described in [68].

### 7.5.2 Buffer

Buffer is an important design consideration to reduce the overhead on the processor. As the DFA checker takes a few cycles to complete its operation, Buffer is used to avoid
the processor stalls and reduce the performance penalty.

### 7.5.3 DFA Checker

It obtains system calls from the Buffer and detects malicious behavior by validating the transition of states. As shown in Figure 7.2, it consists of a transition table and a control logic to detect the malicious behavior. The transition table stores all the state transitions, which are obtained from the offline learning phase. As discussed in Section 7.2, we use a frequency parameter to differentiate between benign and malicious traces for a few attack patterns (Type II and VI). Based on the system call \((sys)\) and current state \((CS)\), the next state \((NS)\) is obtained from the transition table. Parallel comparators compare \(NS\) with the accepting states of the DFAs (stored in the FPGA registers). If \(NS\) is an accepting state, the counter is incremented once and the corresponding threshold frequency is obtained from threshold table (THT). Then frequency comparator compares the threshold frequency and the counter value. For those attack behaviors which do not have the repetitive pattern, we store its threshold frequency as 1. Once the threshold frequency matches to the counter value, we flag the program as malicious.

Our design can handle multiple DFAs. The \(CS\) of each DFA is initialized to its initial state. During checking, we check whether \(sys\) matches with each DFA on its \(CS\). If there is a transition \((CS,\ sys,\ NS)\) in the DFA, we change its \(CS\) to \(NS\); if there is no such transition, the \(CS\) of the DFA is not changed.

### 7.6 Evaluation

#### 7.6.1 Evaluation of Learning

To show the effectiveness of the proposed approach, we evaluated our dataset with 7 DFA models (corresponding to the 7 attack types). Our dataset consists of 168 malware collected from VXHEAVEN [3] and 370 benign samples. For each attack type, we used 50% of the samples for learning the DFA and remaining 50% for testing. As shown in the middle part of Table 7.1, our approach can detect all malware test samples (numbers
TABLE 7.1: Detection performances of Malguard with 50% (middle part) and 30% (right part) of samples as the training dataset

<table>
<thead>
<tr>
<th>Attack Behavior</th>
<th>Malware</th>
<th>Benign</th>
<th>Malware</th>
<th>Benign</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type I</td>
<td>14</td>
<td>14 (14)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Type II</td>
<td>22</td>
<td>21 (21)</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>Type III</td>
<td>5</td>
<td>5 (5)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Type IV</td>
<td>6</td>
<td>5 (5)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Type V</td>
<td>6</td>
<td>6 (6)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Type VI</td>
<td>30</td>
<td>29 (29)</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>Type VII</td>
<td>3</td>
<td>2 (2)</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

inside the parentheses represent total test samples). To evaluate the false positives, we tested benign samples by using DFA without frequency and with frequency. Our results show that, for attack type II and VI, our approach has combined false positives of > 6%. Using frequency along with DFA, it reduces to zero. We also evaluated the approach with a smaller training set which is 30% of the total malwares and the remaining 70% as the testing set and the results shown in the right part of the Table 7.1. These results suggest that smaller number of training examples do make the detection rate lower. We analyzed the results and the main reason for lower detection rate is that the training set does not contain an example for those undetected samples in the testing set and thus its attack model is not represented in the learned DFAs.

7.6.2 Evaluation of Detection Capability

We implemented detector on FPGA device of Virtex-5 (LX110T) using Xilinx ISE 14.5. To evaluate the area of sysbin, we used Synopsis Design Compiler tool at 65 nm technology. To verify the functionality and to estimate the performance of the overall design, we integrated the hardware components (frequency and cycles) in Multi2sim, a cycle-accurate processor simulator, and configured it to 32-bit, x86@2GHz. The system calls were traced during program execution on the simulator.
### Table 7.2: Resource utilization of DFA checker

<table>
<thead>
<tr>
<th>Resources</th>
<th>Flip flops</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>21 (&lt;1%)</td>
</tr>
<tr>
<td>LUTs</td>
<td>18 (&lt;1%)</td>
</tr>
<tr>
<td>BRAMs</td>
<td>1 (&lt;1%)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycles</td>
</tr>
<tr>
<td>Frequency (MHz)</td>
</tr>
<tr>
<td>Memory</td>
</tr>
</tbody>
</table>

#### 7.6.2.1 Results

For the proof of concept, we implemented sysbin and DFA checker. Our results show that sysbin has a negligible area overhead of 0.003% (3588.84 $\mu m^2$) for the processor. DFA checker was implemented in FPGA. Since Linux has above 300 system calls, we used 9-bits for representing the system calls. In total, there are 427 transitions for 7 attack types, so we use 9-bits to represent current state register. The transition table has a depth of 427 corresponding to the number of transitions, while each row is of 9-bits (to represent the states). As shown in the Table 7.2, the memory requirement is just 481 bytes. The memory requirement is very low as compared to [158]. This is because we only store the DFAs for attack behavior, while [158] stores the DFA for a whole benign program. Since the system call traces can be very large for real world applications, the memory requirement will be substantial for their method. Moreover, their approach requires a DFA for each program, which requires a large amount of memory.

About the performance, our DFA checker takes 4 clock cycles running at 265 MHz to detect the attack behavior, while their method [158] takes 3 clock cycles to detect anomaly. It is a marginal difference, which is not significantly important. Malguard has low resources utilization on the FPGA (<1% on Virtex-5). We selected a small buffer size (10 entries) in our design to hold the system calls while the checker performs the validation. The smaller buffer size is sufficient because the gap between system calls is large enough so that the checker can complete its operation before the buffer gets filled up.
7.6.3 Threats to Validity

There are several threats to the validity of the proposed approach in this chapter. The runtime traces used as input to extract precedence rules are collected by running the malwares in a sandbox. The first threat is that our approach may not be able to collect the malicious traces for malwares who detect and evade the sandbox environments. The other threat to validity of this approach is that only 7 types of malwares are studied in this approach, this approach may not be effective for other types of malwares. The third threat to validity is the frequency counter used to distinguish malwares from benign software is based on the observation that malwares repeat a subsequence many more times while benign software only invokes the subsequence a few times. This observation may not hold for other types of malwares.

7.7 Summary

In this chapter, we propose an online malware detection and defense approach based on automatic learning of attack behavior models. We presented the design of our implementation which detects malware at runtime. We also introduced a frequency parameter to model repetitive malicious behavior. The evaluation shows that our approach has a high detection rate with no false positive. Malguard facilitates real-time malware detection with low performance and memory overhead.

Modeling malicious behavior with system calls is first proposed by Forrest et al. [76]. Existing approaches [32, 45] use machine learning techniques to learn models for their malicious behavior. These approaches can detect malware more accurately than other dynamic approaches but still use more time and memory for detection and thus their capability for runtime defense is limited. Hardware performance counter is used to detect malware by Demme et al. [70], whose false positive rate is high. Das et al. [68] implemented a machine learning algorithm in hardware to reduce the performance overhead but detection process is still slower than our approach because the machine learning algorithm is much more resource intensive than our DFA based approach.
Rahmatian et al. [158] proposed a host-based intrusion detection using FPGA, which is closely related to our work. This approach used the system call sequences to build a finite state machine (FSM) to model the behavior of benign programs. However, it has several limitations. First, each program needs to be profiled to generate corresponding FSM. It does not allow unknown programs to run on the system. Second, the FSM can be significantly large for real-world applications. This will require a large memory. In contrast, our approach uses DFA for attack patterns only, which requires only a small amount of memory. In addition, our method does not require profiling beforehand and therefore, it allows unknown programs to run on the system.
Chapter 8

Conclusions

This thesis aims to automatically learn behavior models of programs and apply the proposed techniques to learn attack behavior models of malicious software so as to detect malware variants. We believe that different applications of behavior models pose different requirements on the learned behavior models and different challenges for learning these models. The central challenge in learning behavior models is related to the competing goals of accuracy and efficiency. In this thesis, we have proposed several fully automatic learning approaches to efficiently learn different forms of behavior models.

8.1 Thesis Summary

In Chapter 3, we propose a new approach to learning more expressive behavior models dubbed as stateful behavior models. By generating the guard conditions from the testing results for DFA learning, our approach introduces an efficient and fully automatic alphabet refinement strategy to learn stateful behavior models. The experiments show that the new approach is much more efficient than existing approaches to learn behavior models with equivalent power of expressiveness.

In Chapter 4, we propose a new approach to learning accurate and complete behavior models which are indispensable for certain applications such as model checking. By using symbolic execution to check and possibly refine the behavior models generated by
testing, our approach solves the low coverage problem of testing. Symbolic execution or other heavy-weight formal techniques (e.g., theorem proving) is indispensable for learning accurate and complete behavior models. Symbolic execution is used in limited places such that the efficiency of the learning approach is maintained.

In Chapter 5, we propose a new approach to learning behavior models from existing client programs for a program library. By leveraging testing results and the knowledge in existing client programs, our approach scales to real world libraries such as classes in the Java standard libraries.

To demonstrate the usefulness of behavior models and the scalability of the proposed learning approaches, we apply the proposed learning approaches to malware analysis. In Chapter 6, we propose to automatically learn attack behavior models of malicious JavaScript programs. The learned attack behavior models are then used to detect other malicious JavaScript variants in the same family. By using attack behavior models, the detection is essentially a semantics-based malware detection approach which is much more accurate and more resilient than signature-based malware detection approaches.

In Chapter 7, we propose to automatically learn attack behavior models of conventional malware in terms of the system calls the malware invokes. By encoding learned attack behavior models of the malware in the hardware, our approach is much more efficient to detect malwares and is more resilient to malware detection evasion techniques.

In conclusion, the approaches presented in this thesis have advanced the state of art of behavior model learning. With the proposed approaches, the learned models can be used in various applications which require the models to be of different power of expressiveness. We regret not conducting more experiments on the effectiveness of learned models when used for their conventional applications such as code recommendation [143] and code completion [41, 161].

### 8.2 Future Works

In this thesis, the quest for more efficient learning of accurate behavior models for programs has been mainly put on the active learning perspective, where the models to
be learned are in the form of or extensions of finite state machines.

One interesting direction to explore is to learn behavior models in other formal models such as context-free grammars which are equivalent to push-down automata and models used for model checking such as Event-Recording Automata [112], Probabilistic Real Time Systems [176], and Communicating Sequential Processes [174]. The learned context-free grammars can be used for security applications such as automated robustness testing and fuzzing [90, 185].

The other possible direction to explore is to learn behavior models of programs under different assumptions. One of the possible future topics is to learn behavior models of black-box programs whose source code is not available. One of potential applications of this learning technique is to reverse-engineer network protocols whose source code are often not available.

All the approaches in the first part of the thesis learn behavior models of a single class. However, a library which often contains multiple interacting classes. In such cases, learn behavior models for multiple interacting objects would be more useful. There are several approaches which can learn behavior models for multiple objects [110, 153], we aim to achieve better performance than existing approaches.

One of the applications of behavior models is to study the evolution of Android SDK or Android Apps especially Android malwares [133–135, 137–139, 194]. We can learn models of one version of the Android SDK (an Android App) and learn models of another version of the Android SDK (the Android App). We can study the evolution of Android SDK (the Android App) by comparing its models of different versions.
Bibliography


[74] Marcelo M. Eler, Andre T. Endo, and Vinicius H. S. Durelli. Quantifying the Characteristics of Java Programs That May Influence Symbolic Execution from a


[100] Alexandros Kapravelos, Yan Shoshitaishvili, Marco Cova, Christopher Kruegel, and Giovanni Vigna. Revolver: An Automated Approach to the Detection of


[117] Shang-Wei Lin, Jun Sun, Hao Xiao, Yang Liu, David Sanan, and Henri Hansen. FiB: Squeezing Loop Invariants by Interpolation between Forward/Backward


