Symbolic Model Learning: New Algorithms and Applications

George Argyros

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2019

Symbolic Model Learning: New Algorithms and Applications

by

George Argyros

Abstract

In this thesis, we study algorithms which can be used to extract, or learn, formal mathematical models from software systems and then using these models to test whether the given software systems satisfy certain security properties such as robustness against code injection attacks. Specifically, we focus in studying learning algorithms for *automata* and *transducers* and the symbolic extensions of these models, namely symbolic finite automata (SFAs). In a high level, this thesis contributes the following results:

1. In the first part of the thesis, we present a unified treatment of many common variations of the seminal L^* algorithm for learning deterministic finite automata (DFAs) as a congruence learning algorithm for the underlying Nerode congruence which forms the basis of automata theory. Under this formulation the basic data structures used by different variations are unified as different ways to implement the Nerode congruence using queries.

2. Next, building on the new formulation of L^*-style algorithms we proceed to develop new algorithms for learning transducer models. Firstly, we present the first algorithm for learning deterministic partial transducers. Furthermore, we extend my algorithm into non-deterministic models by introducing a novel, generalized congruence relation over string transformations which is able to capture a subclass of string transformations with regular lookahead. We demonstrate that this class is able to capture many practical string transformation from the domain of string sanitizers in Web applications.

3. Classical learning algorithms for automata and transducers operate over finite alphabets and have a query complexity that scales linearly with the size of the alphabet. However, in practice, this dependence on the alphabet size hinders the performance of the algorithms. To address this issue, we develop the MAT^* algorithm for learning symbolic finite state automata (s-FAs) which operate over infinite alphabets. In practice, the MAT^* learning algorithm allow us to plug custom transition learning algorithms which will efficiently infer the predicates in the transitions of the s-FA without querying the whole alphabet set.

4. Finally, we use our learning algorithm toolbox as the basis for the development of a set of black-box testing algorithms. More specifically, we present Grammar Oriented Filter Auditing (GOFA), a novel technique which allows one to utilize my learning algorithms to evaluate the robustness of a string sanitizer

or filter against a set of attack strings given as a context free grammar. Furthermore, because such grammars are many times unavailable, we developed `sfadiff` a differential testing technique based on symbolic automata learning which can be used in order to perform differential testing of two different parser implementations using s-FA learning algorithms and we demonstrate how our algorithm can be used to develop program fingerprints. We evaluate our algorithms against state-of-the-art Web Application Firewalls and discover over 15 previously unknown vulnerabilities which result in evading the firewalls and performing code injection attacks in the backend Web application. Finally, we show how our learning algorithms can uncover vulnerabilities which are missed by other black-box methods such as fuzzing and grammar-based testing.

Contents

List of Figures

List of Tables

Acknowledgments

This thesis would not have been possible without the people that supported me throughout the years I've been in Columbia. I would like to start by thanking my co-authors that contributed in the research presented in this thesis: Suman Jana, Margus Veanes, Loris D'Antoni, Aggelos Kiayias, Ioannis Stais and my advisor Angelos Keromytis.

I would like to particularly thank my undergraduate advisor Aggelos Kiayias, who has been a long-term collaborator in a number of projects presented in this thesis as well as a mentor and good friend. Ioannis Stais has also been a close collaborator for a number of years and contributed significantly in bringing this thesis closer to the practical setting.

Beyond the research presented in this paper I had the wonderful opportunity to collaborate with a number of great people from within the department. I would like to particularly thank Junfeng Yang and Roxana Geambasu for working with me and for being quite supportive during my studies.

My colleagues in Columbia played a significant role in my life both professionally and personally. Specifically, I would like to thank Theofilos Petsios, Vaggelis Atlidakis, Suphannee Sivakorn, Dimitris Mitropoulos and Marios Pomonis.

Last but not least, I would like to thank my advisors Tal Malkin and Angelos Keromytis for giving me the opportunity to come to Columbia and for supporting me throughout the years of my studies.

It goes without saying that this thesis would not exist without the continuous support of my family and friends. In both joyful and difficult times you were always there for me, thank you.

Chapter 1

Introduction

In modern years, the wide adoption of computer systems in every aspect of our lives have revolutionized modern societies. From the Internet revolution, to self-driving cars and heart pacemakers which are remotely controlled by software systems, our lives are becoming more and more dependent on the correct functionality of the software and hardware systems we develop.

In this computer-dependent world, the development of tools and algorithms which allow us to analyze properties of software and hardware systems is of paramount importance. However, due to it's generality, this problem is very difficult to tackle effectively. In it's more general mathematical form the problem is unsolvable in a formidable manner: the famous Rice's theorem [74] from the early 1950s, states that any non-trivial semantic property of a computer program is undecidable. Here, by non-trivial semantic property we basically mean any program property which cannot be derived simply by the syntactic structure of the program, i.e. it depends on the *semantics* of the program.

In order to cope with undecidability, the research community developed a large body of approximation techniques such as the seminal *abstract interpretation* [29] framework, where the program is analyzed with respect to an abstract domain and then the program is executed with respect to some abstract semantics which are easier to analyze. In more general terms, the main avenue of research in order to prove program properties is the following: Initially, we use a simpler computational

model which can be analyzed efficiently with respect to the desired property and then, we construct an approximation of the original program in the selected computational model and analyze the approximation instead of the original program. Constructing the approximation is usually performed with specialized algorithms such as static analysis algorithms.

This thesis will follow the same general avenue of research however, we will study a different way of constructing the approximations of the system to be analyzed. More specifically, we develop novel active learning algorithms which can be used to extract formal models from software systems by actively querying the target system, producing a model of the system and finally, refining the model using counterexamples, i.e. inputs where the output of the model is not consistent with the output of the target system. This learning model, which is called learning with a minimally adequate teacher (MAT), is a natural learning model where the algorithm is able to ask the target system queries with arbitrary inputs and obtain the output of the system in these queries and moreover, to *test* whether a candidate model is correct or obtain a counterexample.

In terms of formal models, we will utilize finite automata for modelling programs with binary output and transducers (automata with output) in order to model general programs. Automata and transducers are among the most fundamental computational models since they present nice algebraic properties, efficient computation of many properties and moreover, they are expressive enough in order to model or approximate many important real-life functionalities such as parsers and string transformation routines.

The field of active learning algorithms for automata and transducers was motivated by a series of negative results which proved NP-Hardness of Ocam-razor style learning algorithms for deterministic finite automata [42] . In 1989, Dana Angluin presented the seminal L^* algorithm [14] which was the first algorithm which can learn deterministic finite automata using membership and equivalence queries in polynomial time and using a polynomial number of queries. Since then, this algorithm has spawn a large number of variations and optimizations as well as a number of applica-

tions in various domains. In the first part of this thesis, we will study the L^* family of algorithms under an algebraic automata-theoretic view; the main advantage of this new formulation of the algorithm is the unification of the main data structures which by this type of algorithms such as the observation table and the classification tree as data structures which implement the Nerode congruence relation using queries. Once these data structures are abstracted away and the algorithm is explained in terms of extending a congruence relation, we can greatly simplify the analysis and presentation of the whole family of L^* algorithms.

Next, we move to our first novel algorithm. Specifically, we present a novel L^*-style algorithm for learning deterministic partial transducers. While learning total deterministic transducers can be achieved using a simple extension to the original L^* algorithm, once we introduce partiality the learning process becomes much more challenging. To address this problem, we develop an independent algorithm which can be used to infer the output of each transition in a single-valued transducer given the underlying state machine of the transducer. Afterwards, we extend the syntactic congruence which forms the equivalent of the Nerode congruence for transducers into a generalized form which can model string transformations functions with regular lookahead and provide an instantiation of our generalized congruence which gives rise to a canonical class of non-deterministic transducers. Finally, we provide an extension of our deterministic learning algorithm for this new class of transducers.

Afterwards, we demonstrate the applicability of our novel learning algorithms in the domain of string sanitizers for Web applications. We demonstrate that our novel class of non-deterministic class can capture a large number of string transformations which are used by popular string sanitization frameworks such as the Internet Explorer and Edge XSS filters. To the best of our knowledge our learning algorithm is the first that can efficiently infer models of such string transformations which can then be used for further analysis of the sanitization routines.

L^* and other similar algorithms work primarily over a finite alphabet Σ and the number of queries performed by the algorithm scales linearly with the size of the alphabet. However, when we want to use this type of algorithms in order to learn

models of parsers and string transformation routines we need to be able to operate our learning algorithms using very large alphabet such as UTF-16 which includes 2^{16} symbols. In order to address this significant scalability issue, we extend the L^* algorithm into symbolic automata (s-FA). Symbolic automata have transitions which operate over predicates instead of individual characters and can therefore represent regular languages over infinite alphabets. In the next chapter of the thesis, we present the MAT^* algorithm which can learn s-FAs over any boolean algebra which is also learnable using membership and equivalence query. We demonstrate that MAT^* allows us to scale automata learning algorithms efficiently into alphabets such as UTF-16. Another important practical implication of this algorithm is the capability to plug any learning algorithm for inferring the transitions of the s-FA independently of the learning algorithm which learns the congruence. We demonstrate this point by presenting and analyzing a statistical learning algorithm which infers the predicates in each transitions by learning from a data-set of s-FAs. Finally, beyond the practical implications of our algorithm we demonstrate how our algorithm provides an almost complete characterization of the set of efficiently learnable s-FAs.

Now that we have developed a comprehensive toolbox of learning algorithms, our next goal is to develop techniques and algorithms which will allow us to use these algorithms for the analysis of systems. We focus in the analysis of Web applications for code injection attacks. Code injection attacks currently present the primary risk factor for Web application security. In a nutshell, code injection attacks occur when the application confuses part of the user input which is intended to be data as code, therefore changing the semantics of the execution of the Web application code. Code injection attacks can result in the execution of arbitrary code in the Web server or the user's browser, leaking of sensitive information and other severe security implications. In order to defend against code injection attacks, Web applications employ a number of different lines of defenses. In this thesis we will focus primarily on two popular defense mechanisms: (1) Web Application Firewalls (WAFs) and string sanitizers.

Web Application Firewalls work as a generic defense mechanism and they are generally deployed independently of the Web Application. While many different ar-

chitectures are available, in their most common form, web application firewalls act as a parser which tries to detect whether an input to the Web application contains a code injection (or other) attack and in this case the request is dropped from further processing by the application. On the other hand, string sanitizers work by taking as input an input to the application which is "unsanitized" and through a series of string transformations such as removing potentially dangerous part of the input and encoding certain sensitive characters, transform the input into a "sanitized" input which is safe for further processing by the Web application.

Our main goal is to use our learning algorithm in order to extract models of Web Application Firewalls and String sanitizers and analyze their robustness against code injection attacks. The first setting we consider is the Grammar Oriented Filter Auditing (GOFA) problem. In the GOFA setting we are given a context free grammar G which describes the set of attack strings for a particular code injection attack. For example, consider the class of SQL Injection attacks; then, the context free grammar G can be the set of valid continuations to SQL statements that start with a particular prefix. Given such a grammar the GOFA problem asks to find a string s belonging to G such that s is bypassing the filter or sanitization routine. We will demonstrate that, by using the context free grammar G in order to simulate an equivalence query as follows: given a model inferred by our learning algorithms, we use the model to see if there exists any string $s \in G$ which bypasses our inferred model. If such an input is found, then we test the candidate attack against the actual filter or sanitizer. If the attack succeeds then we have effectively solved the problem. Otherwise, notice that the string s is a counterexample to our model and therefore can be used to further refine the model.

While the GOFA algorithm can be used to efficiently test the robustness of filters and sanitizers in a black-box manner when the grammar G is available, such a detailed description of the set of attack strings is often not available or not accurate enough in order to thoroughly test the robustness of firewalls and sanitizers against code injection attacks. For example, in order to thoroughly evaluate the robustness of Cross Site Scripting (XSS) filters and sanitizers one would need a grammar describing the

set of HTML statements which result in Javascript execution. However, the HTML standard is implemented quite differently by each different browser and these small variations actually play a significant role in the evaluation of the robustness of string sanitizers and filters. To address this important issue we develop SFADiff a technique building on top of our GOFA algorithm. In a nutshell, instead of being given the grammar G as an input to the GOFA algorithm, we utilize our learning algorithms in order to infer the set of attack strings. For example instead of using the HTML standard as the set of attack strings for the GOFA algorithm, we can utilize our learning algorithms to infer regular approximations of the HTML standard parsed by different browsers. Finally, we show how these techniques can be also used in order to generate *program fingerprints* which can be used to distinguish between different implementations using only black-box queries.

In order to evaluate the effectiveness of our GOFA and SFADIff algorithms we use our algorithms to evaluate the robustness of popular Web Application Firewalls against common code injection attacks such as SQL Injection (SQLi) and Cross Site Scripting (XSS). Our GOFA algorithm is able to find more than 10 previously unknown bypasses against Mod-Security, the most popular open source WAF, while our SFADiff algorithm found 6 different XSS attacks bypassing PHPIDS and Expose two popular WAFs. Moreover, we demonstrate how our algorithms can be used to built a fingerprint tree which can be used in order to distinguish between different WAF products using only black-box queries. Finally, we demonstrate that, when used in conjunction with our transducer learning algorithm, the GOFA algorithm can uncover sophisticated recursive XSS attacks against sanitizers. We showcase this point by demonstrating how our system can bypass a sanitization function in a popular PHP application and construct a valid XSS attack.

1.1 Bibliographical Note

The results presented in this thesis have been also peer reviewed and presented in the following conferences: The GOFA algorithm along with a preliminary version of

the MAT* algorithm were presented under the title "Back in Black: Towards Formal Black-box Analysis of Sanitizers and Filters" in the "2016 IEEE Symposium on Security and Privacy". The SFADiff extension was presented under the title "SFADiff: Automated Evasion Attacks and Fingerprinting using Black-box Differential Automata Learning" in the "Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security". The MAT* algorithm was presented under the title "The Learnability of Symbolic Automata" in "2018 International Conference on Computer Aided Verification". The work on transducer learning algorithms is currently under peer review.

Chapter 2

Related Work

2.1 Automata Learning

Hardness of Automata Learning. Active learning algorithms for Deterministic Finite Automata and other related classes were initated after a series of negative results on various passive learning models. Gold [42] shows that finding the minimum DFA consistent with a set of samples is NP-Hard while, a few years later, the problem was shown to be NP-Hard to approximate within any polynomial [72]. Moreover, a representation-independant hardness result for learning automata based on cryptographic assumptions was proven by Kearns and Valiant [52]. Even in the case where access to membership queries is provided, certain models such as non-deterministic finite automata, context free grammars, unions of DFAs and others remain hard to learn [16].

The L^* Algorithm. In order to cope with hardness results, Angluin [14] introduced the Minimally Adequate Teacher (MAT) learning model and the L^* algorithm. After its introduction, Anlguin's algorithm was improved and many variations were introduced; Rivest and Schapire [75] showed how to improve the query complexity of the algorithm and introduced the binary search method for processing counterexamples. The classification tree data structure was introduced by Kearns and Vazirani [53]. Balcazar et al. [18] describe a general approach to view the different variations of Angluin's algorithm. Isberner et al. [51] recently developed the TTT algorithm which

provide a space optimal variation of the L^* algorithm which also contributes practical improvements in the classification tree data structure by rearranging the nodes of the tree. Implementations for most variations of the L^* algorithm can be found in the LearnLib library [73].

Extensions to other models. A large number of extensions to classical automata exist with different properties and applications and extensions of the L^* algorithm have been developed for such models as well. Residual Finite State Automata (RFSA) [35] are canonical non-deterministic automata which can be exponentially more succint than DFAs. A learning algorithm for RFSAs was developed by Bollig et al. [22]. Adaptations of the L^* were also developed for alternating [15] and nominal [63] which are more expressive than the classical deterministic finite automata. Finally, another important model is register automata [25] which are canonical automata models which incorporate restricted use of registers and for which different learning algorithms were developed [49, 13].

Other learning algorithms. Beyond the setting of active learning, the `RPNI` algorithm [67] is usually employed in order to learn deterministic finite automata from a set of samples. The algorithm will learn a correct DFA when a characteristic sample of the regular language is provided (identification in the limit). Finally, a line of work on kernel methods for identifying regular languages was developed more recently [28, 55].

Learning transducers. Transducers were introduced as a mathematical object by Marcel-Paul Schutzenberger and were used extensively in the past in natural language processing [64]. However, lately transducers (and their extensions) have seen extensive applications in the field of programming languages as an underlying formalism for many domain specific languages [47, 41, 61]. A minimization algorithm for deterministic finite state transducers was developed by Mohri [65].

An adaptation of the L^* algorithm to transducers [84] and Mealy machines [77] were introduced after the development of the L^* algorithm. A general treatment of the field of grammatical inference can be found in [33]. An algorithm for learning non-deterministic Mealy machines was developed in [54], however, the algorithm required

a stronger type of queries than output queries and has an exponential running time. The OSTIA algorithm is a popular passive learning algorithm for total transducers developed by Oncina et al. [68].

2.2 Symbolic Automata and Transducers

Symbolic automata and transducer were initially introduced to cope with large alphabets that were used in regular expressions and as a theoretical tool to reason about automata with infinite alphabets. This line of work was initiated with the introduction of symbolic automata [82], although similar constructions were suggested much earlier [86]. The introduction of symbolic finite state transducers (SFTs) followed shortly afterwards with the developed of BEK [48], a domain specific language for string sanitizers while the corresponding theory can be found in a follow up work by Bjorner et al. [21]. A minimization algorithm for symbolic automata was developed by D'Antoni and Veanes [31] while a minimization for symbolic transducers was developed by Saarikivi and Veanes [76]. Extensions to richer symbolic models and domain specific languages were developed in the following years [37, 83, 81, 32]. A general survey of the field of symbolic automata and transducers can be found in [39].

Learning Symbolic Automata and Transducers. In the inference of symbolic automata and transducers there are two relevant recent works. Botincan and Babic [23] used symbolic execution in combination with the Shabaz-Groz algorithm in order to infer symbolic models of programs as symbolic lookback transducers. Although the authors claim that equivalence of symbolic lookback transducers(SLT) is decidable a paper published recently by D'Antoni and Veanes [38] shows that equivalence of SLTs is in fact undecidable. The problem of learning regular languages over large alphabets has attracted attention even before the introduction of symbolic automata through various techniques such as alphabet refinement [50] and parametric languages [20]. The first algorithm to address learning of symbolic automata was developed by Maller and Mens [60], however the algorithm assumed the existence of an equivalence oracle which provides counterexamples of minimal length.

2.3 Applications of Automata Learning

In the context of testing, automata learning algorithms were used in order to infer specifications for model checking, a concept called black-box checking [70, 43]. The Vasileski-Chow algorithm [27], an algorithm for checking compliance of two automata, given an upper bound on the size of the black-box automaton. This algorithm however, has a worst case exponential complexity a fact which makes it impractical for real applications. Another important area of application for automata learning application is learning models of network protocols. Recent examples include learning and model checking models of the SSH protocol [40], learning TLS/SSL state machines [34], botnet command and control centers [26] and TLS hostname verification routines [79]. Finally, we point out that a general review of applications of automata learning algorithms is presented in a recent review article [80].

2.4 Web Application Analysis

There is a large body of work regarding whitebox program analysis techniques that aim at validating the security of sanitizer code. The SANER [19] project uses static and dynamic analysis to create finite state transducers which are overapproximations of the sanitizer functions of programs. Minamide [62] constructs a string analyzer for PHP which is used to detect vulnerabilities such as cross site scripting. He also describes a classification of various PHP functions according to the automaton model needed to describe them. The Reggae system [58] attempts to generate high coverage test cases with symbolic execution for systems that use complex regular expressions. Wasserman and Su [85] utilize Context free grammars to construct overapproximations of the output of a web application. Their approach could be used in order to implement a grammar which can then be used as an equivalence oracle when applying the cross checking algorithm for verifying equality between two different implementations.

Chapter 3

Background

In this section, we will provide a general introduction to the formal models we will use in this thesis. For a more complete description of basic concepts we recommend basic theory of computation textbooks [57].

3.1 Strings and Languages

Given a set Σ, Σ^* denotes the set of all finite sequences of elements over Σ. We do not distinguish here between Σ and unit-length sequences in Σ^*, assuming thus that $\Sigma \cap \Sigma^* = \Sigma$. The empty sequence is denoted by ϵ. We frequently refer to Σ as an *alphabet*, its elements are called *characters* and sequences of characters are called *strings*. We write s, t, u, v, w for strings and $a, b, c, \alpha, \beta, \gamma$ for characters. We let x, y, z range over characters as well as strings. For any integers m and n let $[m..n]$ denote the range $\{i \mid m \leq i \leq n\}$. Given a *bottom* element $\perp \notin \Sigma^*$ we write $\Sigma^*_\perp$ for $\Sigma^* \cup \{\perp\}$. We use Γ similarly to Σ. We use Σ for input alphabets and Γ for output alphabets of string functions. We denote the Boolean domain as $\mathbb{B} = \{\mathbf{T}, \mathbf{F}\}$.

The length of a string s is denoted by $|s|$. For $s \in \Sigma^*$ and $i \in [1..|s|]$, let $s[i..]$ be the *suffix* of s from position i, let $s[..i]$ be the *prefix* of s upto position i, and let $s[i]$ be the character at position i.

3.1.1 String Operations.

For $u, v, w \in \Sigma^*$ and $a \in \Sigma$ we define the following operations:

- $u \preccurlyeq v$: u is a prefix of v; $u \prec v$ means $u \preccurlyeq v$ and $u \neq v$.
- $v \succcurlyeq u$: u is a suffix of v; $v \succ u$ means $v \succcurlyeq u$ and $u \neq v$.
- $u \sqcap v$: the maximal common prefix of u and v; $u \sqcap \bot = u$ and $\bot \sqcap v = v$.
- $u \sqcup v$: the maximal common suffix of u and v; $u \sqcup \bot = u$ and $\bot \sqcup v = v$.
- $u{\cdot}v$ (or uv for short): the product (concatenation) of u with v.
- for $k \geq 1$, $i \in [1..k]$, $v_i \in \Sigma^*$, let $\bigotimes_{i=1}^{k} v_i$ denote the generalized product $v_1{\cdot}v_2 \cdots v_k$.
- if $w = u{\cdot}v$ then $w \mathbin{/} v \stackrel{\text{def}}{=} u$ (*right division of w by v*).
- if $w = u{\cdot}v$ then $u \mathbin{\backslash} w \stackrel{\text{def}}{=} v$ (*left division of w by u*).
- if $L \subseteq \Sigma^*$ then $\boldsymbol{prefixes}(L) \stackrel{\text{def}}{=} \{v \mid \exists u \in L.v \preccurlyeq u\}$; L is *prefix closed* if $\boldsymbol{prefixes}(L) = L$.

Let $\bot \preccurlyeq \bot$ but, forall $u \in \Sigma^*$, $\bot \not\preccurlyeq u$ and $u \not\preccurlyeq \bot$. For all operations, other than $\sqcap$ and $\sqcup$, the result is $\bot$ whenever some argument is $\bot$. Given a function $f : \Sigma^* \to \Gamma_\bot^*$, the *domain* of f is $\boldsymbol{dom}(f) \stackrel{\text{def}}{=} \{v \in \Sigma^* \mid f(v) \neq \bot\}$, f is *monotone* when forall $u, v \in \boldsymbol{dom}(f)$ if $u \preccurlyeq v$ then $f(u) \preccurlyeq f(v)$. The following properties are used.

Proposition 1. *Let X be a nonempty subset of strings. There exist fixed witnesses $w_1, w_2 \in X$ such that $\bigsqcap X = w_1 \sqcap w_2$ and for all $x \in X$ either $\bigsqcap X = x \sqcap w_1$ or $\bigsqcap X = x \sqcap w_2$.*

3.1.2 Regular Expressions

Given a fixed finite alphabet Σ we make use of standard notation of regular expressions (over Σ) and if r is a regular expression then $[\![r]\!]$ is the corresponding language that r denotes. For example, $[\![\texttt{.*}]\!] = \Sigma^*$ and $[\![\texttt{[\^{}a].*|()}]\!]$ is the set of all strings that do not start with character `a`, provided for example that Σ is ASCII.

3.1.3 Derivatives

Given character $a \in \Sigma$ and language $L \subseteq \Sigma^*$, the *derivative of L with respect to a* is the language $a'L \stackrel{\text{def}}{=} \{v \in \Sigma^* \mid a{\cdot}v \in L\}$. L is *nullable* if $\epsilon \in L$. When we work with *regular* languages, we make use of the property that any derivative of a regular language is also regular. For example $\texttt{b}'[\![\texttt{[\^{}a].*|()}]\!] = [\![\texttt{.*}]\!]$ and $\texttt{a}'[\![\texttt{[\^{}a].*|()}]\!] = \emptyset$.

3.1.4 Automata

We will now proceed to define finite state automata which form the basis of the models we study in this thesis.

Definition 1. A finite automaton (FA) is a tuple $\mathcal{A} = (Q, q_0, F, \delta)$ where Q is the set of states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states and $\delta \subset Q \times \Sigma \times Q$ is the transition relation.

In this thesis, we will work primarily with deterministic finite automata(DFA), where determinism is defined as follows:

Definition 2. A finite automaton $\mathcal{A} = (Q, q_0, F, \delta)$ is deterministic if and only if $(q, \alpha, p) \in \delta \wedge (q, \alpha, p') \in \delta$, then we have that $p = p'$.

In other words, a finite automaton is deterministic if and only if every transition from a state on a specific input symbol have a unique target state.

An important property of deterministic finite automata (and FA automata in general) is that they form a Boolean algebra, i.e. they are closed under the boolean operations of intersection,union and negation. Moreover, equivalence of DFAs can be efficiently checked in time $O(n \log n)$ and emptiness can be also checked efficiently. These properties make DFAs an attractive model for analyzing programs.

3.1.5 Transducers

A *transducer* is a tuple $A = (\Sigma, \Gamma, Q, q_0, F, \Delta, \lambda)$ where Σ is the *input alphabet*, Γ is the *output alphabet*, Q is a set of *states*, $q_0 \in Q$ is the *initial state*, $F \subseteq Q$ is the set

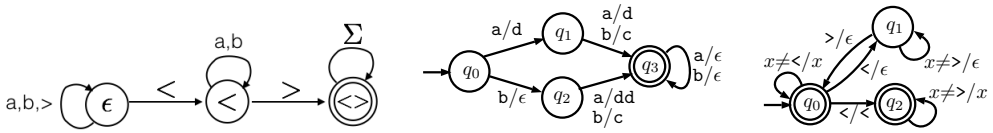

Figure 3-1: Examples of automata and transducers. (left:) A deterministic finite automaton accepting the language <[^>]*>. (middle:) A partial deterministic transducer. (right:) A total functional non-deterministic transducer that removes HTML tags.

of *final states*, $\Delta \subseteq Q \times \Sigma \times \Gamma^* \times Q$ is the *transition relation*, $\lambda \in \Gamma^*$ is the *output prefix*. A is *finite* if all of its components are finite.

We let $p \xrightarrow{a/u} q$ denote $(p, a, u, q) \in \Delta$. Transducer A is a simple computational model which consumes input symbols and produces output symbols. Let the transitive closure of the transition relation of A, denoted Δ^*, be the least subset T of $Q \times \Sigma^* \times \Gamma^* \times Q$ such that

- if $q \in Q$ then $(q, \epsilon, \epsilon, q) \in T$;
- if $p_{i-1} \xrightarrow{a_i/u_i} p_i$ for $i \in [1..k]$ then $(p_0, \bigotimes_{i=1}^{k} a_i, \bigotimes_{i=1}^{k} u_i, p_k) \in T$.

A state q is *reachable* if there exist u and v such that $(q_0, u, v, q) \in \Delta^*$; we say that u *accesses* q. A state q is *live* if there exist u and v and $p \in F$ such $(q, u, v, p) \in \Delta^*$; we say that u *is enabled from* q. Transducer A is said to be *trim* if all of its states are reachable and live. For each state $q \in Q$ let the *transduction of A (from q)* denoted $\mathcal{T}_A$ ($\mathcal{T}_A^q$) be the following subset of $\Sigma^* \times \Gamma^*$,

$$\mathcal{T}_A^q \stackrel{\text{def}}{=} \{(u, v) \mid \exists p \in F.(q, u, v, p) \in \Delta^*\}, \quad \mathcal{T}_A \stackrel{\text{def}}{=} \{(u, \lambda \cdot v) \mid (u, v) \in \mathcal{T}_A^{q_0}\}.$$

The *domain of A*, $\boldsymbol{dom}(A)$, is the set of all u such that there exists some v such that $(u, v) \in \mathcal{T}_A$. A is *total* when $\boldsymbol{dom}(A) = \Sigma^*$ and *partial* otherwise. A is *functional* if, for all $u \in \Sigma^*$ there is at most one $v \in \Gamma^*$ such that $(u, v) \in \mathcal{T}_A$. A is *deterministic* if, for all $q \in Q_A$ and $a \in \Sigma$ there exist at most one u and p such that $(q, a, u, p) \in \Delta$. If A is deterministic then A is also functional. Figure 3-1(right) shows a trim total nondeterministic functional finite state transducer. Figure 3-1(left) is deterministic and partial. In this paper, we are going to work exclusively

with functional transducers. For functional transducers A we adopt the view of the transduction of A as a partial function over Σ^* or, because it is more convenient to work with total functions, as a function from Σ^* to $\Gamma^*_\perp$ such that for all $u \in \Sigma^*$, if $u \in \textbf{\textit{dom}}(A)$ and $(u, v) \in \mathcal{T}_A$ then $\mathcal{T}_A(u) = v$, and if $u \notin \textbf{\textit{dom}}(A)$ then $\mathcal{T}_A(u) = \perp$.

Earliest Normal Form

For $q \in Q$ define $\widehat{q}$ as the *greatest common prefix* of all the outputs originating from q, i.e., $\widehat{q} \stackrel{\text{def}}{=} \bigsqcap\{v \mid \exists p \in F : \exists u \in \Sigma^* : (q, u, v, p) \in \Delta^*\}$. We let $\bigsqcap \emptyset \stackrel{\text{def}}{=} \perp$, so that $\widehat{q} = \perp$ when q is not live. Observe that $\widehat{q} = \epsilon$ for all $q \in F$ because $(q, \epsilon, \epsilon, q) \in \Delta^*$ for all q. The *earliest normal form* of A or $ENF(A)$ is defined as follows.

$$ENF(A) \stackrel{\text{def}}{=} (\Sigma, \Gamma, Q, q_0, F, \{(p, a, \widehat{p} \setminus (u{\cdot}\widehat{q}), q) \mid (p, a, u, q) \in \Delta\}, \lambda{\cdot}\widehat{q_0}).$$

Observe that $\widehat{p} \preccurlyeq u{\cdot}\widehat{q}$ for all $(p, a, u, q) \in \Delta$ so the output $\widehat{p} \setminus (u{\cdot}\widehat{q})$ is indeed well-defined.

3.1.6 Context Free Grammars

A straightforward generatlization to automata and transducers is the addition of a stack to the corresponding models. The corresponding models are called pushdown automata and transducers and they are one of the most classic models which combine rich expressive properties with low complexity. An equivalent definition of pushdown automata is given through context free grammars which we will extensively in order to provide formal specifications of attacks.

Definition 3. A Context Free Grammar (CFG) G is a 4-tuple $G = (V, \Sigma, R, S)$ where V is the set of non-terminals, Σ is the set of terminals, $R \subseteq V \times (V \cup \Sigma)*$ is the set of productions and $S \in V$ is the initial symbol.

While certain properties of CFGs are undecidable such as computing whether two grammars are equivalent, other important properties can be computed in polynomial time. Important for our applications is the computation of the intersection between a

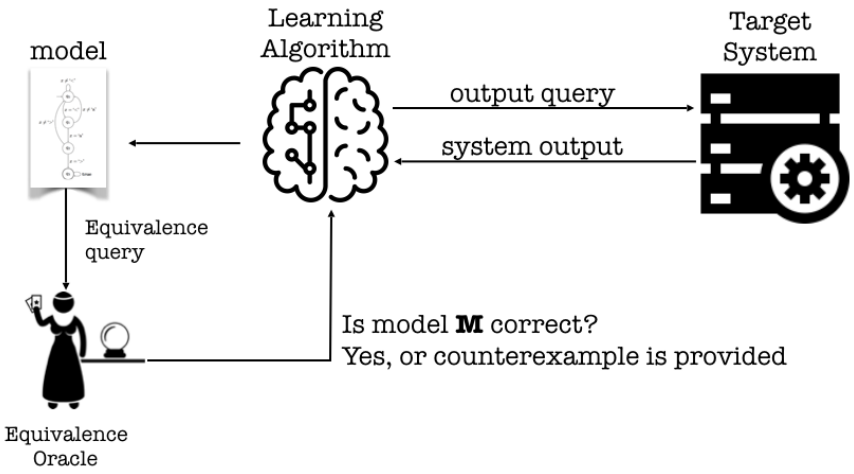

Figure 3-2: The Minimally Adequate Teacher (MAT) learning model.

CFG and a DFA which can be performed in polynomial time and moreover, checking a CFG for emptiness which can also be performed in polynomial time.

3.2 Learning Model

The algorithms described in this paper are active learning algorithms operating in a learning model called learning with a Minimally Adequate Teacher (MAT) and also learning from membership and equivalence queries [14, 53]. Under this model, a learning algorithm which is learning a target function $\mathbf{f} : \Sigma^* \to \Gamma^* \cup \{\perp\}$ is given the ability to perform two types of queries:

- **Output queries:** Also called membership queries in the case f is a boolean function. For any input $s \in \Sigma^*$, an output query will return the value of the function $\mathbf{f}(s)$.

- **Equivalence queries:** Once the learning algorithm generates a candidate model $\mathbf{h}$ for the function $\mathbf{f}$, an equivalence query is performed to verify correctness of the model. If the model is correct, i.e. for all $s \in \Sigma^*$ we have that $\mathbf{f}(s) = \mathbf{h}(s)$ then, the query will return $\mathbf{T}$. Otherwise, a counterexample $c \in \Sigma^*$ is provided such that $\mathbf{f}(c) \neq \mathbf{h}(c)$.

Relation to PAC learning. The most traditional learning model used in learning theory is the Probably Approximately Correct (PAC) model. In this model, a learning algorithm have no query access to the target function and only receives samples from the function according to some target distribution. The goal of the learning algorithm is to output a hypothesis (i.e. model) which, with high probability, has a small error on future samples from the same distribution. Under this model, even deterministic finite automata are hard to learn under cryptographic assumptions [52]. However, with a simple reduction [53] one can show that the existence of a MAT algorithm for a class of functions implies that the function is PAC-learnable when output queries are provided to the learning algorithm.

Chapter 4

Congruences and Distinguishability

Automata and transducers are computational models based on simple state machine graphs. However, these graphs can be perceived as emerging from algebraic equivalence properties of the underlying function which is computed by the automaton or the transducer. When we consider the state machines which are emerging from these underlying equivalence properties we obtain normal forms for the automata and transducers computing a specific function. More importantly, all the learning algorithms in the L^* family of algorithms, can be viewed as algorithms which iteratively learn the underlying equivalence relation and indeed, under this view, many concepts and data structures in the learning algorithms arise naturally under this formulation of the problem.

We will start this chapter by describing the Nerode congruence which forms the basis for regular languages and then proceed to describe the *syntactic congruence* from which canonical forms of transducers arise. Finally, we will discuss how all these equivalence relations can be defined through their negations, a fact which form the basis of learning.

4.1 Equivalence Relations

Before describing the equivalence relations which form the basis of automata and transducers we will first give a refreshment of *equivalence relations*.

Definition 4 (Equivalence relation)**.** A binary relation $\sim$ over a set X is said to be an equivalence relation if and only if the following hold for any $a, b, c \in X$:

1. **Reflexivity:** $a \sim a$.
2. **Symmetry:** $a \sim b \implies b \sim a$.
3. **Transitivity:** If $a \sim b$ and $b \sim c$, then $a \sim c$.

Despite the fact that the relations we will describe next are dumped congruences, they are primarily equivalence relations over strings.

4.2 Nerode Congruence

Despite the fact that automata were introduced in 1943, the algebraic underlying of automata theory came more than one decade later by Myhil and Nerode who, independently, showed that DFAs, and therefore regular languages, can be viewed as emerging from an equivalence relation over strings with respect to the target language. We will now formally define the Nerode Congruence:

Definition 5 (Nerode Congruence)**.** Let $\mathbf{f} : \Sigma^* \to \mathbb{B}$ be a boolean function. Given two strings $u, v \in \Sigma^*$ the Nerode congruence is defined as follows:

$$u \sim v \overset{\text{def}}{\iff} \forall w \in \Sigma^* : \mathbf{f}(uw) = \mathbf{f}(vw) \tag{4.1}$$

It is easy to verify that the Nerode congruence is an equivalence relation. A $\sim$-equivalence class is denoted by $\langle u \rangle_{/\sim}$ or $\langle u \rangle$ when $\sim$ is clear from the context, and given $S \subseteq \Sigma^*$, $\langle S \rangle \overset{\text{def}}{=} \{\langle x \rangle \mid x \in S\}$. Given the congruence, one can construct the corresponding DFA as follows:

$$\texttt{DFA}(\mathbf{f}) \overset{\text{def}}{=} (\langle \Sigma^* \rangle, \langle \epsilon \rangle, \langle \{u \mid \mathbf{f}(u) = \mathbf{T}\} \rangle, \{\langle u \rangle \xrightarrow{a} \langle u \cdot a \rangle \mid u \in \Sigma^*, a \in \Sigma\})$$

It follows that $\texttt{DFA}(\mathbf{f})$ is a minimal DFA for $\mathbf{f}$. Notice that the congruence relation can be defined for any function $\mathbf{f}$ regardless of whether the function is regular or not.

The seminal theorem proved by Myhill and Nerode shows that when the number of equivalence classes in this congruence relation is finite then the corresponding function is regular.

Theorem 1 (Myhill-Nerode)**.** *Let* $\mathbf{f} : \Sigma^* \to \mathbb{B}$ *be an arbitrary function. Then,* $\mathbf{f}$ *is regular if and only if the number of equivalence classes in the corresponding Nerode congruence is finite.*

4.3 Syntactic Congruence

Like automata, transducers can emerge from a similar underlying congruence relation like the Nerode congruence. While the main idea is again to force state equivalence with respect to the behavior of the target function on the set of different suffixes, here we have to take into account the fact that an output is produced during each step of the computation. The syntactic congruence is an equivalence relation which defines state equivalence with respect to general outputs.

Fix $\mathbf{f} : \Sigma^* \to \Gamma_\perp^*$. Define $\widehat{\mathbf{f}}(u)$ as the output prefix of $\mathbf{f}$ that depends only on input prefix u and define $\mathbf{f}_u$ as the continuation function of $\mathbf{f}$ after input u that cuts the prefix $\widehat{\mathbf{f}}(u)$. Then two strings $u, v \in \Sigma^*$ are *syntactically congruent* iff $\mathbf{f}_u = \mathbf{f}_v$. Formally:

$$\widehat{\mathbf{f}}(u) \stackrel{\text{def}}{=} \bigsqcap_{w \in \Sigma^*} \mathbf{f}(u{\cdot}w), \quad \mathbf{f}_u(w) \stackrel{\text{def}}{=} \widehat{\mathbf{f}}(u) \setminus \mathbf{f}(u{\cdot}w), \quad u \sim v \stackrel{\text{def}}{\iff} \mathbf{f}_u = \mathbf{f}_v \tag{4.2}$$

One can now construct a transducer from $\mathbf{f}$, $\mathcal{T}_\mathbf{f}$,

$$\mathcal{T}_\mathbf{f} \stackrel{\text{def}}{=} (\Sigma, \Gamma, \langle \Sigma^* \rangle, \langle \epsilon \rangle, \langle \boldsymbol{dom}(\mathbf{f}) \rangle, \{\langle u \rangle \xrightarrow{a/\sigma_\mathbf{f}(u,a)} \langle u{\cdot}a \rangle \mid u \in \Sigma^*, a \in \Sigma\}, \widehat{\mathbf{f}}(\epsilon))$$

where $\sigma_\mathbf{f}(u, a) \stackrel{\text{def}}{=} \widehat{\mathbf{f}}(u) \setminus \widehat{\mathbf{f}}(u{\cdot}a)$ is the output produced from state $\langle u \rangle$ for a, provided that $\langle u{\cdot}a \rangle$ is live, else $\sigma_\mathbf{f}(u, a) \stackrel{\text{def}}{=} \epsilon$. One can show that $\mathcal{T}_\mathbf{f}$ is in ENF and also minimal. The following basic property holds for the output function.

Proposition 2. *For any* $u \in \Sigma^*$ *we have that* $\bigsqcap_{\alpha \in \Sigma} \sigma(u, \alpha) = \epsilon$.

Intuition. While the Nerode congruence is largely self explained, in the sense that state equivalence basically imply that two strings have the same behavior with respect to the function $\mathbf{f}$ irrespectively from the suffix. However, in the syntactic congruence the definition is more convoluted. The main idea behind the syntactic congruence is to define the concept of "the output produced so far by the function $\mathbf{f}$" without invoking the concept of a transducer or states. In a deterministic transducer it is evident that given a common prefix the output produced up to that point will be the same irrespective of the suffixes used. This concept is captured formally in the syntactic congruence using the $\widehat{\mathbf{f}}(u)$ term, which provides the common prefix over an infinite number of suffixes and this way provides a definition of the "output produced so far". Once we have this definition, state equivalence is defined as equality of the output produced by the different suffixes.

4.4 Distinguishing predicates

Both the Nerode and the Syntactic congruence require the computation of the function on an infinite number of inputs (suffixes). Therefore, unless an explicit form of the congruence is given such as a DFA or a transducer, using simply queries to the target function, as in the case of learning, is impossible to decide whether two strings are equivalent. For this reason, Instead of reasoning directly about it we reason about its negation through witnesses of distinguishability. As we will show, both congruences can be computed efficiently when a set of distinguishing predicates is avaiable. For the following we will consider a congruence to be either the syntactic or the Nerode congruence.

A predicate $\phi(x)$ over strings *distinguishes* s *and* t (or *distinguishes* $\langle s \rangle$ *and* $\langle t \rangle$)

- $\phi(s) \not\Leftrightarrow \phi(t)$
- for all $u, v \in \Sigma^*$ if $u \sim v$ then $\phi(u) \Leftrightarrow \phi(v)$

In other words a distinguishing predicate separates at least two congruence classes.

For every finite equivalence relation there exists a finite set of predicates which distinguishes between all the different equivalence classes.

We use a set $\mathcal{R}$ of strings that represent distinct congruence classes. One can show that $\mathcal{R}$ can be chosen to be *prefix closed*, intuitively elements of $\mathcal{R}$ are access strings to distinguishable states. A set Φ of predicates is *saturated for* $\mathcal{R}$ if for all $s, t \in \mathcal{R}$ there exists $\phi \in \Phi$ such that ϕ distinguishes s and t.

Given a saturated set Φ for a congruence relation $\sim$ we can then compute the congruence relation between $u, v \in \Sigma^*$ as follows:

$$u \sim v \Leftrightarrow \bigwedge_{\phi \in \Phi} \phi(u) \Leftrightarrow \phi(v) \tag{4.3}$$

For the remainder of the paper we will commonly define a congruence relation as $(\mathcal{R}, \Phi)$ with respect to a prefix closed set of representatives $\mathcal{R}$ of $\sim$ and a saturated set Φ of predicates.

4.5 Black-box distinguishability

Now that we defined distinguishing predicates we will describe the type of predicates which can be used in order to distinguish equivalence classes in the case of the congruences we have defined so far.

4.5.1 Nerode Congruence

In the case of the Nerode congruence distinguishing between equivalence classes is straightforward. Let $\mathcal{P}_w^{\mathtt{dom}} \stackrel{\text{def}}{=} (\mathbf{f}(w) = \mathbf{T})$. Then, it follows from the definition of equivalence that

$$u \not\sim v \implies \exists w, \mathbf{f}(uw) \neq \mathbf{f}(vw)$$

Therefore, it follows that for every two equivalence classes $\langle u \rangle, \langle v \rangle$ there exists a $w \in \Sigma^*$ such that the predicate $\mathcal{P}_w^{\mathtt{dom}}$ distinguishes between $\langle u \rangle$ and $\langle v \rangle$.

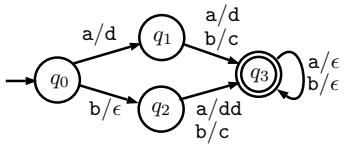

Figure 4-1: Partial transducer.

4.5.2 Syntactic Congruence

Black-box distinguishability in transducers has been studied implicitly in the context of Mealy machine and total transducer learning algorithms [77, 17]. In Mealy machines, exactly one output symbol is produced for each input symbol and therefore, checking the suffix of certain length suffices in order to extract the corresponding output produced. On the other hand, in the case of total transducers we have that $\widehat{\mathbf{f}}(u) = \mathbf{f}(u)$ and therefore, the value $\mathbf{f}_u(w)$ can be computed as $\mathbf{f}_u(w) = \mathbf{f}(u) \setminus \mathbf{f}(uw)$.

It is therefore natural to ask whether similar checks are enough to determine the output $\mathbf{f}_u(w)$ for some suffix w in the general setting of partial transducers as well. However, as the following example demonstrates, such simple suffix checks are inherently unable to distinguish between different states.

Example 1. Let $\mathbf{f}$ be the underlying function of the transducer $\mathcal{T}_\mathbf{f}$ shown in figure 4-1. Then $\widehat{\mathbf{f}}(\mathtt{a}) = \mathbf{f}(\mathtt{aa}) \sqcap \mathbf{f}(\mathtt{ab}) = \mathtt{d}$ and $\widehat{\mathbf{f}}(\mathtt{b}) = \mathbf{f}(\mathtt{bb}) \sqcap \mathbf{f}(\mathtt{ba}) = \epsilon$. Here $q_0 = \langle\epsilon\rangle$, $q_1 = \langle\mathtt{a}\rangle$, $q_2 = \langle\mathtt{b}\rangle$ and $q_3 = \langle\mathtt{aa}\rangle$. We have that

$$\mathbf{f}_\mathtt{a}(\mathtt{a}) = \widehat{\mathbf{f}}(\mathtt{a}) \setminus \mathbf{f}(\mathtt{aa}) = \mathtt{d} \setminus \mathtt{dd} = \mathtt{d}, \quad \mathbf{f}_\mathtt{a}(\mathtt{b}) = \mathtt{c},$$
$$\mathbf{f}_\mathtt{b}(\mathtt{a}) = \widehat{\mathbf{f}}(\mathtt{b}) \setminus \mathbf{f}(\mathtt{ba}) = \epsilon \setminus \mathtt{dd} = \mathtt{dd}, \quad \mathbf{f}_\mathtt{b}(\mathtt{b}) = \mathtt{c}$$

States $\langle\mathtt{a}\rangle$ and $\langle\mathtt{b}\rangle$ are distinguishable with the input suffix $\mathtt{a}$, but there is no *position conflict* which is detectable between them because $\mathbf{f}_\mathtt{b}(w) \succcurlyeq \mathbf{f}_\mathtt{a}(w)$ for all w. However, notice that since $\mathbf{f}(aa) = \mathbf{f}(ba) = \mathtt{dd}$, no simple suffix check is capable of using this input to distinguish between the states accessed by a and b respectively. ⊠

Example 1 is important because it demonstrates that simple methods of proving distinguishability in a black-box manner that were used by previous learning algorithms *cannot be ported into the setting of partial transducers.*

A class of distinguishing predicates

We now describe a class of predicates which are provably distinguishable for any two distinct equivalence classes. We start by defining the *suffix extraction* function $\mathcal{J}$ which can be used to compute $\mathbf{f}_u(w)$ given only black-box access to the function $\mathbf{f}$.

$$\mathcal{J}(u, w, w_1) = (\mathbf{f}(u{\cdot}w) \sqcap \mathbf{f}(u{\cdot}w_1)) \setminus \mathbf{f}(u{\cdot}w) \tag{4.4}$$

The following lemma establishes the connection between $\mathcal{J}$ and $\mathbf{f}_u$. It is important also to note that $\mathcal{J}(u, w, w_1)$ is *effectively computable* for any given u, w, w_1 because $\mathbf{f}(x)$ is assumed to be effectively computable for any x and the involved string operations are effectively computable.

Lemma 1. *For any $u, w \in \Sigma^*$ there exists $w_1 \in \Sigma^*$ such that $\mathcal{J}(u, w, w_1) = \mathbf{f}_u(w)$.*

Moreover, the following property holds.

Proposition 3. *For any u, w, w_1, we have that $\mathbf{f}_u(w) \succcurlyeq \mathcal{J}(u, w, w_1)$.*

We will now proceed to define two types of distinguishing predicates which can be used in order to distinguish between different equivalence classes based on either differences in the domain or in the output.

Domain Distinguishability The first type of distinguishing predicates we will use are the *domain distinguishing predicates* $\mathcal{P}_w^{\mathtt{dom}}$ indexed by a string $w \in \Sigma^*$ which were also used in order to distinguish equivalence classes in the Nerode congruence. In the context of transducers these predicates are used when we have two equivalence classes $\langle u \rangle, \langle v \rangle$ which can be distinguished using a suffix w such that $\mathbf{f}(uw) \in \boldsymbol{dom}(\mathbf{f})$ but $\mathbf{f}(vw) \notin \boldsymbol{dom}(\mathbf{f})$ or vice versa.

Output Distinguishability As we demonstrated in example 1 distiguishing based on the output after two states is a non-trivial task. In order to complete this task we utilize the suffix extraction function defined above. In order to distinguish between two states using the suffix extraction function we choose appropriate w_1, w_2 such

that, for two different access strings u, v we have that $\mathcal{J}(u, w_1, w_2) \neq \mathcal{J}(v, w_1, w_2)$. Since, by lemma 1, we can always find such w_1, w_2, the suffix extraction function avoids the problems of simply checking the suffix as in example 1. To convert this test into a distinguishing predicate, we fix the value that $\mathcal{J}$ should have on a fixed set of strings w_1, w_2. More specifically, we define the output distinguishing predicate $\mathcal{P}^{\mathtt{out}}_{w_1,w_2,t}(u) \stackrel{\text{def}}{=} (\mathcal{J}(u, w_1, w_2) = t)$. It is easy to verify that $\mathcal{P}^{\mathtt{out}}_{w_1,w_2,t}$ for appropriately chosen w_1, w_2, t satisfies the definition of a distinguish predicate.

It is important to note that both kind of distinguishing predicates can be effectively evaluated. The following proposition states that output and domain distinguishing predicates can be used to distinguish between any $s, t \in \mathcal{R}$ for which there exists w such that $\mathbf{f}_s(w) \neq \mathbf{f}_t(w)$.

Proposition 4. *For all $s, t \in \mathcal{R}$ such that $s \nsim t$, there exists either a domain distinguishing predicate or an output distinguishing predicate that distinguishes s from t.*

Proof. In the case that there exist a sequence w such that $sw \in \boldsymbol{dom}(\mathbf{f})$ but $tw \notin \boldsymbol{dom}(\mathbf{f})$ or vice versa then, it is clear that the domain distinguishing predicate $\mathcal{P}^{\mathtt{dom}}_{w}$ distinguishes s from t. Let us consider the case where s, t have an output conflict, i.e. there exists w such that $\mathbf{f}_s(w) \neq \mathbf{f}_t(w)$. Without loss of generality, we assume that $|\mathbf{f}_s(w)| \geq |\mathbf{f}_t(w)|$ and choose w_1 such that $\mathbf{f}_s(w_1) \sqcap \mathbf{f}_s(w) = \epsilon$. We claim that the output distinguishing predicate $\mathcal{P}^{\mathtt{out}}_{w,w_1,\mathbf{f}_s(w)}$ distinguishes s from t. Indeed, consider the following cases:

- **Position conflict.** Assume that there exists some position k such that $\mathbf{f}_s(w)[k] \neq \mathbf{f}_t(w)[k]$. Then, clearly $\mathcal{J}(s, w, w_1) \neq \mathcal{J}(t, w, w_1)$.
- **Length conflict.** Assume that $\mathbf{f}_s(w) \succ \mathbf{f}_t(w)$ or, in other words, that $\mathbf{f}_s(w) = v\mathbf{f}_t(w)$ for some $v \neq \epsilon$. Then, by proposition 3 we have that $|\mathcal{J}(t, w, w_1)| \leq |\mathbf{f}_t(w)| < |\mathbf{f}_s(w)|$ and therefore, it follows that $\mathcal{J}(t, w, w_1) \neq \mathbf{f}_s(w)$.

Keep in mind that, since we assumed that $|\mathbf{f}_s(w)| \geq |\mathbf{f}_t(w)|$, no other case of length conflict exists. □

Example 2. Let us consider a set of distinguishing predicates which distinguish between all states in the transducer of figure 4-1. Notice that states q_3 and $\{q_0, q_1, q_2\}$ are distinguished by the domain predicate $\mathcal{P}_{\epsilon}^{\mathtt{dom}}$. Moreover, states q_0 and $\{q_1, q_2\}$ are distinguished by the domain predicate $\mathcal{P}_{\mathtt{a}}^{\mathtt{dom}}$ and finally states q_1 and q_2, which are indistinguishable using suffix checks, are distinguished by the predicate $\mathcal{P}_{\mathtt{a,b,d}}^{\mathtt{out}}$. ⊠

4.6 Building a DFA model

At this point, we have described how we can go from computing the congruence using an infinite number of suffixes into computing the congruence through distinguishing predicates which can be done using a bounded number of queries. However, we still need query access in order to be able to compute the negated representation of the congruence. We will now describe an algorithm which can be used in order to build a DFA model of the congruence (either the Nerode or Syntactic) which can be used withouth querying the target function. While we described such a construction in the mathematical definition of both congruences, here we will give an explicit algorithmic form which can be used by the learning algorithms.

Algorithm 1 `getDFA` Algorithm

Require: $(\mathcal{R}, \Phi)$ is a congruence relation, Σ is the alphabet
  function $L^*(\mathbf{f}, (\mathcal{R}, \Phi), \Sigma)$
    $Q \leftarrow \mathcal{R}$
    $q_0 \leftarrow \epsilon$
    $F \leftarrow \{r | r \in \mathcal{R} \wedge \mathbf{f}(r) = \mathbf{T}\}$
    $\Delta \leftarrow \emptyset$
    for $r \in \mathcal{R}$ **do**
      for $\alpha \in \Sigma$ **do**
        Let r' be such that $r' \sim r\alpha$
        $\Delta \leftarrow \Delta \cup (r, \alpha, r')$
    return $(Q, q_0, F, \Sigma, \Delta)$

Given access a congruence $(\mathcal{R}, \Phi)$ we can convert the congruence to a DFA representation as follows:

- The set of states is the set of equivalence class representatives $Q = \mathcal{R}$.

- The initial state is $q_0 = \epsilon$.
- The set of final states is $F = \{u | u \in Q \wedge \mathbf{f}(u) = \mathbf{T}\}$.
- The transition function Δ is constructed as follows: For each state $u \in Q$ and alphabet symbol $\alpha \in \Sigma$, we use the congruence relation in order to find $v \in Q$ such that $u\alpha \sim v$. Then, we add the transition (u, α, v) in Δ.

Therefore, we can see that once a finite congruence is obtained for a function $\mathbf{f}$, it is straightforward to construct a DFA representation of the congruence. This implies that learning a DFA is reducing to learning the underlying Nerode congruence. We will use similar ideas in order to learn partial transducers in section 6. Algorithm 1 presents the pseudocode for the algorithm for getting a DFA from the congruence relation.

From congruence to transducer. In the case that the target function is a boolean function, once we construct the DFA representation of the congruence we have effectively recovered the entire function. However, in the case of a general function the DFA representation will only correspond to the syntactic congruence. Recovering the entire transducer corresponds to also computing the output function σ. We will discuss how this is achieved for different transducer models in chapter 6.

4.7 Partial Congruence

During the execution of the learning algorithm, various approximations to the congruence will be constructed and utilized by the learning algorithm. In this section we will formally define the class of intermediate approximations to the target congruence constructed by our algorithms. Such a definition is important in order to be able to analyze algorithms which operate given such approximate information.

Definition 6. For a congruence $\sim_{\mathbf{f}} \stackrel{\text{def}}{=} (\Phi, \mathcal{R})$ we define the partial congruence $\sim_{\mathbf{h}} \stackrel{\text{def}}{=} (\mathcal{H}, \mathcal{R}_{\mathcal{H}})$ of $\sim_{\mathbf{f}}$ as follows:

- $\mathcal{R}_{\mathcal{H}} \subseteq \mathcal{R}$ is a prefix-closed subset of $\mathcal{R}$.

- $\mathcal{H} \subseteq \Phi$ is a subset of Φ which is saturated for $\mathcal{R}_\mathcal{H}$.

In the case of the Nerode congruence, the partial congruence $(\mathcal{R}, \Phi)$ gives rise to an approximation $\mathbf{h}$ of $\mathbf{f}$ using the same DFA construction we described in the previous section. In the case of the syntactic congruence, we will also define the partial output function σ_h as follows:

Definition 7. Given a partial congruence $\sim_\mathbf{h}$, we define the partial output function $\sigma_\mathbf{h}$ as follows, for $u \in \Sigma^*$ and $a \in \Sigma$:

$$\sigma_\mathbf{h}(u, a) \stackrel{\text{def}}{=} \sigma_\mathbf{f}(r, a) \text{ where } r \text{ is the member of } \mathcal{R}_\mathcal{H} \text{ such that } r \sim_\mathbf{h} u \tag{4.5}$$

Thus, the choice of the output depends on the set $\mathcal{R}_\mathcal{H}$ of representatives which is also why $\mathcal{R}_\mathcal{H}$ is an integral part of the definition of $\sim_\mathbf{h}$. The effect of removing representatives and distinguishing predicates amounts to collapsing of certain equivalence classes. More formally:

Proposition 5. *Let $\sim_\mathbf{f} = (\Phi, \mathcal{R})$ be a congruence and $\sim_\mathbf{h} = (\mathcal{H}, \mathcal{R}_\mathcal{H})$ be a partial congruence of $\sim_\mathbf{f}$. Then the following properties hold:*

1. ***Persistence of congruence:*** *For all $u, v \in \Sigma^*$, if $u \sim_\mathbf{f} v$ then $u \sim_\mathbf{h} v$.*
2. ***Collapse of equivalence classes:*** *Let $s \in \mathcal{R} \setminus \mathcal{R}_\mathcal{H}$. Then, there exists $r \in \mathcal{R}_\mathcal{H}$ such that for all $u \in \Sigma^*$, if $u \sim_\mathbf{f} s$ then $u \sim_\mathbf{h} r$.*

Moreover, the following useful property follows trivially from the definition:

Proposition 6. *For all $r \in \mathcal{R}_\mathcal{H}$ and $a \in \Sigma$, we have that $\sigma_\mathbf{h}(r, a) = \sigma_\mathbf{f}(r, a)$.*

Given a partial congruence, the learning algorithms described in this thesis will iteratively extend the congruence by adding new equivalence classes until a saturated set of distinguishing predicates is obtained. We will now formally define an extension to a partial congruence.

Definition 8. Given a partial congruence $\sim_\mathbf{h} = (\mathcal{H}, \mathcal{R}_\mathcal{H})$, an extension to $\sim_\mathbf{h}$ is a tuple $(r, \phi) \in \Sigma^* \times \mathbb{P}$ such that:

1. $r \notin \mathcal{R}_{\mathcal{H}}$ and $\mathcal{R}_{\mathcal{H}} \cup \{r\}$ is a prefix-closed set.

2. Let $r_i \in \mathcal{R}_{\mathcal{H}}$ such that $r_i \sim_{\mathbf{h}} r$. Then, ϕ is a distinguishing predicate for r_i, r.

One can easily see that the above definition implies that the partial congruence defined as $(\mathcal{H} \cup \phi, \mathcal{R}_{\mathcal{H}} \cup \{r\})$ is also a partial congruence which contains one more equivalence class than $\sim_h$. The main invariant we will prove for the learning algorithms is that each counterexample will provide us with a valid extension to the current partial congruence until all equivalence classes are recovered.

4.8 Implementing a partial congruence

Implementing the congruence relation is usually performed in terms of the corresponding state machine of the automaton or the transducer. If two strings end up in the state then they are congruent with respect to the underlying congruence relation (either Nerode or Syntactic). However, when we compute the congruence relations based on distinguishing predicates and queries we will utilize different data strctures in order to efficiently compute equivalence between different strings with respect to a specific congruence. We will now explore two different data structures which are commonly used in variations of the L^* algorithm for this purpose, the observation table [14] and the classification tree [53] also sometimes referred as discrimination tree [51]. In order for these data structures to effectively implement a (partial) congruence we would like them to support the following basic functionalities:

1. **Equivalence checking:** The most basic property of any data structure implementing a congruence is to be able to decide equivalence between two strings. In other words, given $u, v \in \Sigma^*$ the data structure must be able to answer whether $u \sim_{\mathbf{h}} v$ with respect to the partial congruence $(\mathcal{R}, \Phi)$.

2. **Extension:** Given a partial congruence $(\mathcal{R}, \Phi)$ and an extension (r, ϕ) the data structure must be able to extend the partial congruence with the given extension.

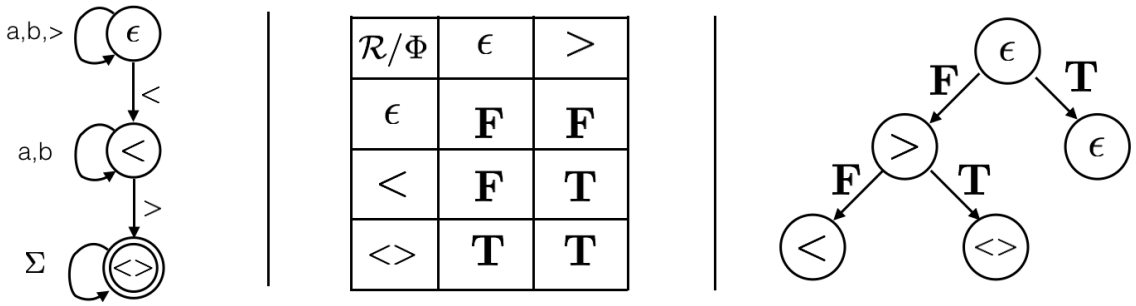

Figure 4-2: (Left:) A DFA accepting the language `<[^>]*>`. (Middle:) The corresponding observation table implementation of the Nerode congruence. (Right:) The corresponding classification tree implementation of the congruence.

We will now proceed to describe each data structure in more detail. In figure 4-2 we present a DFA (left) and the corresponding congruence implementation using an observation table (middle) and a classification tree (right).

4.8.1 Observation Table

The first data structure we will describe is called the observation table. This data structure was originally introduced by Angluin as part of the seminal L^* algorithm [14] and was later improved by Rivest and Schapire [75].

Definition 9. An *observation table* OT is a tuple (S, W, O) where:

- $S \subset \Sigma^*$ is the set of access strings (i.e. states).
- $W \subset \mathbb{P}$ is the set of *distinguishing predicates*.
- $O : S \times W \to \mathbb{B}$ is the *lookup function* with *row* indices from S and *column* indices from W.

For $s \in S$, let $\boldsymbol{row}(s) : W \to \mathbb{B}$ denote the row vector of the table corresponding to the access string s:

$$\boldsymbol{row}(s) \stackrel{\text{def}}{=} \{x \mapsto O(s, x) \mid x \in W\}$$

Deciding Equivalence

Equivalence in the observation table is checked by finding, for any string u, the class $r \in S$ such that $u \sim r$. It follows that we can easily check equivalence of arbitrary strings by finding the corresponding equivalence classes they belong. In order to find the equivalence class for a string u we add u as a new row in the table and fill the corresponding entries in order to get the row vector $\boldsymbol{row}(u)$. Afterwards, we find the corresponding row of a string $r \in S$ such that $\boldsymbol{row}(u) = \boldsymbol{row}(r)$.

Closedness of the observation table The astute reader might notice that, in the case that a partial congruence is represented by the observation table, it may be the case for a string u such that $\boldsymbol{row}(u) \neq \boldsymbol{row}(r)$ for all $r \in S$ because u is congruent to a yet unknown equivalence class! In this case, we say that the observation table is *not-closed*. During the execution of the L^* algorithm the strings for which equivalence will be checked will always be of the form $r\alpha$ where $r \in S$ and $a \in \Sigma$. If for all such strings we have that $\boldsymbol{row}(r\alpha) = \boldsymbol{row}(r')$ for some $r' \in R$ then the table is *closed* and a DFA model can be constructed. Otherwise, the string $r\alpha$ is used as a new equivalence class and is added as a row in the table. Notice that in this case the distinguishing predicates already present in the table are enough to distinguish between the new class and all other classes already present. In general, one can show that the number of distinguished predicates required to obtain a saturated set is at most $|\langle\Sigma^*\rangle| - 1$ and at least $\log_2 |\langle\Sigma^*\rangle|$.

Extending the observation table

Given an extension (r, ϕ), we extend the observation table by adding the distinguishing predicate ϕ in the set W and the new access string r in the set S.

4.8.2 The Classification Tree

The classification tree is a more efficient alternative to the observation table which was introduced by Kearns and Vazirani [53]. The main difference from the observation

table is that the distinguishing predicates are organized in a binary tree which allows more efficient evaluation of the congruence relation. We will now proceed to give a formal definition of the classification tree.

Definition 10. A classification tree $T = (V, L, E)$ is a binary tree such that:

- $V \subset \mathbb{P}$ is the set of nodes.
- $L \subset \Sigma^*$ is the set of leafs.
- $E \subset V \times V \times \mathbb{B}$ is the transition relation. For $(v, u, b) \in E$, we say that v is the parent of u and furthermore, if $b = \mathbf{T}$ (resp. $b = \mathbf{F}$) we say that u is the **T**-child (resp. **F**-child).

Intuitively, given any internal node $\phi \in V$, any leaf l_T reached by following the **T**-child of ϕ can be distinguished from any leaf l_F reached by the **F**-child using ϕ.

Initialization

To initialize the CT data structure, we use a query on the empty word ϵ. Then, we create a CT with two nodes, a root node labeled with ϵ and one child also labeled with ϵ. The child of the root is either a **T**-child or **F**-child, according to the result of the query on ϵ.

Deciding equivalence with the `sift` operation

Reducing a string $s \in \Sigma^*$ to the corresponding equivalence class is perfomed using an operation called `sift`. The `sift`(s) operation performs the following steps:

1. Set the current node to be the root node of the tree and let ϕ be the label at the root. Check the value of the predicate $\phi(s)$.
2. Let $b = \phi(s)$. Select the b-child of the current node and repeat step 2 until a leaf is reached.
3. Once a leaf is reached, return the access string with which the leaf is labelled.

Note that, until both children of the root node are added, we will have inputs that may not end up in any leaf node. In these cases our `sift` operation will return $\perp$ and, in the case of a learning algorithm, we will add the queried input as a new leaf in the tree similarly with the closedness property of the observation table. However, in the case of the tree, having an undefined target equivalence class can occur at most one time (afterwards, all paths will lead to a known equivalence class), while in the case of the observation table, closedness can be violated arbitrarily often.

Extending the classification tree

Given an extension (r, ϕ) we proceed to extend the partial congruence represented by the classification tree as follows: Let r' be an access string such that $r \sim_{\mathbf{h}} r'$ based on the current classification tree. Then, we replace the leaf holding r' with a new subtree with three nodes such that

- The root of the subtree is labelled with the distinguishing predicate ϕ.
- Assume without loss of generality that $\phi(r) = \mathbf{T}$. Then, we have that the **T**-child of the root is labelled with r and the **F**-child with r'. If $\phi(r) = \mathbf{F}$, then the opposite labels are assigned to each child.

Chapter 5

Learning Deterministic Finite Automata

We will start this chapter with the presentation of the L^*algorithm for learning deterministic finite automata using membership and equivalence queries.

The classical way under which the L^*algorithm is presented is as an algorithm which successively discovers new states in the target DFA. In this section we will study the L^*algorithm as a congruence learning algorithm. As we saw above, inferring the Nerode congruence while having query access to the target function, allows one to easily reconstruct a DFA representation of the target function (if such finite representation exists).

5.0.1 Technical Description.

The algorithm starts with an equivalence relation containing a single equivalence class accessed by the empty string (recall that the set of representatives is prefix-closed) and a single distinguishing predicate $\mathcal{P}_{\epsilon}^{\mathtt{dom}}$ distinguishing between final and non-final states. The congruence can be implemented using any appropriate data structure discussed in the previous section such as the classification tree or the observation table. Given any such congruence we use either a classification tree or an observation table data structure to represent it and then construct a DFA. Finally, once we built

the DFA model, we submit an equivalence query. If the model is correct the algorithm terminates. On the other hand, given a counterexample we invoke the counterexample processing algorithm described below which utilizes the counterexample in order to extract a new representative and distinguishing predicate which will then extend the partial congruence.

The full pseudocode of the algorithm is available in algorithm 2 while the counterexample processing routine in algorithm 3.

5.0.2 Processing Counterexamples.

The main idea behind using a counterexample s to extend the partial congruence with a new equivalence class (in terms of a new representative and distinguishing predicate) is the following:

Let $\gamma_i = r_i \cdot s[i+1..]$ where $r_i \sim_{\mathbf{h}} s[..i]$. In other words, γ_i is obtained by taking the prefix of length i of s and finding the representative based on $\sim_{\mathbf{h}}$. Then, we concatenate the resulting r_i with the remaining suffix and check the value of the string γ_i using a query to the target function. Notice that $\mathbf{f}(\gamma_0) = \mathbf{f}(s)$ and moreover, $\mathbf{f}(\gamma_{|s|}) \neq \mathbf{f}(\gamma_{|s|})$. Therefore, there exists a breakpoint j where $\mathbf{f}(\gamma_{j-1}) = \mathbf{f}(s)$ but $\mathbf{f}(\gamma_j) \neq \mathbf{f}(s)$. We claim that $r_{j-1}s[j]$ is a new representative accessing an undiscovered equivalence class (or equivalently state) and $s[j+1..]$ is a new distinguishing string which can separate strings in the equivalence class r_j from strings in the new equivalence class $r_{j-1}s[j]$.

The following lemma formally proves that the counterexample processing algorithm will provide us with a valid extension to the current partial congruence.

Lemma 2. *The tuple* $(r_{j-1}s[j], s[j+1..])$ *is an extension to the partial congruence* $(\mathcal{R}, \Phi)$.

Proof. The fact that $\mathcal{R}_{\mathcal{H}}$ remains prefix closed is trivial. Regarding distinguishability notice that, by definition of γ_j we have that $\mathbf{f}(r_{j-1}s[j]s[j+1..]) \neq \mathbf{f}(r_j s[j+1..])$ but we have that $r_{j-1}s[j] \sim_{\mathbf{h}} r_j$ and therefore $\mathcal{P}^{\mathtt{dom}}_{s[j+1..]}$ is distinguishing for $r_{j-1}s[j]$ and r_j. □

Optimizing counterexample processing with binary search. Searching for the breakpoint j can be done by sequentially running the process of generating γ_i and checking whether the breakpoint is found for all $i \leq |s|$. However, we are not necessarily interested in obtaining the smallest breakpoint j. Any index j such that $\mathbf{f}(\gamma_{j-1}) = \mathbf{f}(s)$ and $\mathbf{f}(\gamma_j) \neq \mathbf{f}(s)$ will suffice. Therefore, in order to speed up the search and reduce the number of queries we can use the following binary search process: We start at the index $j = |s|/2$ and check whether $\mathbf{f}(\gamma_j) = \mathbf{f}(s)$. If $\mathbf{f}(\gamma_j) = \mathbf{f}(s)$ we recursively apply the same process on the index $3|s|/4$ or in the index $|s|/4$ on the opposite case until we have found an index j such that j is a valid breakpoint. Algorithm 3 presents the binary search counterexample processing algorithm.

Remarks. We should point out that certain properties of the congruence are not really fundamental and can be implemented in different ways. For example, the set of representatives can be constructed in a way that is not prefix closed while the set of strings used by distinguishing predicates can be constructed in order to be suffix closed [53]. Such variations on the properties of the congruence are affected by the details of the counterexample processing algorithm, however the general principle of detecting a breakpoint which provides both the new representative as well as the new distinguishing predicate is a common theme across all variations.

5.0.3 Correctness and Complexity

The correctness and the complexity of the L^* algorithm is summarized in the following theorem:

Theorem 2. *Let $A = (Q, q_0, F, \Delta)$ be a DFA over an alphabet Σ. Then, the L^* algorithm will learn A using $O(|Q|^2|\Sigma| + |Q| \log m)$ membership and $|Q|$ equivalence queries, where m is the length of the longest counterexample provided to the algorithm.*

Proof. The correctness and termination of the algorithm follow from lemma 2 since, after at most $|Q|$ counterexamples all equivalence classes will be recovered and a correct DFA model will be constructed. Since the algorithm starts with a partial congruence with a single equivalence class and a single distinguishing predicates and

extends the congruence with an additional equivalence class (i.e. state) with each counterexample, it follows that $|Q|$ counterexamples are required in order to recover all equivalence classes. Processing each counterexample requires $\log m$ queries using the binary search algorithm described above. Finally, constructing a DFA model requires $O(|Q||\Sigma|)$ queries using either the observation table or the classification tree implementation of the partial congruence relation. □

Algorithm 2 L^*Algorithm

Require: **f** is the target function, $\mathcal{E}$ is an equivalence oracle

```
function L*(f, E)
    R ← {ε}
    Φ ← {P_ε^dom}
    h ← getDFA(R, Φ)
    while E(h) ≠ T do
        s ← getCounterexample()
        (r, φ) ← processCounterexample(s)
        R ← R ∪ {r}
        Φ ← Φ ∪ {φ}
        h ← getDFA(R, Φ)
    return h
```

Algorithm 3 Counterexample processing algorithm

Require: **h** is the partial congruence, s is the counterexample.

```
function processCounterexample(h, s)
    L ← 0
    R ← |s| − 1
    while L ≠ R do
        m ← (L + R)/2
        if f(γ_m) = f(s) then
            L ← m + 1
        else
            H ← m − 1
    return (r_m s[m], P^dom_{s[m+1..]})
```

Chapter 6

Learning Deterministic Transducers

In this chapter we will describe novel L^*-style algorithms for learning transducer models. We will start by presenting a non-trivial extension of the L^*algorithm for partial deterministic transducers. Partiality introduces non-trivial challenges in learning since concepts such as distinguishability become more difficult to verify in the setting of partial functions. In order to address this challenge, we extend to concept of black-box distinguishability for transducers and introduce a novel *output label inference* algorithm which can be used in order to learn the output labels of a transducer given it's state machine structure. Afterwards, we introduce a generalized, indexed congruence relation which can used as the basis of defining canonical non-deterministic transducer models, and we develop such a class called visibly non-deterministic transducers and show that this class can also be learned efficiently. As we will demonstrate in our evaluation this class is effective in modelling many string transformation functions which are commonly encountered in Web applications such as calls to the `preg_replace` function and other similar code constructs.

6.1 Overview

Figure 6-1 presents the overall algorithmic learning framework under which our learning algorithms operate. In a nutshell, our algorithms work by first inferring an approximation of the underlying state machine (or equivalence relation) of the target

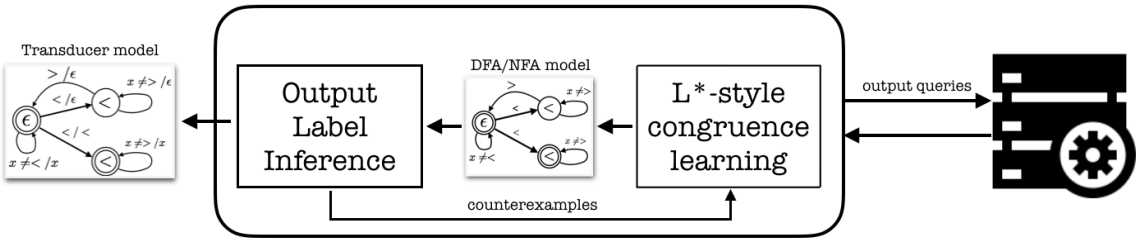

Figure 6-1: The overall algorithmic learning framework.

transducer. Given such a state machine we describe, in section 6.3, an algorithm which can recover the output labels of the target transducer. This algorithm is very general and can be applied even to non-deterministic models as long as they are functional. Finally, given a counterexample, we proceed to refine the state machine by adding previously undiscovered states.

6.2 Learning Total Transducers

As a warm-up for the T^* algorithm let's consider the problem of adapting the L^* algorithm in the case of deterministic transducers. As we will see the fact the target function is total allows us to easily evaluate the syntactic congruence and therefore adapting the L^* algorithm in this case is straightforward.

As we describe in our overall algorithmic framework in figure 6-1, we will split learning into two components, a congruence learning component which allows us to recover the syntactic congruence and an output label inference component which, given the partial syntactic congruence, infers the corresponding output function. We will now describe the two components in detail.

6.2.1 Learning the Syntactic Congruence

In order to learn the syntactic congruence we will use the identical high level L^* algorithm from the previous section including the counterexample processing method introduced there. This shows the advantage of describing the L^* algorithm in terms of a generic congruence. However, we will address certain aspects of the algorithm

which are different.

Distinguishability. In the original L^* algorithm the distinguishing predicates constructed were all domain distinguishing predicates. However, in the case of total transducers, output distinguishing predicates need to be used in order to distinguish between different equivalence classes (i.e. states).

6.2.2 Learning the output function $\sigma_{\mathbf{f}}$

Once, we have a partial syntactic congruence, we proceed to use it in order to infer the output function $\sigma_{\mathbf{f}}$. Again, the fact that the transducer is total makes this task straightforward as for all $r \in \mathcal{R}, \alpha \in \Sigma$ we have that $\sigma_{\mathbf{f}}(r, \alpha) = \mathbf{f}(r) \setminus \mathbf{f}(r\alpha)$. As we will see in the next section, computing the output function becomes highly non-trivial once we introduce partiality in the target functions.

6.2.3 The Algorithm

As we mention above, the algorithm follows the same high level description as the L^* algorithm, shown in algorithm 2. However, since the target function is total, the empty string is not distinguishing between any states and therefore, we start with the congruence $(\mathcal{R}, \Phi) = (\{\epsilon\}, \emptyset)$. Moreover, given a partial syntactic congruence, instead of invoking the `getDFA` algorithm, we first invoke the `getDFA` to obtain a state machine and afterwards, we use the algorithm described above to add the corresponding output in each transition.

The second difference is the way we process counterexamples. As in the L^* algorithm, given a counterexample s, we generate the strings γ_i in the same way. However, instead of checking whether $\mathbf{f}(\gamma_i) = \mathbf{f}(s)$, we would like to only compare the suffix of the generated output in order to distinguish according to the syntactic congruence. Therefore, we perform the test $\mathbf{f}(r_i) \setminus \mathbf{f}(\gamma_i) = \mathbf{f}(s[..i]) \setminus \mathbf{f}(s)$. After we obtain the breakpoint j, we generate the output distinguishing certificate parametrized by $(s[j+1..], \epsilon, \mathbf{f}(s[..j]) \setminus \mathbf{f}(s))$.

Given the above changes, the remaining part of the L^* algorithm remains the same

and the correctness proof and complexity follow in the same fashion.

6.3 Output Label Inference

In this section we introduce a central component of all our learning algorithms. Our goal in this section is to define and solve the Output Label Inference (`OLI`) problem. An input to the problem is the syntactic congruence $\sim_{\mathbf{f}}$ corresponding to $(\Phi, \mathcal{R})$. In its full generality the input is allowed to be a partial congruence of $\sim_{\mathbf{f}}$ but it is useful to explain first the algorithm in terms of the full syntactic congruence. The essential part of $\sim_{\mathbf{f}}$ used by `OLI` is the *DFA induced by* $\sim_{\mathbf{f}}$, the DFA has $\mathcal{R}$ as its set of states, the initial state is ϵ, final state set is $\mathcal{R} \cap \boldsymbol{dom}(\mathbf{f})$, and the transition function is $\delta(q, a) = p$ for $q, p \in \mathcal{R}$ and $a \in \Sigma$ where $p \sim_{\mathbf{f}} q{\cdot}a$. The purpose of the algorithm is to calculate the output function $\sigma_{\mathbf{f}}$. More concisely:

`OLI:` The input parameters are query access to $\mathbf{f}$ and the DFA induced by $\sim_{\mathbf{f}}$. The task is to compute the output function $\sigma_{\mathbf{f}}$ and thus the finite state transducer $\mathcal{T}_{\mathbf{f}}$.

In the transducer formulation of the problem, we are given as input the target transducer with the output labels hidden and the ability to query the target transducer with any input of our choice. The goal is to efficiently recover the output labels of the target transducer in ENF.

6.3.1 `OLI` Algorithm

Before starting to delve into the details of the `OLI` algorithm, we will define the set of enabling suffixes, a concept which will be important for the operation of the algorithm.

Definition 11. For an $r \in \mathcal{R}$ we say that $e \in \Sigma^*$ is an *enabling suffix* if $re \in \boldsymbol{dom}(\mathbf{f})$ and for all prefixes $p \in \boldsymbol{prefixes}(e) \setminus \{e, \epsilon\}$ we have that $rp \notin \boldsymbol{dom}(\mathbf{f})$.

Our algorithm will construct us a suffix-closed set of enabling suffixes E containing one enabling suffix for each representative. For $r \in \mathcal{R}$ we will denote by $e_r \in E$ the

enabling suffix for r. Suffix-closedness is enforced by the following property for all $e_r \in E$:

$$e_r = \begin{cases} \epsilon, & \text{if } r \in \boldsymbol{dom}(\mathbf{f}), \\ e_r[1] \cdot e_{\delta(r, e_r[1])}, & \text{otherwise.} \end{cases} \tag{6.1}$$

Finally, if there doesn't exist a suffix e_r such that $r \cdot e_r \in \boldsymbol{dom}(\mathbf{f})$, i.e. r is accessing a dead-end state, then we define $e_r = \perp$. Our algorithm for solving the `OLI` problem is based on a simple formula for computing the output function for any state. More specifically, let $u \in \mathcal{R}, \alpha \in \Sigma$. Then we have that:

$$\sigma(u, \alpha) = \widehat{\mathbf{f}}(u) \setminus (\mathbf{f}(u \cdot \alpha \cdot e_{\delta(u,\alpha)}) \nearrow \mathbf{f}_{\delta(u,\alpha)}(e_{\delta(u,\alpha)})) \tag{6.2}$$

Moreover, the following lemma shows a way to compute the prefix based on the values of the suffix:

Lemma 3. *For any sequence $u \in \mathcal{R}$ we have that:*

$$\widehat{\mathbf{f}}(u) = \bigsqcap_{\alpha \in \Sigma} (\mathbf{f}(u\alpha \cdot e_{\delta(u,\alpha)}) \nearrow \mathbf{f}_{\delta(u,\alpha)}(e_{\delta(u,\alpha)})) \tag{6.3}$$

Proof. We have

$$\begin{aligned} \bigsqcap_{\alpha \in \Sigma} (\mathbf{f}(u\alpha \cdot e_{\delta(u,\alpha)}) \nearrow \mathbf{f}_{u\alpha}(e_{\delta(u,\alpha)})) &= \bigsqcap_{\alpha \in \Sigma} (\widehat{\mathbf{f}}(u) \cdot \sigma(u, \alpha)) \\ &= \widehat{\mathbf{f}}(u) \bigsqcap_{\alpha \in \Sigma} \sigma(u, \alpha) \\ &= \widehat{\mathbf{f}}(u) \end{aligned}$$

□

Using lemma 3 we conclude that the only unknown in equation 6.2 is the value of $\mathbf{f}_r(e_r)$ for all access strings $r \in \mathcal{R}$. If r is a final state, then we can simply compute $\mathbf{f}_r(e_r) = \mathbf{f}(r) \setminus \mathbf{f}(r \cdot e_r)$. However, in the case that r is a non-final state, the computation of $f_r(e_r)$ becomes non-trivial.

At a high level, our `OLI` algorithm will iteratively approximate the value of $\mathbf{f}_r(e_r)$

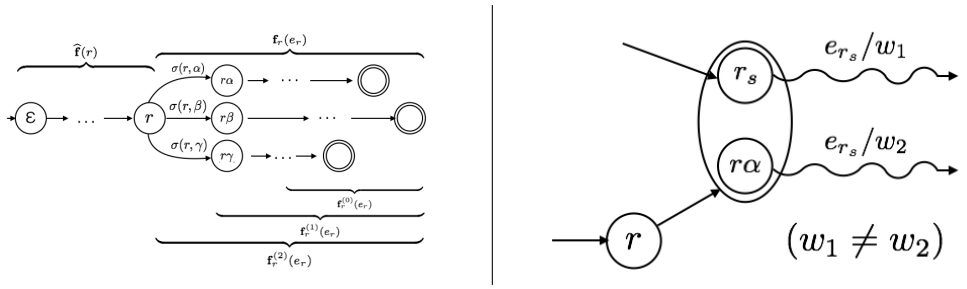

Figure 6-2: (Left:) Iterative approximations of the $\mathbf{f}_r(e_r)$ value by the `OLI` algorithm. (Right:) Demonstration of a vulnerable transition $(r, a, r_s) \in \mathcal{R}_\mathcal{H} \times \Sigma \times \mathcal{R}_\mathcal{H}$.

for each different state $r \in \mathcal{R}$ until a fix-point is reached. More formally, the algorithm works as follows:

1. Initialize $\mathbf{f}_r^{(0)}(e_r) = \epsilon$ and $\mathbf{f}_r^{(-1)}(e_r) = \perp$ for all $r \in \mathcal{R}$.
2. While $\exists r \in \mathcal{R} : \mathbf{f}_r^{(i)}(e_r) \neq \mathbf{f}_r^{(i-1)}(e_r)$ repeat the following steps sequentially for all $r \in \mathcal{R}$:
   (a) $\widehat{\mathbf{f}}^{(i+1)}(r) = \bigsqcap_{\alpha\in\Sigma}(\mathbf{f}(r\alpha{\cdot}e_{\delta(r,\alpha)}) \nearrow \mathbf{f}^{(i)}_{\delta(r,\alpha)}(e_{\delta(r,\alpha)}))$.
   (b) $\forall\alpha \in \Sigma : \sigma^{(i+1)}(r,\alpha) = \widehat{\mathbf{f}}^{(i+1)}(r) \nwarrow (\mathbf{f}(r\alpha{\cdot}e_{\delta(r,\alpha)}) \nearrow \mathbf{f}^{(i)}_{\delta(r,\alpha)}(e_{\delta(r,\alpha)}))$.
   (c) $\mathbf{f}_r^{(i+1)}(e_r) = \widehat{\mathbf{f}}^{(i+1)}(r) \nwarrow \mathbf{f}(r{\cdot}e_r)$.
3. Return $\sigma^{(i)}$ as the result.

Note that in the return value $\sigma(r,\alpha) = \perp$ whenever $\delta(r,\alpha)$ is a deadend.

6.3.2 Correctness and Complexity

In this section our main goal is to prove the correctness of the OLI algorithm. More specifically, we will prove the following theorem:

Theorem 3. *The `OLI` algorithm on input a congruence $(\Phi, \mathcal{R})$ with $n = |\mathcal{R}|$ over an alphabet Σ and query access to the function $\mathbf{f}$ will recover the output function using $|\Sigma|n$ output queries and in time $O(n^2|\Sigma|m)$ where $m = \max_{r\in\mathcal{R},\alpha\in\Sigma} |\sigma(r,\alpha)|$.*

The main idea in our analysis is to define a class of states for which the value of $\mathbf{f}_r(e_r)$ will be computed correctly from the first iteration. We will show that such

$r \in \mathcal{R}$ necessarily exist and moreover, that the correct values computed for these states will eventually propagate in subsequent iterations into all the other states. We will start with the basic definition of the BAD state .

Definition 12. A state $r \in \mathcal{R}$ is called α-BAD if Σ can be partitioned into two sets $\mathcal{G}$ and $\mathcal{B}$ such that:

- For every $\beta \in \mathcal{B}$ we have that $\sigma(r, \beta) = \epsilon$ and $\alpha \preccurlyeq \mathbf{f}_r(e_r)$.
- For every $\beta \in \mathcal{G}$ we have that $\alpha \preccurlyeq \sigma(r, \beta)$.

An $r \in \mathcal{R}$ is called BAD when there exists $\alpha \in \Sigma$ such that r is α-BAD.

The intuition behind the definition of BAD states, is that, for this class of $r \in \mathcal{R}$, the initial computation of $\widehat{\mathbf{f}}^{(1)}(r)$ will result in over-approximating the value of $\widehat{\mathbf{f}}(r)$. As we will show now, for all $r \in \mathcal{R}$ which are not BAD the value of $\mathbf{f}_r(e_r)$ will be computed correctly from the first iteration.

Lemma 4. *If an $r \in \mathcal{R}$ is not BAD then $\mathbf{f}_r^{(1)}(e_r) = \mathbf{f}_r(e_r)$.*

Proof. Since we initially start with $\mathbf{f}_r^{(0)}(e_r) = \epsilon$ for all $r \in \mathcal{R}$, it follows the computation of $\widehat{\mathbf{f}}(r)$ is equivalent to

$$\widehat{\mathbf{f}}(r) = \bigsqcap_{\alpha \in \Sigma} \mathbf{f}(r \cdot \alpha \cdot e_{\delta(r,\alpha)}) \tag{6.4}$$

Since r is not BAD one of the following holds:

1. There exist $\alpha, \beta \in \Sigma$ such that $\sigma(r, \alpha) \sqcap \sigma(r, \beta) = \epsilon$ and both $\sigma(r, \alpha)$ and $\sigma(r, \beta)$ are non-empty. It follows easily that in this case $\widehat{\mathbf{f}}(r)$ will be computed correctly.
2. There exists $\alpha, \beta \in \Sigma$ such that $\sigma(r, \alpha) = \sigma(r, \beta) = \epsilon$ but $\mathbf{f}_{r\alpha}(e_{\delta(r,)\alpha}) \sqcap \mathbf{f}_{r\beta}(e_{\delta(r,\beta)}) = \epsilon$. Again, in this case, it follows that $\widehat{\mathbf{f}}(r)$ will be computed correctly.

□

Our next task is to prove that even if an r is BAD, if any neighboring r_α is not BAD the subsequent iteration of the algorithm will correctly set the value for r. We start with the following lemma:

Lemma 5. *For any $i > 0$ and any $r \in \mathcal{R}$ we have that $\mathbf{f}_r(e_r) \succcurlyeq \mathbf{f}_r^{(i)}(e_r)$.*

Proof. By induction in i. For $i = 0$ the result holds trivially. For the inductive step, notice that the inductive hypothesis implies that $\widehat{\mathbf{f}}(r) \preccurlyeq \widehat{\mathbf{f}}^{(i)}(r)$ from which the result follows using equation 6.2. □

Lemma 6. *Let $r \in \mathcal{R}$ be α-BAD and assume that there exists $\beta \in \mathcal{B}$ with $r_b = \delta(r, \beta)$, such that $\mathbf{f}_{r_\beta}^{(i)}(e_{r_\beta}) = \mathbf{f}_{r_\beta}(e_{r_b})$. Then, $\mathbf{f}_r^{(i+1)}(e_r) = \mathbf{f}_r(e_r)$.*

Proof. We have that

$$\begin{aligned}
\widehat{\mathbf{f}}^{(i+1)}(r) &= \prod_{\alpha\in\Sigma}(\mathbf{f}(r\alpha{\cdot}e_{\delta(r,\alpha)}) \nearrow \mathbf{f}^{(i)}_{\delta(r,\alpha)}(e_{\delta(r,\alpha)})) \\
&= (\widehat{\mathbf{f}}(r){\cdot}\sigma(r,\beta)) \sqcap \prod_{\alpha\in\Sigma\wedge\alpha\neq\beta}(\mathbf{f}(r\alpha{\cdot}e_{\delta(r,\alpha)}) \nearrow \mathbf{f}^{(i)}_{\delta(r,\alpha)}(e_{\delta(r,\alpha)})) \\
(\beta \in \mathcal{B} \implies \sigma(r,\beta) = \epsilon) &= (\widehat{\mathbf{f}}(r) \sqcap \prod_{\alpha\in\Sigma\wedge\alpha\neq\beta}(\mathbf{f}(r\alpha{\cdot}e_{\delta(r,\alpha)}) \nearrow \mathbf{f}^{(i)}_{\delta(r,\alpha)}(e_{\delta(r,\alpha)})) \\
(\textit{lemma 5}) &= \widehat{\mathbf{f}}(r)
\end{aligned}$$

Therefore, in the $i+1$ iterations of the algorithm, the value of $\widehat{\mathbf{f}}(r)$ will be computed correctly and thus, $\mathbf{f}_r^{(i+1)}(e_r) = \mathbf{f}_r(e_r)$. □

At this point, we have that the correct results will eventually propagate and correct the values in the BAD states. The final piece is to prove that for every BAD state there exists a suffix which leads to a state which is not BAD showing that eventually all BAD states will be fixed.

Lemma 7. *Let $r \in \mathcal{R}$ be a state such that r is α-BAD. Then, for every $\beta \in \mathcal{B}$ we have that $\delta(r, \beta)$ is either α-BAD or not BAD.*

Proof. Assume that $r_b = \delta(r, \beta)$ is γ-BAD for some $\gamma \in \Sigma$. Then, by definition 12, we have that $\gamma \preccurlyeq \mathbf{f}_{r_b}(e_{r_b})$, a contradiction, since $\alpha \preccurlyeq \mathbf{f}_{r_b}(e_{r_b})$ since $\beta \in \mathcal{B}$. □

For the following lemma we will denote by $\mathcal{G}_r, \mathcal{B}_r$ the partition for the BAD state r.

Lemma 8. *Let r be an α-BAD state. Then there exists a sequence w_r, with $|w_r| \leq |\mathcal{R}|$ such that:*

1. *$\delta(r, w_r)$ is not BAD.*
2. *For all $p_i \in \textbf{\textit{prefixes}}(w_r)$ we have that $\delta(r, p_i)$ is α-BAD and moreover, $p_i[|p_i|] \in \mathcal{B}_{\delta(r,p_i)}$.*

Proof. Since r is α-BAD, by definition of $\widehat{\mathbf{f}}(r)$, there exists a w such that $rw \in \textbf{\textit{dom}}(\mathbf{f})$ and moreover, $\gamma \preccurlyeq \mathbf{f}(rw)$ for some symbol $\gamma \neq \alpha$. Let $r_\gamma = \delta(r, w_\gamma)$ be such that $w_\gamma \preccurlyeq w$ and $\gamma \preccurlyeq \sigma(r_\gamma, w_\gamma[|w_\gamma|])$. Moreover let $r_i = \delta(rw_\gamma[1..i],)$ for $i < |w_\gamma|$. It follows that r_γ is either γ-BAD or not BAD.

Now, consider the path starting at r and extending to r_γ through all prefixes of w_γ:

$$r \xrightarrow{w_\gamma[1]} r_1 \xrightarrow{w_\gamma[2]} r_2 \xrightarrow{w_\gamma[3]} \cdots \xrightarrow{w_\gamma[|w_\gamma|-1]} r_\gamma \xrightarrow{w_\gamma[|w_\gamma|]} \cdots \tag{6.5}$$

Notice that for all r_i we have that $\sigma(r_i, w_\gamma[i]) = \epsilon$ because the first output symbol is produced by $\delta(r, w_\gamma)$. Moreover, starting from r_1 we have, by lemma 7, that every r_i is either α-BAD or not BAD. Finally, notice that r_γ cannot be α-BAD because $\gamma \preccurlyeq \sigma(r_\gamma, w_\gamma[|w_\gamma|])$. Putting the above together, we get that there exists a k such that for all $i < k$ we have that r_i is α-BAD and moreover, r_k is not BAD.

Therefore, by setting $w_r = w[..k]$ we get the result. To show that $|w_r| \leq |\mathcal{R}|$ we notice that since there are at most $n = |\mathcal{R}|$ states, after at most n steps, we will have a state repeating and therefore we can trim the path accordingly. □

We are finally ready to prove theorem 3:

Proof of theorem 3. Consider any $r \in \mathcal{R}$. If r is not BAD then by lemma 4, we have that $\mathbf{f}_r^{(1)}(e_r) = \mathbf{f}_r(e_r)$. Now consider an $r \in \mathcal{R}$ such that r is α-BAD. By lemma 8 there exists a w with length $|w| \leq |\mathcal{R}|$ such that $\delta(r, w)$ is not BAD and $\delta(r, w[..|w|-1])$ is α-BAD. By lemma 6, in the subsequent iteration of $i = 2$, we have that $\mathbf{f}^{(i)}_{\delta(r,w[..|w|-1])}(e_{\delta(r,w[..|w|-1])})$ will be computed correctly, and therefore, by reapplying lemma 6 we have that after at most $n = |\mathcal{R}|$ iterations we will have that $\mathbf{f}_r^{(n)}(e_r) = \mathbf{f}_r(e_r)$ will be computed correctly for all $r \in \mathcal{R}$ and the algorithm will

terminate. Regarding the time and query complexity of the algorithm, notice initially that we have proved that within at most $n = |\mathcal{R}|$ iterations, the `OLI` algorithm will terminate. Within each iteration the common prefix of $|\Sigma|$ sequences of length at most nm which can be done in time linear in $|\Sigma|nm$. Therefore, the total time complexity is $O(n^2|\Sigma|m)$. Moreover, for each state the algorithm performs $|\Sigma|$ output queries, for a total of $|\Sigma|n$ queries. □

The `OLI` problem and word equations. An alternative approach to solve the `OLI` problem would be to pose it as a word equation problem. More specifically, since by definition we have that the output of the function is a series of concatenations of the output function, we can create a single variable for each output function value and attempt to solve the following linear system of word equations: where, for $r \in \mathcal{R}$ and $\alpha \in \Sigma$ the variable χ_r^α denotes the variable corresponding to the value of $\sigma(r, \alpha)$. One can easily prove that any solution to the above system of equations would be a valid assignment for the output labels in the corresponding transducer formulation of the function. However, not all these solutions would correspond to a valid output function as defined in this paper. Therefore, the following additional constraint has to be added into the system which enforces a unique solution which corresponds to the solution found by the `OLI` algorithm presented in this section:

$$\forall r \in \mathcal{R} : \prod_{\alpha \in \Sigma} \chi_r^\alpha = \epsilon \tag{6.6}$$

Notice that the addition of this constrains still give a linear system of word equations. Unfortunately, solving linear systems of word equations is an NP-Complete problem []. While more restricted fragments are known to be solvable in polynomial time, we didn't find any efficiently solvable fragments which can capture the instances generated by the `OLI` algorithm. Under this formulation our `OLI` algorithm can be viewed as a word equation solver for a fragment of linear word equations as defined by the problem definition. An interesting future work direction is to explore applications of similar ideas in the context of word equation solving.

6.3.3 Robust Output Label Inference

In the setting of learning transducers, when we invoke the `OLI` algorithm the input given to the algorithm will be an incomplete representation of the state machine of the transducer, i.e. a partial congruence. In order to be able to utilize `OLI` in the context of a transducer learning algorithm this aspect needs to be taken into account.

In this section, we will describe a variation of the `OLI` algorithm which adds a robustness check in each iteration in order to produce a *certificate* of the current approximation computed by the algorithm. This certificate can be used to verify the current approximation and will play a significant role when we analyze the behavior of the OFL algorithm under partial congruences. Recall that, for a string $r \in \mathcal{R}$ and an enabling suffix e_r, we say that $\mathbf{C}(r) \in \Sigma^*$ is a certificate for w if $\mathcal{J}(r, e, \mathbf{C}(r)) = w$.

Our goal in this section is to develop an algorithm such that, for each approximation $\mathbf{f}_r^{(i)}(e_r)$ we will also obtain a certificate $\mathbf{C}^{(i)}(r)$ for $\mathbf{f}_r^{(i)}(e_r)$ proving the current approximation. Before describing the internals of the certificate generation algorithm, we will define the *reason* for each approximation computed by the `OLI` algorithm.

Definition 13. For $r \in \mathcal{R}$ and an approximation $\mathbf{f}_r^{(i+1)}(e_r)$ computed by the `OLI` algorithm, we define the *reason* for the approximation as

$$\mathbf{R}^{(i)}(r) = \arg\min_{\alpha \in \Sigma} |\mathbf{f}(r\alpha \cdot e_{\delta(r,\alpha)}) \nearrow \mathbf{f}^{(i)}_{\delta(r,\alpha)}(e_{\delta(r,\alpha)})| \tag{6.7}$$

Intuitively, the certificate for each approximation is computed as follows: During the first iteration of the `OLI` algorithm, the common prefix between all equations for each state $r \in \mathcal{R}$ will be removed to yield the first update. In this case, the certificate proving the first approximation is simply a suffix $e_\delta(r, \alpha)$ for some $\alpha \in \Sigma$ such that $\mathbf{f}(re_{\delta(r,\alpha)}) \sqcap \mathbf{f}(re_{\delta(r,e_r)}) = \mathbf{f}_r^{(1)}(e_r)$. Afterwards, the newly computed approximations will be used to further prune the common prefix in successive iterations of the algorithm. In this case, the reason function is used to determine which transition (symbol) caused a new update in the current approximation.

More formally, the certificate for each successive approximation of the `OLI` algorithm is defined as follows:

Definition 14. For $i \geq 0$ we define by $\mathbf{C}^{(i)}(r)$ to be the certificate for $\mathbf{f}_r^{(i)}(e_r)$, defined as follows.

$$\mathbf{C}^{(0)}(r) = \perp \tag{6.8}$$

For $i = 1$, let $\beta \in \Sigma$ be such that $\widehat{\mathbf{f}}^{(1)}(r) = \mathbf{f}(r{\cdot}e_r) \sqcap \mathbf{f}(r\beta{\cdot}e_{\delta(r,\beta)})$[1]. Then, we have that:

$$\mathbf{C}^{(1)}(r) = \begin{cases} \mathbf{C}^{(0)}(r), & \text{if } \mathbf{f}_r^{(1)}(e_r) = \mathbf{f}_r^{(0)}(e_r), \\ \beta{\cdot}e_{\delta(r,\beta)}, & \text{otherwise.} \end{cases} \tag{6.9}$$

For $i > 1, r \in \mathcal{R}$, define $\rho = \mathbf{R}^{(i)}(r)$. Then we have that

$$\mathbf{C}^{(i)}(r) = \begin{cases} \mathbf{C}^{(i-1)}(r), & \text{if } \mathbf{f}_r^{(i)}(e_r) = \mathbf{f}_r^{(i-1)}(e_r), \\ \rho{\cdot}\mathbf{C}^{(i-1)}(\delta(r,\rho)), & \text{otherwise.} \end{cases} \tag{6.10}$$

Note that the computation of the certificates can be incorporated within the main execution loop of the `OLI` algorithm. We call this augmented version of `OLI`, the Robust Output Label Inference (`ROLI`) algorithm. The overall `ROLI` algorithm is depicted in algorithm 4. The following theorem states the correctness of the certificate generation algorithm:

Theorem 4 (Certificate Validity)**.** *For every $i \geq 0$, if $\mathbf{C}^{(i)}(r) \neq \perp$, then $\mathbf{C}^{(i)}(r)$ is a certificate for $\mathbf{f}_r^{(i)}(e_r)$.*

6.3.4 The `OLI` algorithm under partial congruences

Now that we have described the full version of the `ROLI` algorithm we will proceed to analyze it's behavior when a partial congruence is given as input. More specifically, we consider the following problem: The algorithm is given output query access to the transducer $\mathbf{f}$ as in the normal `OLI` setting however, the algorithm is given access to a partial congruence $\sim_{\mathbf{h}}$ defined by $(\mathcal{H}, \mathcal{R}_{\mathcal{H}})$ instead of being given access to the congruence for $\mathbf{f}$.

[1]The existence of β is guaranteed by proposition 1

Algorithm 4 Robust output function learning algorithm.

Require: $(\mathcal{H}, \mathcal{R}_{\mathcal{H}})$ is a partial congruence with respect to $\mathbf{f}$.

function VERIFYCERTIFICATE($\mathbf{C}, r, e, w$)
  if $\mathcal{J}(r, e, \mathbf{C}) = w$ **then**
    return T
  else
    return F
function ROLI($\mathcal{H}, \mathcal{R}_{\mathcal{H}}$)
  if $\exists r \in \mathcal{R}$ such that $e_r \neq \perp \wedge \mathbf{f}(re_r) = \perp$ **then**
    return $\perp$
  $\forall r \in \mathcal{R}_{\mathcal{H}} : \mathbf{f}_r^{(0)}(e_r) \leftarrow \epsilon$
  $\forall r \in \mathcal{R}_{\mathcal{H}} : \mathbf{C}^{(0)}(r) \leftarrow \perp$
  while $\exists r \in \mathcal{R}_{\mathcal{H}} : \mathbf{f}_r^i \neq \mathbf{f}_r^{(i-1)}$ **do**
    for $r \in \mathcal{R}_{\mathcal{H}}$ **do**
      if $\exists \alpha \in \Sigma$: VerifyCertificate$(\mathbf{C}^{(i)}(r\alpha), r\alpha, e_{\delta(r,\alpha)}, \mathbf{f}^{(i)}_{\delta(r,\alpha)}(e_{\delta(r,\alpha)})) = \mathbf{F}$ **then**
        return $\perp$
      $\widehat{\mathbf{f}}^{(i)}(r) \leftarrow \bigsqcap_{\alpha\in\Sigma}(\mathbf{f}(r\alpha{\cdot}e_{\delta(r,\alpha)}) \nearrow \mathbf{f}^{(i-1)}_{\delta(r,\alpha)}(e_{\delta(r,\alpha)}))$
      $\sigma^{(i)}(r, \alpha) \leftarrow \widehat{\mathbf{f}}^{(i)}(r) \setminus (\mathbf{f}(r\alpha{\cdot}e_{\delta(r,\alpha)}) \nearrow \mathbf{f}^{(i-1)}_{\delta(r,\alpha)}((e_{\delta(r,\alpha)}))$
      $\mathbf{f}_r^{(i)}(e_r) \leftarrow \widehat{\mathbf{f}}^{(i)}(r) \setminus \mathbf{f}(r{\cdot}e_r)$
      if $\mathbf{f}_r^{(i)} = \mathbf{f}_r^{(i-1)}$ **then**
        $\mathbf{C}^{(i)}(r) \leftarrow \mathbf{C}^{(i-1)}(r)$
      else
        /* Set $\mathbf{C}^{(1)}(r)$ according to def 14, afterwards use the following formula */
        $\mathbf{C}^{(i)}(r) \leftarrow \mathbf{R}^{(i)}(r){\cdot}\mathbf{C}^{(i-1)}(\delta(r, \mathbf{R}^{(i)}(r)))$
  return $\sigma^{(i)}$

The ideal outcome for the algorithm when given as input a partial congruence $\sim_{\mathbf{h}}$ would be to recover the partial output function $\sigma_{\mathbf{h}}$ and subsequently reconstruct the transducer $\mathbf{h}$ defined by the partial congruence $\sim_{\mathbf{h}}$ and the partial output function $\sigma_{\mathbf{h}}$. For example, in traditional Mealy machine and total transducer learning algorithms, the intermediate hypothesis constructed are always equivalence to the transducer defined by $\sim_{\mathbf{h}}$ and $\sigma_{\mathbf{h}}$ (because the output labels are computed easily). However, as we will see, the fact that algorithm is given access to $\sim_{\mathbf{h}}$ but is querying the transducer $\mathbf{f}$, may cause a certain number of issues. In this case the robust version of the OLI algorithm is important in order to simplify the analysis.

First, notice that it is obvious that, as long as $\mathbf{f}$ and $\mathbf{h}$ are equal on all queries

performed by the OLI algorithm, then, by theorem 3 the ROLI algorithm will recover the partial output function $\sigma_{\mathbf{h}}$.

Our next task is to analyze the conditions under which the results from the queries will disagree between $\mathbf{f}$ and $\mathbf{h}$. We start with the following definition.

Definition 15. Let $(r, \alpha, r_s) \in \mathcal{R}_{\mathcal{H}} \times \Sigma \times \mathcal{R}_{\mathcal{H}}$ be a transition in the DFA incduced by $\sim_{\mathbf{h}}$. We say that the tuple (r, α, r_s) is *vulnerable* if the following conditions hold:

1. $r\alpha \sim_{\mathbf{h}} r_s$ and $r\alpha \not\sim_{\mathbf{f}} r_s$.

2. $\mathbf{f}_{r_s}(e_{r_s}) \neq \mathbf{f}_{r\alpha}(e_{r_s})$.

In simpler words, a transition from a state r with a symbol α is vulnerable if it is directed to an incorrect state (i.e. the state accessed by $r\alpha$ and r_s are different in the transducer $\mathbf{f}$) and moreover, $f_{r_s}(e_{r_s})$ is distinguishing for $r\alpha$ and r_s.

As we will show now, if a discrepancy occurs in the results of the output queries performed by the OLI algorithm when querying $\mathbf{f}$ versus querying $\mathbf{h}$, then some transition (r, α, r_s) is vulnerable.

Proposition 7. *Consider the set of strings S which are queried by $\mathtt{OLI}(\mathcal{H}, \mathcal{R}_{\mathcal{H}})$. Assume that there exists $t \in S$ such that $\mathbf{f}(t) \neq \mathbf{h}(t)$. Then, there exists $(r, \alpha, r_s) \in \mathcal{R}_{\mathcal{H}} \times \Sigma \times \mathcal{R}_{\mathcal{H}}$ such that (r, α, r_s) is vulnerable.*

In the proof of proposition 7 with the following lemma:

Lemma 9. *For a deterministic transducer $\mathbf{f} = (\sim_{\mathbf{f}}, \sigma_{\mathbf{f}})$ and $u \in \Sigma^*$, the transduction of $\mathbf{f}$ on u can be written as:*

$$\mathbf{f}(u) = \bigotimes_{i \in [1..|u|]} \sigma(u[..i-1], u[i]) \tag{6.11}$$

We now proceed with the main proof.

Proof. Every $t \in \Sigma^*$ queried by the OLI algorithm can be written as $t = r\alpha \cdot e_{\delta(r,\alpha)}$ for some $r \in \mathcal{R}_{\mathcal{H}}, \alpha \in \Sigma$ and an enabling suffix $e_{\delta(r,\alpha)}$. Moreover, by using lemma 9 we

have that

$$\begin{aligned} \mathbf{f}(t) &\neq \mathbf{h}(t) \\ \implies \bigotimes_{i\in[1..|t|]} \sigma_{\mathbf{f}}(t[..i-1], t[i]) &\neq \bigotimes_{i\in[1..|t|]} \sigma_{\mathbf{h}}(t[..i-1], t[i]) \\ \implies \bigotimes_{i\in[1..|r|]} \sigma_{\mathbf{f}}(r[..i-1], r[i])\cdot\sigma_{\mathbf{f}}(r,\alpha)\cdot\mathbf{f}_{r\alpha}(e_{\delta(r,\alpha)}) &\neq \bigotimes_{i\in[1..|r|]} \sigma_{\mathbf{h}}(r[..i-1], r[i])\cdot\sigma_{\mathbf{h}}(r,\alpha)\cdot\mathbf{h}_{r\alpha}(e_{\delta(r,\alpha)}) \end{aligned}$$

Notice now that, by definition, we have that $\mathcal{R}_{\mathcal{H}}$ is a prefix closed set and therefore, for each $u \preccurlyeq r$ we have that $u \in \mathcal{R}_{\mathcal{H}}$. By applying proposition 6 we get that

$$\bigotimes_{i\in[1..|r|]} \sigma_{\mathbf{f}}(r[..i-1], r[i])\cdot\sigma_{\mathbf{f}}(r,\alpha) = \bigotimes_{i\in[1..|r|]} \sigma_{\mathbf{h}}(r[..i-1], r[i])\cdot\sigma_{\mathbf{h}}(r,\alpha) \quad (6.12)$$

Using the above equation we simplify the overall expression as follows:

$$\begin{aligned} \bigotimes_{i\in[1..|r|]} \sigma_{\mathbf{f}}(r[..i-1], r[i])\cdot\sigma_{\mathbf{f}}(r,\alpha)\cdot\mathbf{f}_{r\alpha}(e_{\delta(r,\alpha)}) &\neq \bigotimes_{i\in[1..|r|]} \sigma_{\mathbf{h}}(r[..i-1], r[i])\cdot\sigma_{\mathbf{h}}(r,\alpha)\cdot\mathbf{h}_{r\alpha}(e_{\delta(r,\alpha)}) \\ \implies \mathbf{f}_{r\alpha}(e_{\delta(r,\alpha)}) &\neq \mathbf{h}_{r\alpha}(e_{\delta(r,\alpha)}) \end{aligned}$$

The result now follows from the fact that the set of enabling suffixes E is suffix-closed. □

We will now derive the main result of this section which constrains the errors that may occur in the output function when inferred by the ROLI algorithm.

Theorem 5 (ROLI robustness)**.** *Assume the ROLI algorithm is executed on a partial congruence and completes without failing and let* $\sigma_{ROLI} = ROLI(\mathcal{H}, \mathcal{R}_{\mathcal{H}})$*. Then, we have that:*

$$\sigma_{ROLI}(r,\alpha) = \begin{cases} \sigma_{\mathbf{h}}(r,\alpha)\cdot(\mathbf{f}_{r_s}(e_{r_s}) \nearrow \mathbf{f}_{r\alpha}(e_{r_s})), & \text{if } (r,a,r_s) \text{ is vulnerable ,} \\ \sigma_{\mathbf{h}}(r,\alpha), & \text{otherwise.} \end{cases} \quad (6.13)$$

Proof. Firstly, notice that, by proposition 7 unless some transition (r,α,r_s) is vul-

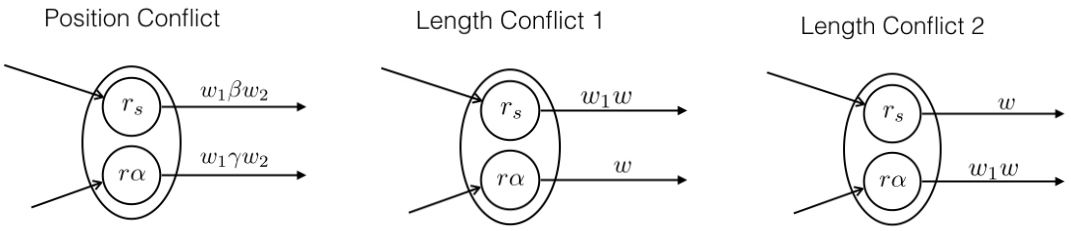

Figure 6-3: The three different types of conflict that may occur on a vulnerable transition (r, α, r_s) as analyzed in the proof of theorem 5. The labels in the outgoing transitions show the output produced by $\mathbf{f}_{r_s}(e_{r_s})$ and $\mathbf{f}_{r\alpha(e_{r_s})}$ respectively.

nerable we have that $\sigma_{\texttt{ROLI}} = \sigma_{\mathbf{h}}$. Now, consider a vulnerable transition (r, α, r_s) such that $r\alpha \sim_{\mathbf{h}} r_s$ but $r\alpha \not\sim_{\mathbf{f}} r_s$.

Let us now consider the process of running the `ROLI` algorithm in the presence of vulnerable transitions. The main issue with vulnerable transitions is that, while for the state r_s the value $\mathbf{f}^{(i)}(e_r)$ will be approximated correctly, when this value is used during the computation of $\widehat{\mathbf{f}}^{(i)}(r)$ instead of utilizing the correct value $\mathbf{f}^{(i)}_{r\alpha}(e_{\delta(r,\alpha)})$, the value $\mathbf{f}^{(i)}_{s}(e_{r_s})$ will be used. In most cases this will have the effect of aborting the algorithm but, as we will prove now in one case the algorithm will continue and end up with an incorrect value for the output label of the vulnerable transition.

We distinguish three cases on the way $w_s = \mathbf{f}_{r_s}(e_{r_s})$ differs from $w_{r\alpha} = \mathbf{f}_{r\alpha}(e_{r_s})$. Moreover, notice that even the certificate suffix will likely have a different corresponding output because we use a certificate computed with respect to r_s however, in reality, the certificate suffix is taken from $\delta(r, \alpha)$. Now, we will discuss each different case separately. The three cases are presented visually in figure 6-3.

1. **Position conflict:** This is the case demonstrated in the left side of figure 6-3. In this case, we have that there exists an index j such that $w_s[|w_s| - j] \neq w_{r\alpha}[|w_{r\alpha}| - j]$. It's easy to notice that in this case the certificate verification operation in the `ROLI` algorithm will fail.

2. **Length Conflict 1($|w_s| > |w_{r\alpha}|$):** This is the case demonstrated in the middle of figure 6-3. Notice that in this case, the length of w_s is larger than the length of $w_{r\alpha}$. Therefore, by proposition 3, regardless of the value the output of the certificate $\mathbf{C}^{(i)}(r_s)$ is taking when we prepend $r\alpha$, we have that

$|\mathcal{J}(r_s, e_{r_s}, \mathbf{C}^{(i)}(r_s))| > |\mathcal{J}(r\alpha, e_{r_s}, \mathbf{C}^{(i)}(r_s))|$ and therefore, the certificate verification will fail.

3. **Length Conflict 2($|w_s| < |w_{r\alpha}|$):** This is the case demonstrated on the right side of figure 6-3. The difference with the previous case is that the output w_s has smaller length that $w_{r\alpha}$. In this case, by setting the output produced by $\mathbf{C}(r_s)$ when computed from $r\alpha$ in an appropriate value we can ensure that the certificate verification will be successful. In this case the prefix of $w_1 = w_{r\alpha} \nslash w_s$ will be pushed into the suffix of the output label $\sigma(r, \alpha)$.

Therefore, in all cases we have that either the algorithm will fail through a certificate verification error, or the prefix of $\mathbf{f}_{r\alpha}(e_{r_s})$ will be pushed into the suffix of $\sigma(r, \alpha)$ and the proof is complete. □

The astute reader might notice that $\mathbf{f}_{r\alpha}(e_{r_s}) \nslash \mathbf{f}_{r_s}(e_{r_s})$ is undefined if $\mathbf{f}_{r_s}(e_{r_s}) \not\succcurlyeq \mathbf{f}_{r\alpha}(e_{r_s})$. Indeed, in most cases of a vulnerable transition, the `ROLI` algorithm will be able to detect the error using the certificate validation mechanism and abort the execution. In fact, there is only one case of a length conflict between $\mathbf{f}_{r_s}(e_{r_s})$ and $\mathbf{f}_{r\alpha}(e_{r_s})$ that will cause an invalid output label to be produced by `ROLI`.

Theorem 5 demonstrates the importance of the adding the certificate validation in the basic version of the `OLI` algorithm. An immediate corollary of theorem 5 is that the only case in which a transition has an invalid label with respect to the partial transducer $\mathbf{h}$ is if some prefix of a subsequent transition is pushed backwards. Let μ be the transducer defined by the partial congruence $\sim_{\mathbf{h}}$ and $\sigma_{\texttt{ROLI}}$. Then, for every state $r \in \mathcal{H}$ and suffix w we conclude that $\mathbf{f}_r(w) \succcurlyeq \mu_r(w)$. This is an important property which we will exploit in our learning algorithm in the next section. Finally, note that this property does not hold for the basic version of the `OLI` algorithm.

6.4 Learning Partial Transducers

Now that we have described and analyzed our main technical tool, the `ROLI` algorithm, we are ready to describe the algorithm for learning deterministic transducers.

6.4.1 High-Level Overview

The algorithm starts with an initial, minimal partial congruence containing only the domain distinguishing predicate $\mathcal{P}_{\epsilon}^{\mathtt{dom}}$ which distinguishes between final and non-final states. In order to implement such a congruence we use the classification tree [53] a data structure commonly used in L^*-style algorithms.

Afterwards, the `OLI` algorithm is used to derive the output function for the congruence relation. Notice here that in this first call to the `OLI` algorithm, no robustness checks are required, since the problem of inferring the output function is trivial when we only have a single state in our transducer model.

Once the first model is created and checked for equivalence we proceed to process the counterexample in order to generate a new distinguishing predicate and access string which are used to extend the current partial congruence (see below). Once we extend our current congruence with the new distinguishing predicate and access string we invoke the `ROLI` algorithm to convert the induced DFA into a transducer model.

On each failure of the execution of the `ROLI` algorithm we extract a new distinguishing predicate and an access string for a new state which extends the partial congruence. Once, the `ROLI` algorithm succeeds we generate a new transducer model and repeat this process until a correct model is constructed. We will now describe each component of the learning algorithm in more detail.

6.4.2 Counterexample Processing

Once the classification tree is constructed, we have access to the induced DFA corresponding to a partial congruence and we proceed to invoke the `ROLI` algorithm and, if the `ROLI` algorithm terminates without an error, we produce a transducer model **h** and submit it for equivalence checking. In this section, we will describe how the learning algorithm handles counterexample resulting either from equivalence queries or from failures in the `ROLI` algorithm.

Domain Counterexamples. The simplest type of counterexamples that may

occur during the execution of the algorithm are *domain counterexamples*. In this type of counterexamples, we have a string s such that $s \in \boldsymbol{dom}(\mathbf{h})$ but $s \notin \boldsymbol{dom}(\mathbf{f})$. These counterexamples may occur either as a result of an equivalence query or during the execution of the `ROLI` algorithm. Notice that domain counterexamples are independent of the output labels in the transducer model and therefore, they only dependent on the induced DFA of the current classification tree. Therefore, we process such counterexamples using the same algorithms used for processing counterexamples in classic automata learning algorithms [75]. Once we run the counterexample processing algorithm we obtain a new access string and a domain distinguishing predicate which are then used in order to extend the classification tree.

`ROLI` **Counterexamples.** By `ROLI` Counterexamples we refer to counterexamples that occur during the execution of the `ROLI` algorithm when the algorithm fails (returns $\perp$). The two cases where this may happen is during the initial queries performed by the algorithm if for some state $r \in \mathcal{R}$ such that $e_r \neq \perp$ we have that $\mathbf{f}(re_r) = \perp$. In this case, we proceed to handle the input string $s = re_r$ as a domain counterexample and generate a new domain distinguishing predicate as described above.

The second case where the `ROLI` algorithm may return $\perp$, is if some certificate verification check fails. We will now proceed to analyze how to extract distinguishing predicates from a certificate verification failure in the `ROLI` algorithm. As stated in theorem 5 the only way under which the `ROLI` algorithm may fail is if we have a vulnerable transition (r, α, r_s) such that $r\alpha \sim_{\mathbf{h}} r_s$ but $r\alpha \not\sim_{\mathbf{f}} r_s$ and moreover, e_s is producing a different output from $r\alpha$ and r_s. Therefore, if the certificate verification is failing for such a pair, then we can conclude that $\mathcal{P}^{\mathtt{out}}_{e_{r_s}, \mathbf{C}^{(i)}(r_s), \mathbf{f}^{(j)}_{r_s}(e_{r_s})}$ is a valid output distinguishing predicate for $r\alpha$ and r_s, where j is the last iteration of the algorithm before failing. We proceed to extend the classification tree splitting the access string r_s with the new access string $r\alpha$ and the output distinguishing predicate $\mathcal{P}^{\mathtt{out}}_{e_{r_s}, \mathbf{C}^{(i)}(r_s), \mathbf{f}^{(j)}_{r_s}(e_{r_s})}$.

Equivalence query counterexamples. The counterexample processing routine, shown in algorithm 6, is responsible with taking a counterexample provided by the equivalence oracle and returning a new distinguishing predicate, either domain

or output. In the case the counterexample is a domain counterexample, we proceed to handle it as described above. If we have an output counterexample t (i.e. for our transducer $\mathbf{h}$ we have $\mathbf{h}(t) \neq \mathbf{f}(t)$ and both $\mathbf{h}(t)$ and $\mathbf{f}(t)$ are not $\perp$), then we proceed with the following algorithm:

In a high level, the algorithm processes each prefix $t[..i]$ of the counterexample and first, the prefix is executed in the model in order to obtain the state r_i accessed by $t[..i]$. Afterwards, the following checks are performed:

1. The first check, is to verify that the output produced by our model $\mathbf{h}$ up to the prefix $t[..i]$ is a prefix of the output produced by the target function $\mathbf{f}$. If we have that $\mathbf{h}(t[..i]) \not\preceq \mathbf{f}(t)$ then, we conclude that in fact the value of the output function for the input $\sigma_{\mathbf{h}}(r_{i-1}, t[i])$ is *overapproximated*. In this case, notice that using $t[i+1..]$ as a certificate will be able to distinguish between r_i and $r_{i-1}t[i]$. Let $w = \mathcal{J}(r_i, e_{r_i}, t[i+1..])$. Then, we create the distinguishing predicate $\mathcal{P}^{\mathtt{out}}_{e_{r_i}, t[i+1..], w}$ and extend the classification tree by splitting r_i with the new access string $r_{i-1}t[i]$.

   Connecting back to theorem 5, this check is responsible of processing counterexamples caused due to vulnerable transitions causing a prefix of an output label to be pushed backwards into the suffix of the output label of a vulnerable transition.

2. Once the first check is passed, we proceed to extract and compare the suffix of the counterexample using a certificate. As a certificate we choose appropriately a sequence from the set $W = \{e_{r_i}, \mathbf{C}(r_i)\}$. Notice that, because the output produced by the strings in the set W, (i.e. the values $\mathcal{J}(r_i, e_{r_i}, \mathbf{C}(r_i))$ and $\mathcal{J}(r_i, \mathbf{C}(r_i), e_{r_i})$) share no common prefix and are non-empty, it follows that one of them will be appropriate to extract any suffix using the $\mathcal{J}$ function. On the case that $\mathbf{C}(r_i) = \perp$ then we can show that e_{r_i} can be used to extract any suffix using the $\mathcal{J}$ function. Assume that $s \in W$ is the certificate selected. Then, we verify that $\mathcal{J}(r_i, t[i+1..], s) = \mathcal{J}(r_{i-1}t[i], t[i+1..], s)$. If this check fails, then we generate the output distinguishing predicate $\mathcal{P}^{\mathtt{out}}_{t[i+1..], s, \mathcal{J}(r_i, t[i+1..], s)}$.

6.4.3 Overall Algorithm

Summarizing the previous section, we now provide the overall learning algorithm for deterministic transducers:

1. Initialize a classification tree $T = (V, L, E)$ with the distinguishing predicate $\mathcal{P}_{\epsilon}^{\texttt{dom}}$ and a single access string ϵ and use queries to set the leaf node to either a **T**-child or a **F**-child.

2. Use the `OLI` algorithm to obtain a transducer model.

3. Repeat the following steps until an equivalence query returns **T**:

   (a) Make an equivalence query on the current transducer model and process any counterexample returned as described in section 6.4.2.

   (b) Once the classification tree is extended, call the `ROLI` algorithm using the DFA induced by the tree.

   (c) While the `ROLI` algorithm returns $\perp$, process the corresponding counterexamples, extend the classification tree and call the `ROLI` algorithm until a transducer model is obtained.

4. return the current transducer model.

Algorithm 5 presents the pseudocode of the deterministic learning algorithm.

6.4.4 Correctness and Complexity

We will now state the following theorem which summarizes the query and time complexity of our algorithm.

Theorem 6 (Learnability of Deterministic Transducers)**.** *Let* $\mathbf{f}$ *be a function representable as a deterministic transducer with* n *states. Then,* $\mathbf{f}$ *is learnable using* $O(n^3|\Sigma| + nm)$ *output and* n *equivalence queries, where* m *is the length of longest counterexample given to the algorithm.*

Algorithm 5 Learning algorithm for deterministic transducers

Require: $\mathcal{O}, \mathcal{E}$ is an output and equivalence oracle for a function $\mathbf{f}$

function LEARNDET($\mathcal{O}, \mathcal{E}$)
- $(\mathcal{H}, \mathcal{R}_{\mathcal{H}}) \leftarrow (\phi, \epsilon)$
- $\sigma_h \leftarrow \texttt{OLI}(\mathcal{H}, \mathcal{R}_{\mathcal{H}})$
- **while** $\mathcal{E}(\mathcal{H}, \mathcal{R}_{\mathcal{H}}, \sigma_h) \neq \top$ **do**
  - $t \leftarrow \texttt{getCounterexample()}$
  - $(\phi_n, r_n) \leftarrow \texttt{processCounterexample}(t)$
  - $(\mathcal{H}, \mathcal{R}_{\mathcal{H}}) \leftarrow (\mathcal{H} \cup \{\phi_n\}, \mathcal{R}_{\mathcal{H}} \cup \{r_n\})$
  - **while** $(\sigma_h = \texttt{ROLI}(\mathcal{H}, \mathcal{R}_{\mathcal{H}})) = \bot$ **do**
    - $(\phi_n, r_n) \leftarrow \texttt{getROLIDistinguishingPredicate()}$
    - $(\mathcal{H}, \mathcal{R}_{\mathcal{H}}) \leftarrow (\mathcal{H} \cup \{\phi_n\}, \mathcal{R}_{\mathcal{H}} \cup \{r_n\})$
- **return** $(\mathcal{H}, \mathcal{R}_{\mathcal{H}}, \sigma_h)$

Algorithm 6 Counterexample processing algorithm

Require: $\mathcal{O}, \mathcal{E}$ is an output and equivalence oracle for a function $\mathbf{f}$

function PROCESSCOUNTEREXAMPLE($t, \mathbf{h}$)
- $r_0 \leftarrow \epsilon$
- **for** $i = 1; i \leq |t|$ **do**
  - **if** $\widehat{\mathbf{h}}(t[..i]) \not\preccurlyeq \mathbf{f}(t)$ **then**
    - $w \leftarrow \mathcal{J}(r_i, e_{r_i}, t[i+1..])$
    - **return** $(\mathcal{P}^{\texttt{out}}_{e_{r_i}, t[i+1..], w}, r_{i-1}t[i])$
  - $r_i \leftarrow \texttt{getAccessString}(t[..i])$
  - $W \leftarrow \{e_{r_i}, \mathbf{C}(r_i)\}$
  - $s \leftarrow \texttt{getCertificate}(W, \mathbf{h}_{r_i}(e_{r_i}))$
  - **if** $\mathcal{J}(r_i, t[i+1..], s) \neq \mathcal{J}(r_{i-1}t[i], t[i+1..], s)$ **then**
    - $w \leftarrow \mathcal{J}(r_i, t[i+1..], s)$
    - **return** $(\mathcal{P}^{\texttt{out}}_{t[i+1..], s, w}, r_{i-1}t[i])$
- // Unreachable code

In terms of time complexity, assuming that the maximum length of an output label is k and each output and equivalence query take constant time, the algorithm will run time $O(n^2(n|\Sigma|k + m))$. The time complexity stems from the fact that building each DFA model requires $|\Sigma|n$ calls to the `sift` procedure where the maximum height of the tree is n. Adding the complexity of counterexample processing and the `ROLI` algorithm we obtain the result.

Chapter 7

Learning Non-Deterministic Transducers

We introduce a subclass class of nonedeterministic transducers that have a canonical form, based on an indexed form of syntactic congruence. The main idea behind this class is the following. Given any state with two or more non deterministic transitions, the output produced by each nondeterministic transition up to the enabling suffix starts with a different prefix. The intuition is that, at this point, we are forced to break into a non-determinsitic choice because we can no longer keep producing output before resolving the lookahead.

The way we capture the different lookaheads needed to resolve the different cases is by detecting *nonmonotonicity* in the behavior of the given function $\mathbf{f} : \Sigma^* \to \Gamma^*_\perp$ and by separating monotonic and nonmonotonic continuations using nondeterminism.

7.0.1 Visible nondeterminism

We define a partition of a language $L \subseteq \Sigma^*$ into two lookahead languages $\widetilde{L}$ and $L \backslash \widetilde{L}$ representing *nonmonotone* and *monotone* behaviors wrt $\mathbf{f}$. Observe that all definitions here have $\mathbf{f}$ as an implicit parameter. For $u \in \Sigma^*$ we let $\boldsymbol{con}(u)$ denote the set of all $v \in \Sigma^*$ such that $u \cdot v \in \boldsymbol{dom}(\mathbf{f})$. Let the *enabling sequences from* u be

the following subset of $\boldsymbol{con}(u)$

$$\boldsymbol{es}(u) \stackrel{\text{def}}{=} \{v \in \boldsymbol{con}(u) \mid \forall x(\epsilon \prec x \prec v \Rightarrow x \notin \boldsymbol{con}(u))\}$$

So, an enabling sequence from u is a valid proper continuation from u that is minimal in the sense that no proper prefix of it suffices as a valid continuation. Observe that $\boldsymbol{con}(u)$ is nonempty iff $\boldsymbol{es}(u)$ is nonempty. For $e \in \boldsymbol{es}(u)$ let $\boldsymbol{nme}(u, e)$ be the set of *non-monotonic extensions of e from u*.

$$\begin{aligned}
\boldsymbol{nme}(u,e) \stackrel{\text{def}}{=} \{w \in \boldsymbol{con}(u) \mid e \prec w \wedge \mathbf{f}(u{\cdot}e) \not\preccurlyeq \mathbf{f}(u{\cdot}w) \wedge & \quad (7.1)\\
\forall x(x \in \boldsymbol{con}(u) \wedge e \prec x \prec w \Rightarrow \mathbf{f}(u{\cdot}e) \preccurlyeq \mathbf{f}(u{\cdot}x))\} & \quad (7.2)\\
\boldsymbol{nme}(u) \stackrel{\text{def}}{=} \bigcup_{e \in \boldsymbol{es}(u)} \boldsymbol{nme}(u,e) & \quad (7.3)
\end{aligned}$$

The intuition is that a non-monotonic extension of an enabling sequence breaks monotonicity locally and therefore requires a nondeterministic choice to be made in the underlying transducer. Condition (7.1) ensures that w is a witness of *nonmonotonicity* while condition (7.2) makes sure that the nonmonotonicity is *local* in the sense that the behavior has been monotonic upto *butlast*(w).

Finally, provided that $\boldsymbol{nme}(u)$ is nonempty we define a partition of a continuation language L that is associated with u. Observe that L here is assumed to be a nonempty subset of $\boldsymbol{con}(u)$.

$$\widetilde{L}^u \stackrel{\text{def}}{=} \boldsymbol{nme}(u){\cdot}\Sigma^* \cap L$$

We write $\widetilde{L}$ for $\widetilde{L}^u$ when u is clear from the context.

7.0.2 Indexed congruence

Let $\widehat{\mathbf{f}}_L(u)$ be the output prefix of $\mathbf{f}$ that depends only on input prefix u for input suffixes from L.

$$\widehat{\mathbf{f}}_L(u) \stackrel{\text{def}}{=} \bigsqcap_{w \in L} \mathbf{f}(u{\cdot}w)$$

Define $\mathbf{f}_u^L : \Sigma^* \mapsto \Gamma_\perp^*$ as the continuation function of $\mathbf{f}$ after input u that omits from the output the common output prefix produced for u, wrt continuations from L:

$$\mathbf{f}_u^L(w) \stackrel{\text{def}}{=} \widehat{\mathbf{f}}_L(u) \setminus \mathbf{f}(u{\cdot}w)$$

We define the indexed congruence relation $\sim_L$ over Σ^* as follows:

$$u \sim_L v \stackrel{\text{def}}{\iff} \forall w \in L : \mathbf{f}_u^L(w) = \mathbf{f}_v^L(w) \tag{7.4}$$

The relations $\sim_L$ induce the following equivalence relation over $\Sigma^* \times 2^{\Sigma^*}$:

$$(u, L) \equiv (v, L') \stackrel{\text{def}}{\iff} L = L' \wedge u \sim_L v$$

A $\equiv$-equivalence class is denoted by $\langle u, L\rangle_{/\equiv}$ or $\langle u, L\rangle$ when $\equiv$ is clear from the context. We write $\widehat{\mathbf{f}}$ for $\widehat{\mathbf{f}}_{\Sigma^*}$ and $\mathbf{f}_u$ for $\mathbf{f}_u^{\Sigma^*}$. We write $\sim$ for $\sim_{\Sigma^*}$.

7.0.3 Visibly nondeterministic transducer

The *visibly nondeterministic transducer* of $\mathbf{f}$, denoted $\boldsymbol{VND}(\mathbf{f})$, is defined as the least fixpoint of $(\Sigma, \Gamma, Q, q_0, F, \Delta, \lambda)$ satisfying the following conditions.

- $q_0 = \langle \epsilon, \boldsymbol{dom}(\mathbf{f})\rangle$ and $q_0 \in Q$;
- if $\langle u, L\rangle \in Q$ and $E \in \{\widetilde{L}^u, L\backslash\widetilde{L}^u\}$, then, for all $x \in \Sigma$, if $x'E \neq \emptyset$ then $\langle u{\cdot}x, x'E\rangle \in Q$ and

$$\langle u, L\rangle \xrightarrow{x/\widehat{\mathbf{f}}_L(u)\setminus\widehat{\mathbf{f}}_{x'E}(u\cdot x)} \langle u{\cdot}x, x'E\rangle \in \Delta;$$

- if $\langle u, L\rangle \in Q$ and $\epsilon \in L$ then $\langle u, L\rangle \in F$;
- $\lambda = \widehat{\mathbf{f}}_{\boldsymbol{dom}(\mathbf{f})}(\epsilon)$.

We have the following main correctness result for $\boldsymbol{VND}(\mathbf{f})$. (Proof is in the appendix.)

Theorem 7. $\mathcal{T}_{\boldsymbol{VND}(\mathbf{f})} = \mathbf{f}$.

Proof. First, we prove (7.5).

$$\forall u, v \in \Sigma^*, x \in \Sigma, L \subseteq \Sigma^*, E \subseteq L : (u \sim_L v \Rightarrow (\widehat{\mathbf{f}}_L(u) \setminus \widehat{\mathbf{f}}_{x'E}(u{\cdot}x) = \widehat{\mathbf{f}}_L(v) \setminus \widehat{\mathbf{f}}_{x'E}(v{\cdot}x))) \tag{7.5}$$

This implies that for all states q of $\boldsymbol{VND}(\mathbf{f})$, and all outgoing transitions from q the output produced is invariant wrt $\equiv$ and therefore the transitions of $\boldsymbol{VND}(\mathbf{f})$ are well-defined. Consider fixed $u, v \in \Sigma^*, x \in \Sigma, L \subseteq \Sigma^*, E \subseteq L$ such that $u \sim_L v$. We prove that $\widehat{\mathbf{f}}_L(u) \setminus \widehat{\mathbf{f}}_{x'E}(u{\cdot}x) = \widehat{\mathbf{f}}_L(v) \setminus \widehat{\mathbf{f}}_{x'E}(v{\cdot}x)$ through a series of equivalence preserving transformations of true statements. First, the following statement holds by definition of $\sim_L$, for all $x{\cdot}w \in L$, and therefore also for all $x{\cdot}w \in E$ (or $w \in x'E$)

$$\widehat{\mathbf{f}}_L(u) \setminus \mathbf{f}(u{\cdot}x{\cdot}w) = \widehat{\mathbf{f}}_L(v) \setminus \mathbf{f}(v{\cdot}x{\cdot}w)$$

This implies in particular that the following statement holds

$$\prod_{w \in x'E} \widehat{\mathbf{f}}_L(u) \setminus \mathbf{f}(u{\cdot}x{\cdot}w) = \prod_{w \in x'E} \widehat{\mathbf{f}}_L(v) \setminus \mathbf{f}(v{\cdot}x{\cdot}w)$$

We can reorder the operations because the first parts are fixed

$$\widehat{\mathbf{f}}_L(u) \setminus \prod_{w \in x'E} \mathbf{f}(u{\cdot}x{\cdot}w) = \widehat{\mathbf{f}}_L(v) \setminus \prod_{w \in x'E} \mathbf{f}(v{\cdot}x{\cdot}w)$$

which is, by definition of $\widehat{\mathbf{f}}_{x'E}$, the same as

$$\widehat{\mathbf{f}}_L(u) \setminus \widehat{\mathbf{f}}_{x'E}(u{\cdot}x) = \widehat{\mathbf{f}}_L(v) \setminus \widehat{\mathbf{f}}_{x'E}(v{\cdot}x)$$

which completes the proof of (7.5).

Next we prove that $\mathcal{T}_{\boldsymbol{VND}(\mathbf{f})} = \mathbf{f}$. Consider a start state $q = \langle u, L \rangle$ and a path in $\Delta_{\boldsymbol{VND}(\mathbf{f})}$ consisting of two transitions starting from q for input characters $a, b \in \Sigma$ and some states $s_1 = \langle v, L_1 \rangle$ and s_2

$$q \xrightarrow{a/\widehat{\mathbf{f}}_L(u) \setminus \widehat{\mathbf{f}}_{a'E}(ua)} s_1 \xrightarrow{b/\widehat{\mathbf{f}}_{L_1}(v) \setminus \widehat{\mathbf{f}}_{b'E_1}(vb)} s_2$$

where we know that $E \subseteq L$, $L_1 = a'E \neq \emptyset$, $E_1 \subseteq L_1$, $b'E_1 \neq \emptyset$, and $ua \sim_{L_1} v$. Let $L_2 = b'E_1$. Then, by using (7.5), it follows that $\widehat{\mathbf{f}}_{L_1}(v) \setminus \widehat{\mathbf{f}}_{L_2}(vb) = \widehat{\mathbf{f}}_{L_1}(ua) \setminus \widehat{\mathbf{f}}_{L_2}(uab)$, and therefore the above path equals

$$q \xrightarrow{a/\widehat{\mathbf{f}}_L(u)\setminus\widehat{\mathbf{f}}_{L_1}(ua)} s_1 \xrightarrow{b/\widehat{\mathbf{f}}_{L_1}(ua)\setminus\widehat{\mathbf{f}}_{L_2}(uab)} s_2$$

from this follows, by generalizing to arbitrary finite paths, given $u \in L_0 = \boldsymbol{dom}(\mathbf{f})$, $|u| = k$, that there exists L_i for $1 \leq i \leq k$ such that

$$\begin{aligned}\langle \epsilon, L_0\rangle \xrightarrow{u[1]/\widehat{\mathbf{f}}_{L_0}(\epsilon)\setminus\widehat{\mathbf{f}}_{L_1}(u[..1])} &\langle u[..1], L_1\rangle \xrightarrow{u[2]/\widehat{\mathbf{f}}_{L_1}(u[..1])\setminus\widehat{\mathbf{f}}_{L_2}(u[..2])} \\ &\langle u[..2], L_2\rangle \xrightarrow{u[3]/\widehat{\mathbf{f}}_{L_2}(u[..2])\setminus\widehat{\mathbf{f}}_{L_3}(u[..3])} \\ &\langle u[..3], L_3\rangle \cdots \langle u, L_k\rangle\end{aligned}$$

where $\epsilon \in L_k$ because $u \in \boldsymbol{dom}(\mathbf{f})$, so $\langle u, L_k\rangle \in F_{\boldsymbol{VND}(\mathbf{f})}$. Then, by using the definition of $\Delta^*_{\boldsymbol{VND}(\mathbf{f})}$ and the definition of $\mathcal{T}_{\boldsymbol{VND}(\mathbf{f})}$, it follows that

$$(u, \widehat{\mathbf{f}}_{L_0}(\epsilon)\cdot\bigotimes_{i=1}^{k}(\widehat{\mathbf{f}}_{L_{i-1}}(u[..i{-}1]) \setminus \widehat{\mathbf{f}}_{L_i}(u[..i]))) \in \mathcal{T}_{\boldsymbol{VND}(\mathbf{f})}$$

We can now apply ($k-1$ times) the simplification that $(x \setminus y)\cdot(y \setminus z) = (x \setminus z)$ when $x \preccurlyeq y$ and $y \preccurlyeq z$, because $\widehat{\mathbf{f}}_L(v) \preccurlyeq \widehat{\mathbf{f}}_{a'E}(va)$ when $E \subseteq L$ and $a'E \neq \emptyset$. It follows that

$$\widehat{\mathbf{f}}_{L_0}(\epsilon)\cdot\bigotimes_{i=1}^{k}(\widehat{\mathbf{f}}_{L_{i-1}}(u[..i{-}1]) \setminus \widehat{\mathbf{f}}_{L_i}(u[..i])) = \widehat{\mathbf{f}}_{L_0}(\epsilon)\cdot(\widehat{\mathbf{f}}_{L_0}(\epsilon) \setminus \widehat{\mathbf{f}}_{L_k}(u)) = \widehat{\mathbf{f}}_{L_k}(u)$$

where the last equality holds because if $x \preccurlyeq y$ then $x\cdot(x \setminus y) = y$.

Let $L = \widetilde{L_{k-1}}^{u[..k-1]}$ be the nonmonotonic lookahead from state $\langle u[..k{-}1], L_{k-1}\rangle$, i.e., this is the next to last state. Since $\epsilon \in L_k$ we know that L_k cannot be $u[k]'L$ because all strings in L have length at least 2. This means that $L_k = u[k]'(L_{k-1}\backslash L)$ which implies that L_k does not contain any nonmonotonic extensions of u. In other words, for all $v \in L_k$, $\mathbf{f}(u) \preccurlyeq \mathbf{f}(uv)$ and so $\widehat{\mathbf{f}}_{L_k}(u) = \mathbf{f}(u)$.

Finally, it follows that $(u, \mathbf{f}(u)) \in \mathcal{T}_{\boldsymbol{VND}(\mathbf{f})}$ and if $u' = u$ then $\mathbf{f}(u) = \mathbf{f}(u')$, so $\boldsymbol{VND}(\mathbf{f})$ is functional. The theorem follows. □

Example 3. The example illustrates the principle behind how *finalizers* from the *subsequential* case are handled by visible nondeterminism. Consider the following functon $\mathbf{f}$ over ASCII strings:

$$\mathbf{f}(\epsilon) = \epsilon, \quad \mathbf{f}(\texttt{\&}) = \texttt{\&}, \quad \mathbf{f}(\texttt{\&a}) = \texttt{\&a}, \quad \mathbf{f}(\texttt{\&am}) = \texttt{\&am}, \quad \mathbf{f}(\texttt{\&amp}) = \texttt{\&amp}, \quad \mathbf{f}(\texttt{\&}) = \texttt{\&},$$

This function represents a small part of an HTML decoder. Assume also that for all other input strings the output is undefined. We illustrate how $\boldsymbol{VND}(\mathbf{f})$ is formed in this case. We have that, $\boldsymbol{dom}(\mathbf{f}) = L_0 = \{\epsilon, \texttt{\&}, \texttt{\&a}, \texttt{\&am}, \texttt{\&amp}, \texttt{\&}\}$, $\boldsymbol{es}(\epsilon) = \{\texttt{\&}\}, \boldsymbol{nme}(\epsilon, \texttt{\&}) = \boldsymbol{nme}(\epsilon) = \emptyset, \widetilde{L_0} = \emptyset$. Let $L_1 = \texttt{\&}'L_0 = \{\epsilon, \texttt{a}, \texttt{am}, \texttt{amp}, \texttt{amp;}\}$. We have the transition

$$\langle \epsilon, L_0 \rangle \xrightarrow{\texttt{\&}/\texttt{\&}} \langle \texttt{\&}, L_1 \rangle$$

and $\boldsymbol{es}(\texttt{\&}) = \{\texttt{a}\}, \boldsymbol{nme}(\texttt{\&}, \texttt{a}) = \boldsymbol{nme}(\texttt{\&}) = \{\texttt{amp;}\}, \widetilde{L_1} = \{\texttt{amp;}\}, L_1 \backslash \widetilde{L_1} = \{\epsilon, \texttt{a}, \texttt{am}, \texttt{amp}\}$. Let $N_2 = \texttt{a}'\widetilde{L_1} = \{\texttt{mp;}\}$, $M_2 = \texttt{a}'(L_1 \backslash \widetilde{L_1}) = \{\epsilon, \texttt{m}, \texttt{mp}\}$. We now get *two* nondeterministic transitions

$$\langle \texttt{\&}, L_1 \rangle \xrightarrow{\texttt{a}/\epsilon} \langle \texttt{\&a}, N_2 \rangle$$
$$\langle \texttt{\&}, L_1 \rangle \xrightarrow{\texttt{a}/\texttt{a}} \langle \texttt{\&a}, M_2 \rangle$$

Next, we get that $\boldsymbol{es}(\texttt{\&a}) = \{\texttt{m}\}, \boldsymbol{nme}(\texttt{\&a}, \texttt{m}) = \boldsymbol{nme}(\texttt{\&a}) = \{\texttt{mp;}\}, \widetilde{N_2} = N_2, \widetilde{M_2} = \emptyset$. Let $N_3 = \texttt{m}'N_2 = \{\texttt{p;}\}$, $M_3 = \texttt{m}'M_2 = \{\epsilon, \texttt{p}\}$. We get, similarly to above, the remaining tarnsitions

$$\langle \texttt{\&a}, N_2 \rangle \xrightarrow{\texttt{m}/\epsilon} \langle \texttt{\&am}, N_3 \rangle \xrightarrow{\texttt{p}/\epsilon} \langle \texttt{\&amp}, \{\texttt{;}\} \rangle \xrightarrow{\texttt{;}/\epsilon} \langle \texttt{\&}, \{\epsilon\} \rangle$$
$$\langle \texttt{\&a}, M_2 \rangle \xrightarrow{\texttt{m}/\texttt{m}} \langle \texttt{\&am}, M_3 \rangle \xrightarrow{\texttt{p}/\texttt{p}} \langle \texttt{\&amp}, \{\epsilon\} \rangle$$

Observe that $\mathbf{f}_u^{\{\epsilon\}}(\epsilon) = \widehat{\mathbf{f}}_{\{\epsilon\}}(u) \setminus \mathbf{f}(u) = \mathbf{f}(u) \setminus \mathbf{f}(u) = \epsilon$ when $u \in \boldsymbol{dom}(\mathbf{f})$, so $u \sim_{\{\epsilon\}} v$

for all $u, v \in \boldsymbol{dom}(\mathbf{f})$. Therefore $\langle \texttt{\&amp}, \{\epsilon\}\rangle = \langle \texttt{\&}, \{\epsilon\}\rangle$ above. The transducer $\boldsymbol{VND}(\mathbf{f})$ is shown in Figure 3-1(middle). $\boxtimes$

Example 4. Let $\mathbf{f}$ be the transduction function of the transducer in Figure 3-1(right). We construct $\boldsymbol{VND}(\mathbf{f})$. Assume $\Sigma = \texttt{ASCII}$. We use regular expressions here. Consider $q_0 = \langle u, L_0 \rangle$ where $L_0 = [\![\texttt{.*}]\!]$ and $u = \epsilon$. Observe $\boldsymbol{con}(\epsilon) = \boldsymbol{dom}(\mathbf{f}) = [\![\texttt{.*}]\!]$ in this case. We have $\boldsymbol{es}(u) = \Sigma$ and we get the following non-monotonic extension for each such initial enabling sequence $e \in \boldsymbol{es}(u)$.

$$\text{for } e \neq \texttt{<}: \;\; \boldsymbol{nme}(\epsilon, e) = \emptyset, \quad \boldsymbol{nme}(\epsilon, \texttt{<}) = [\![\texttt{<[^>]*>}]\!]$$

To understand why $\boldsymbol{nme}(\epsilon, \texttt{<})$ is $[\![\texttt{<[^>]*>}]\!]$ note that $\texttt{<} \preccurlyeq \texttt{<a<a>}$ but $\mathbf{f}(\texttt{<}) \not\preccurlyeq \mathbf{f}(\texttt{<a<a>})$ while at the same time for any proper prefix x of $\texttt{<a<a>}$ it holds that $\mathbf{f}(u{\cdot}\texttt{<}) \preccurlyeq \mathbf{f}(u{\cdot}x)$. It follows that

$$\widetilde{L_0} = [\![\texttt{<[^>]*>.*}]\!], \quad L_0 \backslash \widetilde{L_0} = [\![\texttt{<[^>]*|[^<].*|()}]\!].$$

In order to compute the transitions from the initial state, we first compute the derivatives $x'L_0\backslash\widetilde{L_0}$ and $x'\widetilde{L_0}$ with respect to each symbol $x \in \Sigma$. We have the following four cases:

$$\begin{aligned}
L_1 = \texttt{<}'\widetilde{L_0} &= [\![\texttt{[^>]*>.*}]\!] \\
L_2 = \texttt{<}'L_0\backslash\widetilde{L_0} &= [\![\texttt{[^>]*}]\!] \\
\text{for } x \neq \texttt{<}: \quad x'\widetilde{L_0} &= \emptyset \\
\text{for } x \neq \texttt{<}: \quad x'L_0\backslash\widetilde{L_0} &= L_0
\end{aligned}$$

Since both L_1 and L_2 are nonempty and distinct, there are two distinct states $q_1 = \langle \texttt{<}, L_1 \rangle$ and $q_2 = \langle \texttt{<}, L_2 \rangle$ with the respective associated continuations L_1 and L_2. Also, since $x'\widetilde{L_0}$ is empty when $x \neq \texttt{<}$ we may compute the derivative wrt L_0, i.e., we get that for $x \neq \texttt{<}$, $x'L_0 = L_0$.

Transitions are computed as follows. Transition $q_0 \xrightarrow{\texttt{</}y_1} q_1$ is computed wrt lookahead $\widetilde{L_0}$, transition $q_0 \xrightarrow{\texttt{</}y_2} q_2$ is computed wrt lookahead $L_0\backslash\widetilde{L_0}$, and, for $x \neq \texttt{<}$,

transition $q_0 \xrightarrow{x/y_3} \langle x, L_0\rangle$ is computed wrt lookahead L_0, where,

$$\begin{aligned} & y_1 = (\widehat{\mathbf{f}}_{\widetilde{L_0}}(\epsilon) \setminus \widehat{\mathbf{f}}_{\texttt{<}'\widetilde{L_0}}(\epsilon\cdot\texttt{<})) = (\epsilon \setminus \widehat{\mathbf{f}}_{L_1}(\texttt{<})) = \epsilon \\ & y_2 = (\widehat{\mathbf{f}}_{L_0\setminus\widetilde{L_0}}(\epsilon) \setminus \widehat{\mathbf{f}}_{\texttt{<}'L_0\setminus\widetilde{L_0}}(\epsilon\cdot\texttt{<})) = (\epsilon \setminus \widehat{\mathbf{f}}_{L_2}(\texttt{<})) = \texttt{<} \\ \text{for } x \neq \texttt{<}: \quad & y_3 = (\widehat{\mathbf{f}}_{L_0}(\epsilon) \setminus \widehat{\mathbf{f}}_{x'L_0}(\epsilon\cdot x)) = (\epsilon \setminus \widehat{\mathbf{f}}_{L_0}(x)) = x \end{aligned}$$

where $y_1 \neq y_2$, so the nondeterministic choice from q_0 is *visible*. Next, fix $x \neq \texttt{<}$. We show that $x \sim \epsilon$ (recall that $L_0 = \Sigma^*$) and so $\langle x, L_0\rangle = q_0$. Recall that $x \sim \epsilon$ holds iff $\widehat{\mathbf{f}}(x) \setminus \mathbf{f}(x\cdot v) = \widehat{\mathbf{f}}(\epsilon) \setminus \mathbf{f}(\epsilon\cdot v)$ for any $v \in \Sigma^*$ and the latter is true because $\widehat{\mathbf{f}}(x) = x$.

States q_1 and q_2 are explored as follows. First, we calculate that $\boldsymbol{nme}(\texttt{<}) = [\![\texttt{[\^{}>]+>}]\!]$.

We consider $q_2 = \langle \texttt{<}, L_2\rangle$ first. Fix $x \neq \texttt{>}$. Since L_2 does not admit $\texttt{>}$ it follows that $\widetilde{L_2} = \emptyset$, so $L_2\setminus\widetilde{L_2} = L_2$. State q_2 ends up with loop $q_2 \xrightarrow{x/x} q_2$ because $x'L_2 = L_2$ and $\texttt{<} \sim_{L_2} \texttt{<}\cdot x$. The output on the transition is x because $(\widehat{\mathbf{f}}_{L_2}(\texttt{<}) \setminus \widehat{\mathbf{f}}_{x'L_2}(\texttt{<}\cdot x)) = (\texttt{<} \setminus \texttt{<}\cdot x) = x$. State q_2 is final because $\epsilon \in L_2$.

We consider $q_1 = \langle \texttt{<}, L_1\rangle$ next. We get that

$$\begin{aligned} \widetilde{L_1} &= \boldsymbol{nme}(\texttt{<}) \cdot \Sigma^* \cap L_1 = [\![\texttt{[\^{}>]+>.*}]\!] \cap [\![\texttt{[\^{}>]*>.*}]\!] = [\![\texttt{[\^{}>]+>.*}]\!] \\ L_1\setminus\widetilde{L_1} &= [\![\texttt{[\^{}>]*>.*}]\!] \setminus \widetilde{L_1} = [\![\texttt{>.*}]\!] \end{aligned}$$

Fix $x \in \Sigma\setminus\{\texttt{>}\}$. It is now easy to calculate that $\texttt{>}'L_1\setminus\widetilde{L_1} = L_0$ and $x'\widetilde{L_1} = L_1$ and that in all other cases the derivative is empty. We also have that $\texttt{<>} \sim_{L_0} \epsilon$ and $\texttt{<}\cdot x \sim_{L_1} \texttt{<}$, so $\langle \texttt{<>}, L_0\rangle = q_0$ and $\langle \texttt{<}\cdot x, L_1\rangle = q_1$. Also, it is straightforward to calculate that the outputs of the transitions from q_1 are empty. Thus the transitions are $q_1 \xrightarrow{x\neq\texttt{>}/\epsilon} q_1$ and $q_1 \xrightarrow{\texttt{>}/\epsilon} q_0$. Here the initial output λ is ϵ. ⊠

Now that we defined the general concept of visible non-determinism and visibly non-deterministic transducers, we will proceed to describe an extension of the algorithm for learning deterministic transducers in order to learn a subclass of the VND class. The main motivation behind this subclass is that, as we will demonstrate in our evaluation, it is able to efficiently capture many regular-expression based string

transformations while being simple enough to allow efficient learnability.

We conjecture that the whole class of VND transducers is in fact efficiently learnable and we consider extending our algorithm to be an interesting direction for future work.

7.0.4 Simple Visibly Non-Deterministic Transducers

We will now describe the additional constraints we impose on the functions in this section. Specifically, we call a function $\mathbf{f}$ to be *simple visibly non-deterministic (SVND)* if it satisfies the following properties:

1. $\mathbf{f}$ is visibly non-deterministic.
2. $\mathbf{f}$ is total.
3. Consider the transducer $\boldsymbol{VND}(\mathbf{f}) = (\Sigma, Q, q_0, F, \Delta, \lambda)$. Then, for a state (u, L) we have that $\widetilde{L}^u \neq \emptyset \implies (u, L) \in F$.

Condition (3) implies that, in the transducer formulation of the function $\mathbf{f}$, non-deterministic transitions can only happen in final states.

There are two main motivations behind the SVND class of functions: Firstly, conditions (2) and (3) greatly simplify the analysis of non-monotonic extensions since they allow us to create a simple oracle in order to check for non-monotonic extensions from a state in the transducer. Moreover, since non-determinism only emerges from final states in the target transducer, the analysis we performed for the `ROLI` algorithm can be easily reused in this case as well.

Secondly, this class is well suited for learning regular-expression based transformations such as those performed using `preg_replace` type functions [71] and which are commonly found in web applications and are a vital part of security-critical components such as input sanitization frameworks. Visible determinism is well suited for such functions, since the default behavior of the transformation is to compute the identity function unless the regular expression is matched in which case an alternative path is triggered. Moreover, such transformations are by definition total and

therefore, the class of simple visibly non-deterministic functions looks appropriate for learning models of such transformations.

Learning algorithm overview. Intuitively, our algorithm views the transducer as an ensemble of deterministic partial transducers. For each input, exactly one transducer is reaching an accepting state and the output produced by that transducer is returned as the output of the transduction. Because the transducer is visibly non-deterministic, we can use the prefixes of each computation in order to distinguish which partial transducer (or which non-deterministic transitions) were used for each input character processed by the transducer.

7.0.5 Extended Classification Tree

As in the deterministic algorithm we use the classification tree data structure in order to distinguish between different states in the target transducer. However, in this case, we extend the tree in order to hold information regarding the monotonic or non-monotonic lookaheads from each transition. More specifically we define the extended classification tree as follows:

Definition 16. An extended classification tree (ECT) is a binary tree $T = (V, L, E)$ where:

- $V \subset (\mathbb{P} \cup (\Sigma^* \times \Sigma^* \times \mathbb{B})$ is the set of internal nodes.
- $L \subset \Sigma^* \times \Sigma^* \times \mathbb{B}$ is the set of leafs.
- $E \subset V \times V \times \mathbb{B}$ is the transition relation. A transition (u, v, b) is called a **T**-child of u if $b = \mathbf{T}$ otherwise, it's called a **F**-child.

The ECT operates exactly as the original classification tree, however each state is holding additional information which determines whether it is part of a non-monotonic extension as we will describe now.

Path Restricted Output Queries The main way we utilize the additional information in the leafs of the extended classification tree is to implement a concept called *path restricted output queries*. First, define a state (u, L) in a visible transducer

to be enabled for an input v if $u \preccurlyeq v$ and moreover $(u \setminus v) \in L$. In other words, a state is enabled for an input v if some prefix of v is accessing the state and moreover, the suffix of v is part of the lookahead of the state.

A path restricted output query is given a leaf $(u, v, b) \in L$ and a string w and is computed as:

$$\mathcal{Q}_{\texttt{path}}(u, v, b) = \begin{cases} \mathbf{f}(w), & \text{if } (u, v, b) \text{ is enabled on } w \\ \perp, & \text{otherwise.} \end{cases} \tag{7.6}$$

Given a leaf (u, v, b) and a string w In order to check whether the state represented by the leaf (u, v, b) is enabled we proceed as follows:

1. Initially, we reduce the string w into a string $p \preccurlyeq w$ such that $u \preccurlyeq p$ such that $p \in \boldsymbol{nme}(u, ())$, or in the case that no such prefix exists, we set $p = w[1]$. If $u \not\preccurlyeq w$ we return $\perp$.

2. Afterwards we check if $(v \preccurlyeq \mathbf{f}(p)) = b$. If the check is passed we return $\mathbf{f}(w)$, otherwise we return $\perp$.

To reduce the string w into the prefix p we use a variation of the path reconstruction algorithm (described below) in order to recover all the non-monotonic paths and select the appropriate one. Now that we have described the path restricted output queries, we can give a better explanation on the structure of the leafs in the extended classification tree:

Because of the additional constraints (2),(3) in the definition of the SVND class we have that for any state all enabling suffixes are of length 1. Therefore, for a state u and symbol α, we can always query $\mathbf{f}(u)$ and $\mathbf{f}(u\alpha)$ and obtain the corresponding output. A tuple (u, v, b) represents the access string u, v represents the prefix generated by the transducer up to u and b is a boolean value which denotes whether the current state satisfies (if it's monotonic) or doesn't satisfy (non-monotonic) the produced prefix. Therefore, states which are parts of non-monotonic extensions are always of the form $(u, v, \mathbf{F})$ while "normal" states are of the form $(u, v, \mathbf{T})$.

Initialization. As in the case of the simple classification tree we initialize the tree with a single domain distinguishing predicate $\mathcal{P}_{\epsilon}^{\texttt{dom}}$. However, in this case we start with two initial leafs, one containing $(\epsilon, \epsilon, \mathbf{T})$ being the $\mathbf{T}$-child of the root node and one containing $(\alpha, \mathbf{f}(\alpha), \mathbf{F})$ being the $\mathbf{F}$-child of the root node. The second leaf represents any possible non-monotonic extensions from the initial state (or the dead-end state if none exist).

Induced NFA construction. Given an extended classification tree, we would like to construct the NFA induced by the tree in order to invoke the `ROLI` algorithm and obtain a transducer model. We proceed similarly as in the deterministic algorithm with two major differences: First, when we compute the transitions from a source state (u, v, b) we utilize path restricted output queries to determine the target for the transition. Second, from each final state (i.e. a state of the form $(u, v\mathbf{T})$), we generate two set of outgoing transitions, one normal monotonic transition and a non-monotonic extension, as follows: For the monotonic extension we compute the outgoing transitions as described above. For the non-monotonic transitions we perform the path restricted queries using the tuple $(u, v, \mathbf{F})$ in order to enforce violation of the monotonicity property.

Extending the classification tree. Extending the extended classification tree with new distinguishing predicates and queries works exactly as in the case of the classification tree.

7.0.6 Induced NFA Verification

Once the induced NFA is constructed and before we invoke the `ROLI` algorithm we proceed to verify that the model is representing an unambiguous transducer model. More specifically, we check whether there exists two different paths leading to a final state. Such paths would violate the single-valuedness property of the transducer and therefore one of them should be invalid. We process such counterexamples as we describe in the corresponding section of ambiguity counterexamples below. Once the NFA model is verified we proceed to invoke the `ROLI` algorithm, generate a transducer model, and process any counterexamples resulting from either the `ROLI` algorithm

or the subsequent equivalence queries. We point out that since SVND transducers presented limited non-determinism, the analysis of the `ROLI` algorithm can be easily ported into this class.

7.0.7 Counterexample Processing

In a nutshell, the counterexample processing algorithm first uses the visibility of non-determinism in order to trace, for each prefix of the counterexample, whether the transition followed by the next character was towards a non-monotonic extension or towards a monotonic extension. After the correct path (i.e. the sequence of non-deterministic choices) are recovered in the target transducer we reduce the problem to the deterministic setting and use the algorithm from our deterministic transducer learning algorithm to further process the counterexample and extract a distinguishing predicate.

Path reconstruction algorithm. We will now proceed to describe our path reconstruction algorithm, which, given a string u returns a sequence of non deterministic choices $p \in \{\mathbf{M}, \mathbf{N}\}^*$ where $p[i] = \mathbf{N}$ if the transducer on input u followed a non-monotonic transition on the target transducer with $u[i]$. The path reconstruction algorithm uses violations of the monotonicity of the target function in order to recover the non-deterministic choices made by the target transducer. The algorithm performs a linear scan on the outputs obtained from all prefixes of u keeps a set T of intervals $[i, j]$ denoting that a non-monotonic extension was followed starting at $u[i]$ and up to $u[j]$. Because certain prefixes of u might follow different paths in the transducer than the ones u is following, some of the paths recovered up to some position i may be invalidated afterwards. More specifically, given a string u the algorithm works as follows:

1. Initially, for each prefix $u[..i-1]$ an output query is performed. For simplicity, denote by $t_i = \mathbf{f}(u[..i-1])$. Moreover the set of intervals is initialized as $T = \emptyset$.

2. For each $i \in [0, |u| - 1$ do:

(a) If $t_i \not\preccurlyeq t_{i+1}$, then let $l < i$ be the largest index such that $t_l \preccurlyeq t_{i+1}$. We add the interval $[l, i+1]$ in the set of intervals T.

(b) Remove and other intervals from T which have a non empty intersection with $[l, i+1]$.

3. Let $p \in \{\mathbf{M}, \mathbf{N}\}^{|u|}$ be the sequence of non-deterministic choices. Set $p[i] = \mathbf{N}$ if there exists an interval which starts at i and $p[i] = \mathbf{M}$ otherwise.

Processing Equivalence query counterexamples. Let s be a counterexample for a model $\mathbf{h}$ produced by the learning algorithm. The overall counterexample processing algorithm proceeds as follows:

1. Run the path reconstruction algorithm and obtain the path p followed by s in the target transducer.

2. Trace the execution of the model $\mathbf{h}$ on the transition choices followed by the path p and execute the deterministic counterexample processing algorithm using *path restricted output queries* to answer any output queries performed by the deterministic counterexample processing algorithm.

The reduction to the deterministic learning algorithm is now straightforward: Because of visibility we can recover the non-deterministic choices in the target transducer and then invoke the deterministic counterexample processing algorithm, taking care to restrict the model in the same path as in the target transducer by utilizing path restricted output queries instead of normal queries. Notice, that because of that, the resulting model as viewed by the deterministic learning algorithm will be partial and therefore, our algorithm from section 6.4 is fundamental for the success of the SVND learning algorithm.

Processing ambiguous input counterexamples. Finally, we now describe a small variation to the above counterexample processing algorithm which can be used in order to process counterexample occurring due to violations of the single valuedness of the model as described above. Given an ambiguous input s for the NFA model $\mathbf{h}$ constructed by the learning algorithm, we proceed as follows: Let p_1, p_2 be the two

distinct ambiguous paths in **h**. We ran the path reconstruction algorithm for the input s and recover that path p_t in the target. Afterwards, let $p_i \in \{p_1, p_2\}$ such that $p_i \neq p_t$. Finally, we run the counterexample processing described above but instead of tracing the execution of the model on p_t we trace the execution on p_i. Notice, that when restricted to p_i, our model **h** reaching a final state, while the target transducer, by using path restricted output queries as an oracle, is returning $\perp$ as an output. Therefore, the string s can be processed as a domain counterexample using the same counterexample processing algorithm we used in the deterministic setting.

7.0.8 Learning Algorithm Summary

Overall, the learning algorithm for SVND follows the same high level loop as the deterministic learning algorithm with the addition of the verification step for the induced NFA model. Once the first model is produced, the same iteration is followed as in the deterministic algorithm where counterexamples (either from `ROLI` or from an equivalence query) are used in order to add new distinguishing predicates in the model until the equivalence query returns **T**.

The following theorem summarizes the correctness and complexity of our algorithm:

Theorem 8. *The class of simple visibly non-deterministic transducers can be learned with $O(mn^3|\Sigma| + nm(n+m))$ output queries and n equivalence queries, where n is the number of states in the target transducer, and m is the length of the longest counterexample.*

The main overhead compared with the deterministic algorithm is the fact that, in many occasions, we replace normal output queries, with path restricted output queries which are more expensive since they require us to query all prefixes of the input string.

Chapter 8

Learning Symbolic Automata

8.1 Background

8.1.1 Boolean Algebras and Symbolic Automata

In symbolic automata, transitions carry predicates over a decidable Boolean algebra. An *effective Boolean algebra* $\mathcal{A}$ is a tuple $(\mathfrak{D}, \Psi, [\_], \bot, \top, \vee, \wedge, \neg)$ where $\mathfrak{D}$ is a set of *domain elements*; Ψ is a set of *predicates* closed under the Boolean connectives, with $\bot, \top \in \Psi$; $[\_] : \Psi \to 2^{\mathfrak{D}}$ is a *denotation function* such that (i) $[\bot] = \emptyset$, (ii) $[\top] = \mathfrak{D}$, and (iii) for all $\varphi, \psi \in \Psi$, $[\varphi \vee \psi] = [\varphi] \cup [\psi]$, $[\varphi \wedge \psi] = [\varphi] \cap [\psi]$, and $[\neg\varphi] = \mathfrak{D} \setminus [\varphi]$.

Example 5 (Equality Algebra)**.** The *equality algebra* for an arbitrary set $\mathfrak{D}$ has predicates formed from Boolean combinations of formulas of the form $\lambda c.\, c = a$ where $a \in \mathfrak{D}$. Formally, Ψ is generated from the Boolean closure of $\Psi_0 = \{\varphi_a \mid a \in \mathfrak{D}\} \cup \{\bot, \top\}$ where for all $a \in \mathfrak{D}$, $[\varphi_a] = \{a\}$. Examples of predicates in this algebra include $\lambda c.\, c = 5 \vee c = 10$ and $\lambda c.\, \neg(c = 0)$.

Definition 17 (Symbolic Finite Automata)**.** A *symbolic finite automaton* (s-FA) M is a tuple $(\mathcal{A}, Q, q_{\text{init}}, F, \Delta)$ where $\mathcal{A}$ is an effective Boolean algebra, called the *alphabet*; Q is a finite set of states; $q_{\text{init}} \in Q$ is the *initial state*; $F \subseteq Q$ is the set of *final states*; and $\Delta \subseteq Q \times \Psi_{\mathcal{A}} \times Q$ is the *transition relation* consisting of a finite set of *moves* or *transitions*.

Characters are elements of $\mathfrak{D}_{\mathcal{A}}$, and *words* or *strings* are finite sequences of characters, or elements of $\mathfrak{D}_{\mathcal{A}}^*$. The empty word of length 0 is denoted by ϵ. A move $\rho = (q_1, \varphi, q_2) \in \Delta$, also denoted by $q_1 \xrightarrow{\varphi} q_2$, is a transition from the *source* state q_1 to the *target* state q_2, where φ is the *guard* or *predicate* of the move. For a state $q \in Q$, we denote by $\texttt{guard}(q)$ the set of guards for all moves from q. For a character $a \in \mathfrak{D}_{\mathcal{A}}$, an *a-move* of M, denoted $q_1 \xrightarrow{a} q_2$ is a move $q_1 \xrightarrow{\varphi} q_2$ such that $a \in [\varphi]$.

An s-FA M is *deterministic* if, for all transitions $(q, \varphi_1, q_1), (q, \varphi_2, q_2) \in \Delta$, $q_1 \neq q_2 \rightarrow [\varphi_1 \wedge \varphi_2] = \emptyset$—i.e., for each state q and character a there is at most one a-move out of q. An s-FA M is *complete* if, for all $q \in Q$, $[\bigvee_{(q,\varphi_i,q_i)\in\Delta} \varphi_i] = \boldsymbol{dom}$—i.e., for each state q and character a there exists an a-move out of q. Throughout the paper we assume all s-FAs are deterministic and complete, since determinization and completion are always possible [39]. Given an s-FA $M = (\mathcal{A}, Q, q_{\text{init}}, F, \Delta)$ and a state $q \in Q$, we say a word $w = a_1 a_2 \cdots a_k$ is *accepted at state* q if, for $1 \leq i \leq k$, there exist moves $q_{i-1} \xrightarrow{a_i} q_i$ such that $q_{\text{init}} = q$ and $q_k \in F$.

For a deterministic s-FA M and a word w, we denote by $M_q[w]$ the state reached in M by w when starting at state q. When q is omitted we assume that execution starts at q_{init}. For a word $w = a_1 \cdots a_k$, we use $w[i..] = a_i \cdots a_k, w[..i] = a_1 \cdots a_i, w[i] = a_i$ to denote the suffix starting from the i-th position, the prefix up to the i-th position and the character at the i-th position respectively. We use $\mathbb{B} = \{\mathbf{T}, \mathbf{F}\}$ to denote the Boolean domain. A word w is called an *access string* for state $q \in Q$ if $M[w] = q$. For two states $q, p \in Q$, a word w is called a distinguishing string, if exactly one of $M_q[w]$ and $M_p[w]$ is final.

8.2 Learning Algorithm Overview

From a mathematical point of view, learning a symbolic state machine amounts to learning the underlying equivalence relation and moreover, learning the predicates that represent the guards between the transitions of the s-FA. In order to provide a general solution to this problem we will assume the existence of an additional learning algorithm for the underlying predicates. Naturally, the ability to learn the underlying

predicates of the target s-FA is a necessary condition in order to guarantee the learnability of the overall s-FA. Next we will discuss two models of learning algorithms for the underlying predicate learning algorithms.

8.2.1 Partition Learning Algorithms

Since we are learning deterministic s-FAs, the set of predicates in the outgoing transitions from any state forms a partition of the symbolic alphabet. Therefore, it is natural to consider algorithms which, given query access to a partition using predicates from the Boolean algebra, are able to recover the partition. As we will see in the MAT^* algorithm, we can use such algorithms as building blocks for our MAT^* algorithm. We will now formally define the problem of learning partitions using queries.

For the following we will consider a partition $\mathcal{S}$ to be a set $\mathcal{S} = \{\phi_1, \ldots, \phi_k\}$ of predicates from a Boolean algebra $\mathcal{A}$ such that $\vee_{\phi \in S} = \mathbf{T}$ and for any $\phi_i, \phi_j \in \mathcal{S}$ with $i \neq j$ we have that $\phi_i \wedge \phi_j = \mathbf{F}$.

Definition 18. In the partition learning problem, query access is given to a target partition $\mathcal{S} = \{\phi_1, \ldots, \phi_k\}$. Queries to the target partition can be performed as follows:

- **Membership Queries:** In a memebership query the input is a symbol $c \in \mathfrak{D}$ and the output returned is

$$\mathcal{S}(c) \stackrel{\text{def}}{=} i > 0 : c \in [\phi_i] \tag{8.1}$$

  In other words, given a symbol c the index of the predicate satisfied by the symbol is given.

- **Equivalence Queries:** In an equivalence query, a model $\mathcal{H}$ of the target partition is provided and the equivalence oracle returns either $\mathbf{T}$ if $\mathcal{H} = \mathcal{S}$, or a symbol c such that $\mathcal{H}(c) \neq \mathcal{S}(c)$.

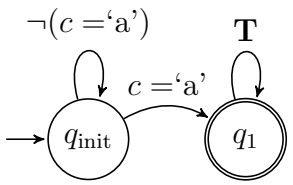

Figure 8-1: An s-FA over equality algebra.

Now that we defined the partition learning problem, we will proceed to define a partition learning algorithm, which is a MAT learning algorithm for the partition learning problem.

Definition 19. A partition learning algorithm $\mathcal{P}_{\mathcal{A}}$ for a boolean algebra $\mathcal{A}$ is a MAT learning algorithm which can learn partitions using predicates from the Boolean algebra $\mathcal{A}$. We will assume that partition learning algorithms perform *proper* equivalence queries, i.e. all models $\mathcal{S}'$ submitted for equivalence checking will satisfy the partition definition.

8.2.2 Predicate Learning Algorithm

While partition learning algorithms are a natural candidate for learning the transitions of the target s-FA they have a significant disadvantage. They do not directly relate the learnability of the underlying predicates with the learnability of the target s-FA. In other words, given an algorithm which is able to learn predicates from the underlying Boolean algebra efficiently, can we guarantee the efficient learnability of a target s-FA which uses predicates from the same Boolean algebra?

Next, we will describe the MAT^* algorithm which provides an answer to this fundamental question. Also we will see that MAT^* can be easily adapted in order to utilize an underlying partition learning algorithm.

8.3 The MAT^* Algorithm

Overview. The main idea behind the MAT^* algorithm is simple: We utilize the traditional L^*algorithm in order to approximate the Nerode congruence of the target

Algorithm 7 s-FA-Learn($\mathcal{O}, \mathcal{E}, \Lambda$) // s-FA Learning algorithm

Require: $\mathcal{O}$: membership oracle, $\mathcal{E}$: equivalence oracle, Λ: algebra learning algorithm.

```
T ← InitializeClassificationTree(O)
S_Λ ← InitializeGuardLearners(T, Λ)
H ← GetSFAModel(T, S_Λ, O)
while E(H) ≠ T do
    w ← GetCounterexample(H)
    T, S_Λ ← ProcessCounterexample(T, S_Λ, w, O)
    H ← GetSFAModel(T, S_Λ, O)
return H
```

s-FA and then, we utilize either the partition or the predicate learning algorithms in order to learn the underlying predicates and build an s-FA model. The main challenge in utilizing the underlying partition or predicate learning algorithms is the lack of a membership and an equivalence oracle for the corresponding target predicates. To address this problem we will show how to utilize the partial congruence relation in order to simulate membership queries to the underlying predicates and moreoveer, utilize the s-FA equivalence oracle to simulate equivalence queries for the underlying predicates.

The pseudocode for the overall MAT^* algorithm can be found in algorithm 7. Observe that the high level structure of the algorithm is identical with the L^*algorithm. The algorithm starts by constructing a simple congruence which is used in order to built the s-FA model. Once a model is built, we submit it for equivalence testing. However, given a counterexample we use it to either extend the congruence or in order to refine the predicates in the s-FA model.

We will now describe each module of the MAT^* algorithm in more detail.

8.3.1 Contructing an s-FA model

Assume we are given a classification tree $T = (V, L, E)$. Our next task is to use the tree along with the underlying algebra learning algorithm Λ to produce an s-FA model. The main idea is to spawn an instance of the Λ algorithm for each potential transition and then use the classification tree to answer membership queries posed

by each Λ instance. Initially, we define an s-FA $\mathcal{H} = (\mathcal{A}, Q_{\mathcal{H}}, q_\epsilon, F_{\mathcal{H}}, \Delta_{\mathcal{H}})$, where $Q_{\mathcal{H}} = \{q_s \mid s \in L\}$—i.e. we create one state for each leaf of the classification tree T. Finally, for any $q \in Q_{\mathcal{H}}$, we have that $q \in F_{\mathcal{H}}$ if and only if $\mathcal{O}(q) = \mathbf{T}$. Next, we will show how to build the transition relation for $\mathcal{H}$. As mentioned above, our construction is based on the idea of spawning instances of Λ for each potential transition of the s-FA and then using the classification tree to decide, for each character, if the character satisfies the guard of the potential transition thus answering membership queries performed by the underlying algebra learner.

Constructing a model using a predicate learning algorithm

Guard inference. To infer the set of guards in the transition relation $\Delta_{\mathcal{H}}$, we spawn, for each pair of states $(q_u, q_v) \in Q_{\mathcal{H}} \times Q_{\mathcal{H}}$, an instance $\Lambda^{(q_u,q_v)}$ of the algebra learning algorithm. We answer membership queries to $\Lambda^{(q_u,q_v)}$ as follows. Let $\alpha \in \mathfrak{D}$ be a symbol queried by $\Lambda^{(q_u,q_v)}$. Then, we return $\mathbf{T}$ as the answer to $\mathcal{O}(\alpha)$ if $\texttt{sift}(u\alpha) = v$ and $\mathbf{F}$ otherwise. Once $\Lambda^{(q_u,q_v)}$ submits an equivalence query $\mathcal{E}(\phi)$ using a model ϕ, we suspend the execution of the algorithm and add the transition (q_u, ϕ, q_v) in $\Delta_{\mathcal{H}}$.

Partition verification. Once all algebra learners have submitted a model through an equivalence query, we have a complete transition relation $\Delta_{\mathcal{H}}$. However, at this point there is no guarantee that for each state q the outgoing transitions from q form a partition of the domain $\mathfrak{D}$. Therefore, it may be the case that our s-FA model $\mathcal{H}$ is in fact non-deterministic and, moreover, that certain symbols do not satisfy any guard. Using such a model in an equivalence query would result in an *improper* learning algorithm and potential problems in the counterexample processing algorithm in Section 8.3.2. To mitigate this issue we perform the following checks:

1. **Determinism check:** For each state $q_s \in Q_{\mathcal{H}}$ and each pair of moves $(q_s, \phi_1, q_u), (q_s, \phi_2, q_v) \in \Delta_{\mathcal{H}}$, we verify that $[\phi_1 \wedge \phi_2] = \emptyset$. Assume that a character α is found such that $\alpha \in [\phi_1 \wedge \phi_2]$ and let $m = \texttt{sift}(s\alpha)$. Then, it must be the case that the guard of the transition $q_s \to q_m$ must satisfy α. Therefore, we check if $m = u$ and $m = v$ and provide α as a counterexample to

$\Lambda^{(q_s,q_u)}$ and $\Lambda^{(q_s,q_v)}$ respectively if the corresponding check fails.

2. **Completeness check.** For each state $q_u \in Q_{\mathcal{H}}$ let $S = \{\phi \mid (q, \phi, p) \in \Delta_{\mathcal{H}}\}$. We check that $[\bigvee_{\phi \in S} \phi] = \mathfrak{D}$. If a symbol $h \notin [\bigvee_{\phi \in S} \phi]$ is found then, let $v = \texttt{sift}(uh)$. Following the same reasoning as above, we provide h as a counterexample to $\Lambda^{(q_u,q_v)}$.

These checks are iterated for each state until no more counterexamples are found. In figure 8-2 we demonstrate instances of failed determinism and completeness checks while learning our running example from figure 8-1 along with the corresponding updates on the predicates. For details regarding the equality algebra learner, see section 8.5.

Optimizing the number of algebra learning instances. Note that in the description above, MAT^* spawns one instance of Λ for each possible transition between states in $\mathcal{H}$. To reduce the number of spawned algebra learning instances, we perform the following optimization: For each state q_s we initially spawn a single algebra learning instance $\Lambda^{(q_s,?)}$. Let α be the first symbol queried by $\Lambda^{(q_s,?)}$ and let $u = \texttt{sift}(s\alpha)$. We return $\top$ as a query answer for α to $\Lambda^{(q_s,?)}$ and set the target state for the instance to q_u, i.e. we convert the algebra learning instance to $\Lambda^{(q_s,q_u)}$. Afterwards, we keep a set $R = \{q_v \mid v = \texttt{sift}(s\beta)\}$ for all $\beta \in \mathfrak{D}$ queried by the different algebra learning instances and generate new instances only for states $q_v \in R$ for which the guards are not yet inferred. Using this optimization, the total number of generated algebra learning instances never exceeds the number of transitions in the target s-FA.

Constructing a model using a partition learning algorithm

When we are given access to a partition learning algorithm instead of a predicate learning algorithm, the construction of the s-FA model is much simples, since the partition verification step performed above can be skipped as all models produced by the partition learning algorithm $\mathcal{P}$ will already be partitions of the alphabet.

Building the model. In order to build the s-FA model, from each state $q_u \in Q$ we spawn a single instance of the predicate learning algorithm $\mathcal{P}^{(q_u)}$ and answer

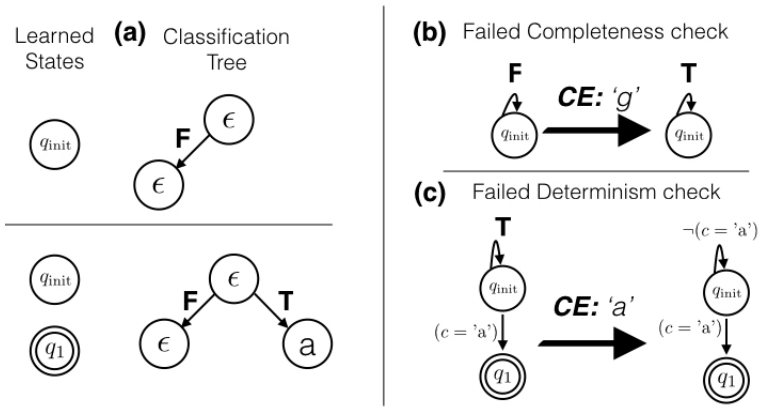

Figure 8-2: (left) Classification tree and corresponding learned states for our running example. (right) Two different instances of failed partition verification checks that occured during learning and their respective updates on the given counterexamples (CE).

membership queries as follows:

$$\mathcal{O}(c) = \texttt{sift}(u\alpha) \tag{8.2}$$

In other words, we answer membership queries in the same exact way, but instead of answering with a Boolean $\{\mathbf{T}, \mathbf{F}\}$ we forward as the label of the target predicate to be the accessing string for the state accessed using the queried symbol c.

Once an equivalence query is performed by the partition learning algorithm $\mathcal{P}^{(q_u)}$, we inspect the generated partition $\mathcal{H}$ and for each predicate $\phi_v \in \mathcal{H}$ we add the transition (u, ϕ_v, v) in the s-FA model. Once this process is repeated for all states, we have a complete s-FA model.

8.3.2 Counterexample Processing

In a nutshell, our algorithm works similarly to the classic Rivest-Schapire algorithm [75] and the TTT algorithm [51] for learning DFAs, where a binary search is performed to locate the index in the counterexample where the executions of the model automaton and the target one diverge. However, once this breakpoint index is found, our algorithm performs further analysis to determine if the divergence is caused by an undiscovered state in our model automaton or because the guard predicate that

consumes the breakpoint index character is incorrect.

Error localization. Let w be a counterexample for a model $\mathcal{H}$ generated as described above. For each index $i \in [1..|w|]$, let $q_u = \mathcal{H}[w[..i]]$ be the state accessed by $w[..i]$ in $\mathcal{H}$ and let $\gamma_i = uw[i+1..]$. In other words, γ_i is obtained by first running w in $\mathcal{H}$ for i steps and then, concatenating the access string for the state reached in $\mathcal{H}$ with the word $w[i+1..]$. Note that, because initially the model $\mathcal{H}$ and the target s-FA start at the same state accessed by ϵ, the two machines are synchronized and therefore, $\mathcal{O}(\gamma_0) = \mathcal{O}(w)$. Moreover, since w is a counterexample, we have that $\mathcal{O}(\gamma_{|w|}) \neq \mathcal{O}(w)$. It follows that, there exists an index j, which we will refer to as *breakpoint*, for which $\mathcal{O}(\gamma_j) \neq \mathcal{O}(\gamma_{j+1})$. The counterexample processing algorithm uses a binary search on the index j to find such a breakpoint.

Breakpoint analysis. Once we find an index j such that $\mathcal{O}(\gamma_j) \neq \mathcal{O}(\gamma_{j+1})$ we can conclude that the transition taken in $\mathcal{H}$ from $\mathcal{H}[w[..j]]$ with the symbol $w[j+1]$ is incorrect. In traditional algorithms for learning DFAs, the sole reason for having an incorrect transition would be that the transition is actually directed to a yet undiscovered state in the target automaton. However, in the symbolic setting we have to explore two different possibilities. Let $q_u = \mathcal{H}[w[..j]]$ be the state accessed in $\mathcal{H}$ by $w[..j]$, $q_v = \texttt{sift}(uw[j+1])$ be the result of sifting $uw[j+1]$ in the classification tree and consider the transition $(q_u, \phi, q_v) \in \Delta_{\mathcal{H}}$. We use the guard ϕ to determine if the counterexample was caused by an invalid predicate guard or an undiscovered state in the target s-FA.

Case 1. Incorrect guard. Assume that $w[j+1] \notin [\phi]$. Note that, ϕ was generated as a model by $\Lambda^{(q_u,q_v)}$ and therefore, a membership query from $\Lambda^{(q_u,q_v)}$ for a character α returns $\mathbf{T}$ if $\texttt{sift}(u\alpha) = v$. Moreover, we have that $\texttt{sift}(uw[j+1]) = v$. Therefore, if $w[j+1] \notin [\phi]$, then $w[j+1]$ is a counterexample for the learning instance $\Lambda^{(q_u,q_v)}$ which produced ϕ. We proceed to supply $\Lambda^{(q_u,q_v)}$ with the counterexample $w[j+1]$, update the corresponding guard or partition and proceed to generate a new s-FA model.

Case 2. Undiscovered state. Assume $w[j+1] \in [\phi]$. It follows that ϕ is behaving as

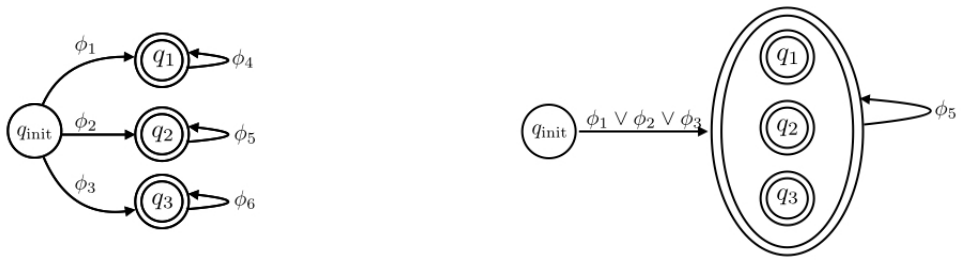

Figure 8-3: (left) A minimal s-FA. (right) The s-FA corresponding to the classification tree of MAT^* with access strings for q_{init} and q_2 and a single distinguishing string ϵ.

expected on the symbol $w[j+1]$ based on the current classification tree. We conclude that the state accessed by $w[..j+1]$ is in fact an undiscovered state in the target s-FA which we have to distinguish from the previously discovered states. Therefore, we proceed to add a new leaf in the tree to access this state. More specifically, we replace the leaf labelled with v with a sub-tree consisting of three nodes: the root is the word $w[j+1..]$, which is the distinguishing string for the states accessed by v and $uw[j+1]$. The **T**-child and **F**-child of this node are labelled with the words v and $uw[j]$ based on the results of $\mathcal{O}(v)$ and $\mathcal{O}(uw[j+1])$.

Finally, we have to take care of one last point: Once we add another state in the classification tree, certain queries that were previously directed to v may be directed to $uw[j]$ once we sift them down in the tree. This change implies that certain previous queries performed by algebra learning instances $\Lambda^{(q_s,q_v)}$ may be given invalid results and therefore, we can no longer guarantee correctness of the generated predicates. To solve this problem, we terminate all instances $\Lambda^{(q_s,q_v)}$ for all $q_s \in Q_{\mathcal{H}}$ and replace them with fresh instances of the algebra learning algorithm.

8.4 Correctness and Completeness of MAT^*

Given a learning algorithm Λ, we use $\mathcal{C}_{\text{m}}^{\Lambda}(n)$ to denote the number of membership queries and $\mathcal{C}_{\text{e}}^{\Lambda}(n)$ to denote the number of equivalence queries performed by Λ for a target concept with representation size n. In our analysis we will also use the following definitions:

Definition 20. Let $\mathcal{M} = (\mathcal{A}, Q, q_0, F, \Delta)$ over a Boolean algebra $\mathcal{A}$ and let $S \subseteq \Psi_{\mathcal{A}}$.

Then, we define:

- The maximum size of the union of predicates in S as $\mathcal{U}(S) \stackrel{\text{def}}{=} \max_{\Phi \subseteq S} |\bigvee_{\phi \in \Phi} \phi|$.
- The maximum guard union size for $\mathcal{M}$ as $\mathcal{B}(\mathcal{M}) \stackrel{\text{def}}{=} \max_{q \in Q} \mathcal{U}(\texttt{guard}(q))$.

The value $\mathcal{B}(\mathcal{M})$ denotes the maximum size that a predicate guard may take in any intermediate hypothesis produced by MAT^* during the learning process. Contrary to traditional L^*-style algorithms, the size of the intermediate hypothesis produced by MAT^* may fluctuate as we demonstrate in the following example.

Example 6. Consider the s-FA in the left side of figure 8-3. When we execute the MAT^* algorithm in this s-FA, and after an access string for q_2 is added to the classification tree, the tree will correspond to the s-FA shown on the right, in which the transition from q_{init} is taken over the union of the individual transitions in the target. Certain sequences of answers to equivalence queries can force MAT^* to first learn a correct model of $\phi_1 \vee \phi_2 \vee \phi_3$ before revealing a new state in the target s-FA.

We now state the correctness and query complexity of our algorithm.

Theorem 9. *Let $\mathcal{M} = (\mathcal{A}, Q, q_0, F, \Delta)$ be an s-FA, Λ be a learning algorithm $\mathcal{A}$ and let $k = \mathcal{B}(\mathcal{M})$. Then, MAT^* will learn $\mathcal{M}$ using Λ with $O(|Q|^2|\Delta|\mathcal{C}_m^\Lambda(k) + |Q|^2|\Delta|\mathcal{C}_e^\Lambda(k)\log m)$ membership and $O(|Q||\Delta|\mathcal{C}_e^\Lambda(k))$ equivalence queries, where m is the length of the longest counterexample given to MAT^*.*

Proof. First, we note that our counterexample processing algorithm only splits a leaf if there exists a valid distinguishing condition separating the two newly generated leafs. Therefore, the number of leafs in the discrimination tree is always at most $|Q|$. Next, note that each counterexample is processed using a binary search with complexity $O(\log m)$ to detect the breakpoint and, afterwards, either a new state is added or a counterexample is dispatched to the corresponding algebra learner.

Each classification tree $T = (V, L, E)$ defines a partition over $\boldsymbol{dom}^*$ and, therefore, an $s - FA\ \mathcal{H}_T$. In the worst case, MAT^* will learn $\mathcal{H}_T$ exactly before a new state in the target s-FA is revealed through an equivalence query. Since $\mathcal{H}_T$ is the result of

merging states in the target s-FA, we conclude that the size of each predicate in $\mathcal{H}_T$ is at most k. It follows that, for each classification tree T, we can get at most $|\Delta_{\mathcal{H}_T}|\mathcal{C}_{\mathrm{e}}^{\Lambda}(k)$ counterexamples until a new state is uncovered on the target s-FA. Note here, that our counterexample processing algorithm ensures that each counterexample will be either a valid counterexample for a predicate guard in $\mathcal{H}_T$ or it will uncover a new state. For each membership query performed by an underlying algebra learner, we have to sift a string in the classification tree which requires at most $|Q|$ membership queries. Therefore, the total number of membership queries performed for each candidate model $\mathcal{H}$ is bounded by $O(|\Delta|(|Q|\mathcal{C}_{\mathrm{m}}^{\Lambda}(k) + \mathcal{C}_{\mathrm{e}}^{\Lambda}(k)\log m)$ where m is the size of the longest counterexample so far. The number of equivalence queries is bounded by $O(|\Delta|\mathcal{C}_{\mathrm{e}}^{\Lambda}(k))$. When a new state is uncovered, we assume that, in the worst case, all the algebra learners will be restarted (this is an overestimation) and therefore, the same process will be repeated at most $|Q|$ times giving us the stated bounds. □

Note that the bounds on the number of queries stated in theorem 9 are based on the worst-case assumption that we may have to restart *all* guard learning instances each time we discover a new state. In practice, we expect these bounds to be closer $O(|\Delta|\mathcal{C}_{\mathrm{m}}^{\Lambda}(k) + (|\Delta|\mathcal{C}_{\mathrm{e}}^{\Lambda}(k) + |Q|)\log m)$ membership and $O(|\Delta|\mathcal{C}_{\mathrm{e}}^{\Lambda}(k) + |Q|)$ equivalence queries.

Minimality of learned s-FA.

Since the MAT^* will only add a new state in the s-FA if a distinguishing sequence is found it follows that the total number of states in the s-FA is minimal. Moreover, MAT^* will not modify in any way the predicates returned by the underlying algebra learning instances. Therefore, if the size of the predicates returned by the Λ instances is minimal, MAT^* will maintain their minimality.

The following theorem shows that it is indeed not possible to learn s-FAs over a Boolean algebra that is not itself learnable.

Theorem 10. *Let Λ^{s-FA} be an efficient learning algorithm for the algebra of s-FAs over a Boolean algebra $\mathcal{A}$. Then, the Boolean algebra $\mathcal{A}$ is efficiently learnable.*

Which s-FAs are efficiently learnable?

Theorem 10 shows that efficient lernability of an s-FA requires efficient learnability of the underlying algebra. Moreover, from theorem 9 it follows that efficiently learnability using MAT^* depends on the following property of the underlying algebra:

Corollary 1. *Let $\mathcal{A}$ be an efficiently learnable Boolean algebra and consider the class $\mathcal{R}_{\mathcal{A}}^{s-FA}$ of s-FAs over $\mathcal{A}$. Then, $\mathcal{R}_{\mathcal{A}}^{s-FA}$ is efficiently learnable using MAT^* if and only if, for any set $S \subseteq \Psi_{\mathcal{A}}$ such that for any distinct $\phi, \psi \in S \implies [\phi \wedge \psi] = \emptyset$, we have that $\mathcal{U}(S) = \texttt{poly}(|S|, \max_{\phi \in S} |\phi|)$.*

At this point we would like to point out that the above condition arises due to the fact that MAT^* is a congruence-based algorithm which successively computes hypothesis automata based on refining a set of access and distinguishing strings which is a common characteristic among all L^*-based algorithms. Therefore, this limitation of MAT^* is expected to be shared by any other algorithm in the same family. Given the fact that after three decades of research, L^*-based algorithms are the only known, provably efficient algorithms for learning DFAs (and subsequently s-FAs), we expect that expanding the class of learnable s-FAs is a very challenging task.

8.5 Learnable Boolean Algebras

We will now describe a number of interesting effective Boolean algebras which are efficiently learnable using membership and equivalence queries.

Boolean Algebras over finite domains. Let $\mathcal{A}$ be any Boolean Algebra over a finite domain $\boldsymbol{dom}$. Then, any predicate $\phi \in \Psi$ can be learned using $|\boldsymbol{dom}|$ membership queries. More specifically, the learning algorithm constructs a predicate ϕ accepting all elements in $\boldsymbol{dom}$ for which the membership queries return true as $\phi = \{c \mid c \in \boldsymbol{dom} \wedge \mathcal{O}(c) = \mathbf{T}\}$. Plugging this algebra learning algorithm into our algorithm, we get the TTT learning algorithm for DFAs without discriminator finalization [51]. This simple example demonstrates that algorithms for DFAs can be viewed as special cases of our s-FA learning algorithm for finite domains.

Equality Algebra. Consider the equality algebra defined in example 5. Predicates in this algebra of size $|\phi| = k$ can be learned using $2k$ equivalence queries and no membership queries. Initially, the algorithm outputs the empty set $\perp$ as a hypothesis. In any subsequent step, the algorithm keeps a list of the counterexamples obtained so far in two sets $P, N \subseteq \boldsymbol{dom}$ such that P holds all the positive examples received so far and N holds all the negative examples. Afterwards, the algorithm finds the smallest hypothesis consistent with the counterexamples given. This hypothesis can be found efficiently as follows:

1. If $|P| > |N|$ then, $\phi = \lambda c. \neg(\bigvee_{d \in N} c = d)$.

2. If $|P| \leq |N|$ then, $\phi = \lambda c. (\bigvee_{d \in P} c = d)$.

It can be easily shown that the algorithm will find a correct hypothesis after at most $2k$ equivalence queries.

Other Algebras. The following Boolean algebras can be efficiently learned using membership and equivalence queries. All these algebras also have approximate fingerprints [14], which means that they are not learnable by equivalence queries alone.

1. **BDD algebra.** The algebra of ordered binary decision diagrams (OBDDs) is efficiently learnable using a variant of the L* algorithm [66].

2. **Tree automata algebra.** Deterministic finite tree automata form an algebra which is also learnable using membership and equivalence queries [44].

3. **s-FA algebra.** s-FAs themselves form an effective Boolean algebra and therefore, s-FAs over s-FAs over learnable algebras are also learnable.

8.6 Learning Equality Partitions from Data

In this section we are going to explore algorithms with which we can solve the partition learning problem using the corpus of data C. Specifically, we will describe a maximum likehood estimation (MLE) algorithm for the partition learning problem.

An advantage of our algorithm is that it works for any predicate family, since we assume no specific structure in the predicates themselves.

A first approach in order to compute the MLE partition would be, given the training set, to directly compute the most likely partition given the corpus C, by taking each partition as an individual element and find the partition with the highest likehood. However, the number of different partitions for large sets is *huge* and therefore, this approach is unlikely to yield any practical results.

A more natural and practical approach is to assume that the occurrence of each set in a partition is independent from the others sets in the same partition conditioned on the fact the collection of sets still forms a partition.

Definition 21. In the Partition Learning Maximum Likehood Estimation problem (`PL-MLE`) the input is a tuple $(T, \Sigma, \mathcal{P}, f_P(\cdot))$ where, f_P is a partial function $f_P : \mathcal{P} \to [0, 1]$ of probabilities over subsets of Σ and $T = \{s_1, s_2, \ldots, s_k\}$ is the training data. The goal in the `PL-MLE` is to find a set $P_m = \{S_1, S_2, \ldots, S_k\}, S_i \in \mathcal{P}$ such that:

1. $\forall i \in [k], S_i \supseteq s_i$.
2. $\bigcup_{i\in[k]} S_i = \Sigma$.
3. $\forall i, j$ s.t $i \neq j \implies S_i \cap S_j = \emptyset$.
4. $\max \prod_{i\in[k]} \Pr[S_i] = \max \prod_{i\in[k]} f_P(S_i) = \max \sum_{i\in[k]} \log f_P(S_i)$.

The first condition implies that when generalizing the observations we will retain the already known symbols into the correct set, the second and third conditions imply that the sets we will select will indeed form a partition of our alphabet while the last condition asserts that we will select the sets that maximize the overall likehood. One technical point that one should take into account is defining the function $f_P(\cdot)$ which defines the probability of each predicate in $\mathcal{P}$. Since the domain of the function is $|\mathcal{P}| \leq 2^{|\Sigma|}$ defining each individual value will require the same amount of space to be given as input. To avoid this exponential blowup in terms of the size of the input we assume that any point in the function not specified in the input is being given an equal uniform value, normalized in order to satisfy $\sum_{S\in\mathcal{P}} f_P(S) = 1$.

Unfortunately, finding such a partition is a computationally intractable problem as we prove in the following theorem.

Theorem 11. *The decision* `PL-MLE` *problem is NP-Complete.*

8.6.1 A Greedy MLE algorithm

Algorithm 8 Greedy partition learning MLE algorithm.

Require: T is a set of tuples (q, Sq) where q is a label and $S_q \subseteq \Sigma$, $\mathcal{P} \subseteq \mathscr{P}(\Sigma)$ is a predicate family and f_P is a function $f_P : \mathcal{P} \to [0, 1]$.

function GREEDYPL-MLE($T, \Sigma, \mathcal{P}, f_P$)
  $P \leftarrow \emptyset$
  $\mathcal{M} \leftarrow \emptyset$
  $D \leftarrow \bigcup_{(q,S_q)\in T} S_q$
  while $|T| > 1$ **do**
    for $(q, S_q) \in T$ **do**
      $\phi_q \leftarrow \arg\max_{\phi\in\mathcal{P}} \mathcal{I}\{S_q \subseteq \phi \wedge \phi \cap D \setminus S_q = \emptyset\} f_P(\phi)$
      $\mathcal{M} \leftarrow \mathcal{M} \cup \{(S_q, \phi_q)\}$
    if $\max_{(S_q,\phi_q)\in\mathcal{M}} f_P(\phi_q) =$ `min_prob` **then**
      $S_{q_M}, \phi_M \leftarrow \arg\min_{(S_q,\phi_q)\in\mathcal{M}} |S_q|$
    else
      $S_{q_M}, \phi_M \leftarrow \arg\max_{(S_q,\phi)\in\mathcal{M}} f_P(\phi)$
    $P \leftarrow P \cup \{\phi_M\}$
    $D \leftarrow D \cup \phi_M$
    $T \leftarrow T \setminus \{(q_M, S_{q_M})\}$
  $P \leftarrow P \cup \{\Sigma \setminus D\}$
  return P

Given the NP-Hardness of computing the MLE for the partition learning problem we will now describe a greedy algorithm which repeatedly assigns the label with the highest probability that is consistent with the data to the corresponding set.

Technical Description.

The pseudocode for the algorithm is presented in algorithm 8. We will now describe each step of the algorithm in more detail. Initially, the algorithm will initialize a set

$D = \cup_{(q,S_q)} S_q$ which contains all the symbols available in the training data (line 4). The goal here is collect all symbols which are already assigned into a label by the training data in order to avoid reassigned them. The next step (line 7) is to select, for each label q the set with the maximum likehood that is compatible with the currently assigned data D and is a superset of the training data available for the label q. The expression $\mathcal{I}\{S \subseteq \phi \wedge \phi \cap D \setminus S_q = \emptyset\}$ denotes an indicator variable, taking a value $1/0$ depending on whether the specified expression holds.

After this process is repeated for all labels, we select the predicate with the highest overall probability (line 13). In case that the maximum (and therefore, all other labels) is equal to the minimum uniform probability, then the algorithm will choose the label with the smallest training data size and generate a predicate containing only the training data and assign it to the corresponding label.

Once this label is assigned, we add all symbols belonging to the newly assigned predicate to the set D in order to exclude them from further selection in the future, remove the label we just assigned from the training data and repeat the process (lines 15-17).

Finally, once all but one labels are assigned, we break out of the main loop and assign the final label to the remaining symbols (line 19). This avoids many technical difficulties since we can guarantee completeness in our partition regardless of the assignment of the first $n-1$ sets. Notice, that this assignment assumes that $\{\Sigma \setminus D\} \in \mathcal{P}$ which may require the assumption that $\mathcal{P} = \mathscr{P}(\Sigma)$.

Correctness and analysis.

The correctness of our algorithm is evident from the technical description above. Indeed, the way we select the predicate with the maximum likehood (line 7) ensures that each selected predicate is a superset of the training data and also that it does not intersect with any other predicates already selected. Finally, the way the last predicate is selected (line 19) ensures that the final set of predicates will partition the alphabet Σ.

The next natural question is how close is our greedy algorithm to the optimal

likehood. It is very easy to show that we will always be within a factor $|T|$ from the log-likehood of the optimal solution.

Theorem 12. *Let $(\Sigma, T, \mathcal{P}, f_P(\cdot))$, where $|T| = k$, be an instance of the* `PL-MLE` *problem. Also, let $C_{OPT} = \max \sum_{i\in[k]} \log f_P(S_i)$ be the optimal log-likehood to the* `PL-MLE` *instance and C_G be the log-estimate of the greedy MLE algorithm. Then, $C_{OPT} \leq k \cdot C_G$.*

We can also show that this simple analysis is tight. In the following we denote by $negl(x)$ to be a negligible function of x, i.e. a function smaller than any polynomial.

Theorem 13. *There exists an instance of the* `PL-MLE` *problem such that $C_{OPT} = kC_G - negl(C_{OPT})$.*

Nevertheless, in practice, the average number of transitions in the SFAs we encountered both in our training set as well as in our test set, have small number of transitions, around $|T| = 2$ on average, and therefore, this greedy algorithm tends to perform well in practice.

8.6.2 A frequency based `GuardGen` algorithm

We will now use our greedy MLE algorithm in order to built a simple frequency based `GuardGen` algorithm, called `MLEGuardgen`, for inferring the transitions of an SFA. Our algorithm presents a very simple, yet efficient, construction which demonstrates the possibilities of our hybrid learning model and our MLE framework.

Training the `GuardGen` algorithm.

Let C be a corpus of SFAs and $M_i = (Q_i, F, q_0, \mathcal{P}, \Delta) \in C$ be an SFA om C. Then, we collect the multiset of predicate guards as

$$\mathcal{G} = \bigcup_{M_i \in C} \bigcup_{q \in Q_i} \bigcup_{\phi \in \texttt{guard}(q)} \phi$$

Notice here, that $\mathcal{G}$ is a multiset thus, repetitions of members are allowed. Given the multiset of all predicate guards we are going to create a probability distribution

over all predicate guards in the predicate family $\mathcal{P}$. Next, we process each predicate from $\mathcal{G}$ and assign a score as $\texttt{score}(\phi) = \texttt{count}_{\mathcal{G}}(\phi)$, where the $\texttt{count}_{\mathcal{G}}(\phi)$ function counts the number of occurences of element ϕ in the multiset $\mathcal{G}$. For all predicates ϕ which are not found in our training data we set $\texttt{score}(\phi) = 0$. Finally, converting the generated scores to a probability distribution is simply a matter of normalization which can be achieved by using the softmax function as:

$$\Pr[\phi] = \sigma(\texttt{score}(\phi)) = \frac{e^{\texttt{score}(\phi)}}{\sum_{\phi_i \in \mathcal{P}} e^{\texttt{score}(\phi_i)}} \tag{8.3}$$

We point out however that, unless a confidence value is required, the score by itself is enough in order to run the greedy MLE algorithm and generate the predicate guards.

Technical Description.

After obtaining the probability distribution from the training phase, the algorithm will initially make a membership query for a random symbol in the alphabet and use the MLE algorithm in order to produce a set P_g of predicate guards for which we submit an equivalence query. Each time a counterexample is given, the corresponding symbol is added to the training data given to the MLE algorithm and a new set of guards is produced until a correct partition is produced.

Analysis. It is evident that for any finite set Σ and a predicate family $\mathcal{P}$ over Σ our algorithm will learn any partition of Σ over $\mathcal{P}$ using at most Σ equivalence queries since, at that point, every element of Σ will be labelled by the equivalence queries made by the algorithm. Proving better bounds on the query complexity of our algorithm depends on the quality of the corpus of SFAs given to the algorithm.

Chapter 9

Applications

Now that we have discussed a large body of learning algorithms for automata and transducers, we will switch gears into algorithms that allow us to use our novel learning algorithms in order to perform black-box testing of large systems.

9.1 Code Injection Attacks

The main application of our techniques lies in the detection of code injection attacks in Web applications. Code injection attacks occur when the application, while processing input data from an utrusted source (typically the user), is confusing part of the data for code a fact which may alter the code executed in various runtimes within the application such as the database, LDAP directories, XML or client side code such as HTML/Javascript. The impact of such attacks can be severe and ranges from the execution of untrusted code into either the Web application server or the client's browser, leaking of sensitive information and others. The OWASP Top Ten, is an annual document which categorizes the top 10 security vulnerabilities for Web Applications in terms of severity. We note that for 2017 the most dangerous vulnerability for Web applications was code injection attacks while the seventh place was also taken by Cross Site Scripting (XSS) another popular code injection vulnerability class.

9.2 Web Application Firewalls and String Sanitizers

There are many different ways with which an application can defend against a code injection attack, but in this chapter we will focus on the two most popular defenses and formalize them in order to fit into our formal models framework.

Web Application Firewalls (WAFs) and Filters. Web Application Firewalls (WAFs) are system which preprocess the input to a Web application and try to detect whether the input contains any malicious data such as an injection attack. If a malicious input is detected, then the request is dropped (or in certain cases just logged). In more general terms we say that WAFs are a type of *filter*. Formally, a filter is defined as a Boolean function $\mathbf{f} : \Sigma \to \mathbb{B}$, such that for any string s, $\mathbf{f}(s) = \mathbf{T}$ only if s belongs to a set of *malicious* inputs. A filter can either be a system such as a WAF which pre-processes the input to the application or a security module within the application that processes the input before it is passed to security critical components. An important aspect of Web Application Firewalls is that they are required by Web applications in order to comply to the PCI standard which set the requirements for web applications which process credit card information. Therefore, it is evident that WAFs are very common systems in industrial environments.

String sanitizers While filters are a common first line of defense, eventually, we would like the Web application to be able to process all requests without having to drop requests which may be malicious: indeed, it is difficult to precisely determine the inputs which contain malicious data and therefore, dropping requests may have the effect of failing to process benign requests which are incorrectly deemed malicious (false positives). A more natural approach is to build functions which *clean* the input from potentially malicious data and then the application can proceed to process the *sanitized* input normally. In practice, such functions are implemented through a series of string transformations in the original input such as encoding potentially dangerous characters, removing malicious parts of the inputs and other similar transformations. More formally, we define a *string sanitizer* to be a function $\mathbf{f} : \Sigma^* \to \Sigma^*$ which takes as input any string s and produces a string $u = \mathbf{f}(s)$ such that u does not contain

any malicious substring.

In the following sections, we will describe algorithms which can be utilized in order to evaluate the robustness of filters and string sanitizers in a *black-box* manner, i.e. given only query access to the target application.

9.3 Grammar Oriented Filter Auditing

In this section, we will define the Grammar Oriented Filter Auditing (GOFA) problem and present a learning based algorithm for solving the problem. Intuitevely, the GOFA problem asks one to assess the robustness of a sanitizer or filter with respect to a context free grammar G which contains a set of attack strings.

Definition 22. In the Grammar Oriented Filter Auditing (GOFA) problem the input is a context free grammar G and query access to an unknown function $\mathbf{f}$ such that:

1. If $\mathbf{f} : \Sigma^* \to \Sigma^*$ is a sanitizer function then, the GOFA problem asks to find $s \in G$ such that there exists an input $u \in \Sigma^*$ such that $\mathbf{f}(u) = s$.

2. If $\mathbf{f} : \Sigma^* \to \mathbb{B}$ is a filter function then, the GOFA problem asks to find $u \in G$ such that $\mathbf{f}(u) = \mathbf{F}$.

One can easily prove that in the general case the GOFA problem requires an exponential number of queries. Simply consider the CFG $\mathcal{L}(G) = \Sigma^*$ and a DFA F such that $\mathcal{L}(F) = \Sigma^* \setminus \{\text{random-large-string}\}$. Then, the problem reduces in guessing a random string which requires an exponential number of queries in the worst case. A formal proof of a similar result was presented by Peled et al. [70].

Our algorithm for the GOFA problem uses a learning algorithm for SFAs utilizing as an equivalence oracle the algorithm in Algorithm 9. The algorithm takes as input a hypothesis machine H. It then finds a string $s \in \mathcal{L}(G)$ such that $s \notin \mathcal{L}(H)$. If the string s is an attack against the target filter, the algorithm outputs the attack-string and terminates. If it is not it returns the string as a counterexample. On the other hand if there is no string bypassing the hypothesis, the algorithm terminates accepting the hypothesis automaton H. Note that, this is the point where we trade

completeness for efficiency since, even though $\mathcal{L}(G \cap \overline{H}) = \emptyset$, this does not imply that $\mathcal{L}(G \cap \overline{F}) = \emptyset$.

Algorithm 9 GOFA Algorithm

Require: Context Free Grammar G, membership oracle O

```
function EQUIVALENCE ORACLE(H)
    G_A ← G ∩ H̄
    if L(G_A) = ∅ then
        return Done
    else
        s ← L(G_A)
        if O(s) = True then
            return Counterexample, s
        else
            return Attack, s
```

Adaptation to sanitizers. The technique above can be generalized easily to sanitizers. Assume that we are given a grammar G as before and a target transducer T implementing a sanitization function. In this variant of the problem we would like to find a string s_A such that there exists $s \in \mathcal{L}(G)$ for which $s_A[T]s$ holds.

In order to determine whether such a string exists, we first construct a pushdown transducer T_G with the following property: A string s will reach a final state in T_G if and only if $s \in \mathcal{L}(G)$. Moreover every transition in T_G is the identity function, i.e. outputs the character consumed. Therefore, we have a transducer which will generate only the strings in $\mathcal{L}(G)$. Finally, given a hypothesis transducer H, we compute the pushdown transducer $H \circ T_G$ and check the resulting transducer for emptiness. If the transducer is not empty we can obtain a string s_A such that $s_A[H \circ T_G]s$. Since T_G will generate only strings from $\mathcal{L}(G)$ it follows that s_A when passed through the sanitizer will result in a string $s \in \mathcal{L}(G)$. Afterwards, the GOFA algorithm continues as in the DFA case.

Comparison With Random Testing. Regarding the usefulness of GOFA algorithm as a security auditing method it is important to consider it in comparison to random testing/fuzzing. Currently, most tools in the black-box testing domain, such as web vulnerability scanners, work by fuzzing the target filter with various attack

strings until a bypass is found or the set of attack strings is exhausted.

We argue that our GOFA algorithm is superior to fuzzing for two reasons:

1. *The number of queries of the GOFA algorithm is independent of the size of the grammar.* On the other hand, when producing random strings from a grammar in order to test a filter a very large number of strings has to be produced. Moreover, testing for modern vulnerabilities such as XSS is very complex, since there is a large number of variations that one should consider(cf. [8]).

2. *Random testing produces no information on the structure of the filter if no attack is found.* Consider the case where one produces a large number of candidate attack strings, but no bypass is found. Then, the auditor is left with no additional information for the filter, other than it rejected the set of strings that was tested. One approach would be to try to infer the structure of an automaton from that set of strings. Unfortunately, inferring the minimal automaton which is consistent with a set of strings is NP-Hard to approximate even within any polynomial factor [72]. On the other hand, as we demonstrate our GOFA algorithm is able to recover on average 90% of the states of the target filter in cases where no attack exists and an expressive enough grammar is given as input.

9.3.1 Approximating a Complete Equivalence Oracle

Although the GOFA algorithm is a suitable equivalence oracle implementation in the case the goal is to audit a target filter, in some cases one would like to recover a complete model of the target filter/sanitizer. In such cases, finding a bypass is not enough. Since we only assume black-box access to the target filter, in order for this problem to be even solvable we have to assume an upper bound on the size of the target filter. In this case, The Vasilevskii-Chow(VC) algorithm [27] exists for checking compliance between a DFA and a target automaton given black-box access to the second.

However, if the DFA at hand has n states and the upper bound given is m then the VC algorithm is exponential in $m - n$. Moreover, the algorithm suffers from the

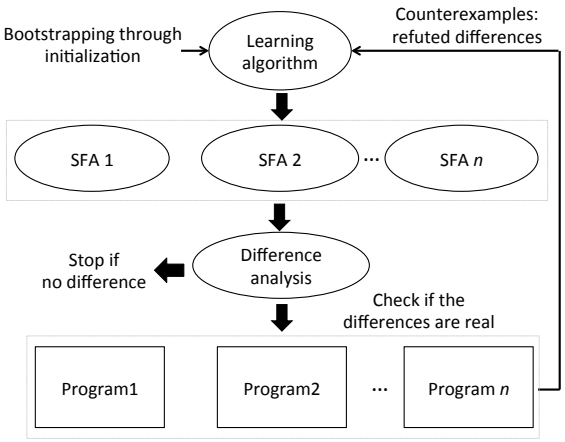

Figure 9-1: SFADiff archtitecture

same limitations in the alphabet size as DFA learning algorithms since every possible transition of the black-box automaton must be checked. Creating a symbolic version of the VC algorithm may be possible however, we will again only get probabilistic guarantees on the correctness of our equivalence oracle.

Another option is to construct a context free grammar describing the input protocol under which the sanitizer should operate and then use random sampling from that grammar to test whether the hypothesis and the target programs are complying. For example, when we test HTML Encoders we might want to construct a grammar with a number of different character sequences such as encoded HTML entities or special characters and test the behavior of the encoder under these strings. We employ this approach in our experiments.Finally, static analysis techniques [85] can be used to generate a CFG describing the output of another implementation of the same sanitizer or filter and then cross check the generated CFG with the target sanitizer using our GOFA algorithm.

9.4 Differential Testing with s-FAs

9.4.1 Basic Algorithm

The main idea behind our differential testing algorithm is to leverage automata learning in order to infer SFA-based models for the test programs and then compare the resulting models for equivalence as shown in Figure 9-1. As mentioned above, this technique has a number of advantages such as being able to generalize from comparing individual input/output pairs and build models for the programs that are examined.

Algorithm 10 provides the basic algorithmic framework for differential testing using automata learning. The algorithm takes two program implementations as input. The first function calls, to the `GetInitialModel` function, are responsible for bootstrapping the models for the two programs. The initialized models are then checked for differences using the `RCADiff` function call. The internals of this function are described in detail in Section 9.4.2. This function is responsible for categorizing the differences in the two models and return a sample set of inputs covering all categories that can cause the two programs to produce different outputs. The algorithm stops if the two models are equivalent. Otherwise, `RCADiff` returns a set of inputs that cause the two SFA models to produce different output.

However, since these differences are obtained by comparing the program models and not the actual programs, they might contain false positives resulting from inaccurate models. To detect such cases, we verify all differences obtained from the `RCADiff` call using the actual test programs. If any input is found not to produce a difference in the implementations, then that input is used as a counterexample in order to refine the model through the `UpdateModel` call. Finally, when a set of differences in the two models is verified to contain only true positives, the algorithm returns the set of corresponding inputs back to the user.

The astute reader may notice that, if no candidate differences are found between the two models, the algorithm terminates. For this reason, model initialization plays a significant role in our algorithm, since the initialized models should be expressive enough in order to provide candidate differences. It is interesting to point out that

Algorithm 10 Differential SFA Testing Algorithm

```
Require: P_1, P_2 are two programs
  function GetDifferences(P_1, P_2)
      M_1 ← GetInitialModel(P_1)
      M_2 ← GetInitialModel(P_2)
      while true do
          S ← RCADiff(M_1, M_2)
          if S = ∅ then
              return ∅
          modelUpdated ← False
          for s ∈ S do
              if P_1(s) ≠ M_1(s) then
                  M_1 ← UpdateModel(M_1, s)
                  modelUpdated ← True
              if P_2(s) ≠ M_2(s) then
                  M_2 ← UpdateModel(M_2, s)
                  modelUpdated ← True
          if modelUpdated = False then
              return S
```

the candidate differences do not have to be real differences.

9.4.2 Difference Analysis

Assume that we found and verified a number of inputs that cause the two programs under test to produce different outputs. One fundamental question is whether we can classify these inputs in certain equivalence classes based on the cause of the deviant behavior. We will now describe how we can use the inferred SFAs in order to compute such a classification. Ideally, we would like to assign in two inputs that cause a difference the same root cause if they follow the same execution paths in the target programs. Since the program source is unavailable, we trace the execution path of the inputs in the respective SFA models.

RCADiff algorithm. Given two SFAs M_1 and M_2, it is straightforward to compute their intersection by adapting the classic DFA intersection algorithm [78]. Let $M_{prod} = (Q_1 \times Q_2, (q_0, q_0), \{(q_i, q_j) : q_i \in F_1 \wedge q_j \in F_2\}, \mathcal{P}, \Delta)$ be the, minimal, product automaton of M_1, M_2. Notice initially, that the reason a difference is observed in the output after processing an input in both SFAs is that the labels of the states reached

in the two machines are different. This motivates our definition of *points of exposure*.

Definition 23. Let M_{prod} be the intersection SFA of M_1, M_2 as defined above. We define the set $\{(q_i, q_j) | (q_i, q_j) \in Q_{prod} \wedge q_i \in Q_1 \wedge q_j \in Q_2 \wedge l(q_1) \neq l(q_2)\}$ to be the *points of exposure* for the differences between M_1, M_2.

Intuitively, the points of exposure are the reasons the differences in the programs are observed through the output of programs. The path to a point of exposure encodes two different execution paths in machines M_1 and M_2 respectively which, under the same input, end up in states producing different output. Thus, we say that any simple path to a point of exposure is a *root cause* of a difference.

Definition 24. Let M_1, M_2 be two SFAs and M_{prod} be the intersection of M_1, M_2. Let $Q_p \subseteq Q_{prod}$ be the points of exposure for M_{prod}. We say that the set of simple paths $S = \{q_0 \xrightarrow{*} q_p | q_p \in Q_p\}$ is the set of *root causes* for the differences between M_1 and M_2.

Equipped with the set of paths our classification algorithm works as follows: Given two inputs causing a difference, we first reduce the path followed by each input into a simple path, i.e. we remove all loops from the path. For example, an input following the path $q_0 \rightarrow q_4 \rightarrow q_5 \rightarrow q_4 \rightarrow q_{10}$ will be reduced to the path $q_0 \rightarrow q_4 \rightarrow q_{10}$. Afterwards, we classify the two inputs in the same root cause if the simple paths followed by the inputs are the same.

Algorithm 11 shows the pseudocode for the `RCADiff` algorithm. The algorithm works by collecting all the distinct root causes from the product automaton using the the `SimplePaths` function call. This function accepts an SFA and a target state and returns all simple paths from the initial state to the target state using a BFS search. Afterwards, each path is converted into a sample input through the function `Path2Input`. This function works by selecting, for each edge $q_i \rightarrow q_j$ in the path, a symbol $\alpha \in \Sigma$ such that $(q_i, \phi, q_j) \in \Delta \wedge \phi(\alpha) = 1$. Finally, these symbols are concatenated in order to form an input that exercise the given path in the SFA.

Algorithm 11 Difference Categorization Algorithm

Require: M_1, M_2 are two SFA Models

function RCADIFF(M_1, M_2)

$M_{prod} \leftarrow$ ProductSFA(M_1, M_2)

$S \leftarrow \emptyset$

for $(q_i, q_j) \in Q_{prod} \mid l(q_i) \neq l(q_j)$ **do**

$S \leftarrow S \cup$ SimplePaths($M_{prod}, (q_i, q_j)$)

return Path2Input(S)

9.4.3 Differentiating Program Sets

In this section, we describe how our original differential testing framework can be generalized into a `GetSetDifferences` algorithm which works as follows: Instead of getting two programs as input, the `GetSetDifferences` algorithm receives two sets of programs $\mathcal{I}_1 = \{P_1, \ldots, P_n\}$ and $\mathcal{I}_2 = \{P_1, \ldots, P_m\}$. Assume that the output of each program is a bit $b \in \{0, 1\}$. The goal of the algorithm is to find a set of inputs S such that, the following condition holds:

$$\exists b \ \forall P_1 \in \mathcal{I}_1, P_1(s) = b \wedge \forall P_2 \in \mathcal{I}_2, P_2(s) = 1 - b$$

While conceptually simple, this extension provides a number of nice applications. For example, consider the problem of finding differences between the HTML/JavaScript parsers of browsers and those of WAFs. While finding such differences between a single browser and a WAF will provide us with an evasion attack against the WAF, the `GetSetDifferences` algorithm allows us to answer more sophisticated questions such as: (i) Is there an evasion attack that will bypass multiple different WAFs? and (ii) Is there an evasion attack that will work across different browsers? Also, as we describe in Section 9.4.4, this extension allows us to produce succinct fingerprints for distinguishing between multiple similar programs.

GetSetDifferences Algorithm. We extend our basic `GetDifferences` algorithm as follows: First, instead of initializing two program models as before, we initialize the SFA models for all programs in both sets accordingly. Similarly, when we verify the candidate differences obtained from the inferred models, all programs in both

sets should be checked. Besides these changes, the skeleton of the GetDifferences algorithm remains the same.

The most crucial and time-consuming part of our extension is the extension to the `RCADiff` functionality in order to detect differences between two sets of models. Recall that RCADiff utilizes the product construction and then finds the simple paths leading to the points of exposure. Given two sets of models, we compute the intersection between all the models in the two sets. Afterwards, we set the points of exposure as follows. Let $q = (q_0, \ldots, q_{m+n})$ be a state in the product automaton. Furthermore, assume that state q_i corresponds to automata M_i from one of the input sets $\mathcal{I}_1, \mathcal{I}_2$. Then, q is a point of exposure if

$$\forall M_i \in \mathcal{I}_1, M_j \in \mathcal{I}_2 \implies l(q_i) \neq l(q_j)$$

With this new definition of the points of exposure, the modified `RCADiff` algorithm proceeds as in the original case to find all simple paths in the product automaton that lead to the points of exposure.

One potential downside of this algorithm is that, its complexity increases exponentially as we add more models in the sets. For example, computing the intersection of m DFA with n states each, requires time $O(n^m)$ while, in general, the problem is PSPACE-complete [56]. That being said, we stress that the number of programs we have to check in practice will likely be small and many additional heuristics can be used to reduce the complexity of the intersection computation.

9.4.4 Program Fingerprints

Formally, the fingerprinting problem can be described as follows: given a set $\mathcal{I}$ of m different programs and black-box access to a server T which runs a program $P_T \in \mathcal{I}$, how can one find out which program is running in the server T by simply querying the program in a black-box manner, i.e. find $P \in \mathcal{I}$ such that $P = P_T$.

In this section, we present two different fingerprinting algorithms that provide different trade-offs between computational and query complexity. Both these algorithms

Algorithm 12 Fingerprint Tree Building Algorithm

Require: $\mathcal{I}$ is a set of Programs

```
function BuildFingerprintTree(I)
    if |I| = 1 then
        root.data ← P ∈ I
        return root
    P_i, P_j ← I
    s ← GetDifferences(P_i, P_j)
    root.data ← s
    root.left ← BuildFingerprintTree(I \ P_i)
    root.right ← BuildFingerprintTree(I \ P_j)
    return root
```

build a binary tree called fingerprint tree that stores strings that can distinguish between any two programs in $\mathcal{I}$. Given a fingerprinting tree, our first algorithm requires $|\mathcal{I}|$ queries to the target program. If the user is willing to perform extra off-line computation, our second algorithm demonstrates how the number of queries can be brought down to $\log m$.

Basic fingerprinting algorithm. The `BuildFingerprintTree` algorithm (shown in Algorithm 12) constructs a binary tree that we call a *fingerprint tree* where each internal node is labeled by a string and each leaf by a program identifier. In order to build the fingerprint tree recursively, we start with the set of all programs $\mathcal{I}$, choose any two arbitrary programs P_i, P_j from $\mathcal{I}$, and use the differential testing framework to find differences between these programs. We label the current node with the differences, remove P_i and P_j from $\mathcal{I}$, and call `BuildFingerprintTree` recursively until a single program is left in $\mathcal{I}$. If $\mathcal{I}$ has only one program, we label the leaf node with the program and return.

Given a fingerprint tree, we solve the fingerprinting problem as follows: Initially, we start at the root node and query the target program with a string from the set that labels the root node of the tree. If the string is accepted (resp. rejected), we recursively repeat the process along the left subtree (resp. right subtree), until we reach a leaf node that identifies the target program.

Time/query complexity. For the following we assume an input set of programs

$\mathcal{I}$ of size $|\mathcal{I}| = m$. Our algorithm has to find differences between all $\binom{m}{2}$ different program pairs. The fingerprint tree resulting from the algorithm will be a full binary of height m. Assuming that the complexity of the differential testing algorithm is D, we get that the overall time complexity of the algorithm is $O(2^{m-1} + \binom{m}{2}D)$. The query complexity of the algorithm is $|\mathcal{I}|$-1 queries, since each query will discard one candidate program from the list.

Reducing queries using shallow fingerprint trees. Notice that, in the previous algorithm, we need m queries to the target program in order to find the correct program because we discard only one program at each step. We can cut down the number of queries by shallower fingerprint trees at the cost of higher off-line computational complexity for building such trees.

Consider the following modification in the `BuildFingerprintTree` algorithm: First, we partition $\mathcal{I}$ into k subsets $\mathcal{I}_1, \ldots \mathcal{I}_k$ of size m/k each. Next, we call `BuildFingerprintTree` algorithm with the set $\mathcal{I}_S = \{\mathcal{I}_1, \ldots, \mathcal{I}_k\}$ as input programs and replace the call to `GetDifferences` with `GetSetDifferences`. This algorithm will generate a full binary tree of height k that can distinguish between the programs in the different subsets of $\mathcal{I}$. We can recursively apply the same algorithm on each of the leafs of the resulting fingerprinting tree, further splitting the subsets of $\mathcal{I}$ until each leaf contains a single program.

Time/query complexity. It is evident that the algorithm will eventually terminate since each subset is successively portioned into smaller sets. Let us assume that $D_{set}(k)$ the complexity of the `GetSetDifferences` algorithm when the input program sets are of size k (see section 9.4.3 for a complexity analysis of $D_{set}(k)$). The number of queries required for fingerprinting an application with this algorithm will be equal to the height of the resulting fingerprint tree. Note that each subset is of size m/k and to distinguish between the k subsets using our basic algorithm we need $k-1$ queries. Therefore we get the equation $T(m) = T(m/k) + (k-1)$ describing the query complexity of the algorithm. Solving the equation we get that $T(m) = (k-1)\log_k m$ which is the query complexity for a given k. When $k = 2$ we will need $\log m$ queries to identify the target program. Since each program provides one bit of information

per query (accept/reject), a straightforward decision tree argument [**?**] provides a matching lower bound on the query complexity of the problem.

Regarding the time complexity of the problem, we notice that, at the i-th recursive call to the modified `BuildFingerprintTree` algorithm, we will have an input set of size m/k^i since the initial set is repeatedly partitioned into k subsets. the overall time complexity of building the tree is $\sum_{i=1}^{\log_k m}(2^{m/k^i} + \binom{m/k^i}{2} D_{set}(m/k^i))$. We omit further details here as the complexity analysis is a straightforward adaptation of the original analysis.

Chapter 10

Evaluation

10.1 Transducer Learning Algorithms Evaluation

10.1.1 Benchmarks

In our evaluation, we focused on real-life benchmarks from web applications and specifically on string transformations which are used in order to protect web-applications from code injections attacks such as Cross-Site Scripting (XSS) attacks. XSS is one of the most important threats for modern web applications according to the recently released OWASP Top Ten 2017 [69]. We will now discuss the set of benchmarks used in our experiments. The interested reader can find a detailed description of each string transformation in the appendix.

- **Encoders/Decoders:** We include certain basic encoders and decoders used extensively in web applications such as HTML encoders/decoders and encoders for quote symbols.
- **Browser Filters:** We also include filters from The Internet Explorer (IE) browser which are designed to modify suspicious parts of the input in order to prevent XSS attacks.
- **Browser innerHTML Mutations:** Mutation XSS [46], is a type of XSS vulnerability that occurs due to the fact that the browser is internally transform-

ing a non-malicious string into a malicious one by applying a number of string transformations internally in the user's browser after the input is cleaned by the application. This may result in converting safe inputs back into malicious ones. Recent work [59] posed the open problem of modeling these transformations as transducers in order to allow a formal analysis. We collected 5 of these transformations from the literature [45] and demonstrate that these transformations can be effectively modelled as SVND transducers.

– **String Sanitizers:** Finally, we include a number of string transformations found in popular XSS sanitization frameworks which are used by popular applications in order to prevent XSS attacks. We include filters from Codeigniter [9], Kses, which is used by Wordpress [10] and the SysPass [11] password manager.

Research questions. The goal of the evaluation of the paper is to provide answers to the following research questions:

Q1: Can the SVND class of transducers capture real life string transformations from our benchmark set and are those transformations efficiently learnable?

Q2: What is the effect of the size of the alphabet on the efficiency of the algorithm.

Q3: Can the SVND learning algorithm be used for the evaluation of XSS sanitizers?

10.1.2 Evaluation of SVND transducer learning

In order to evaluate our SVND learning algorithm we used an alphabet of 74 symbols containing most printable ASCII characters. We chose this alphabet because it was large enough to cover all the character utilized by our benchmarks while at the same time keeping the performance of the algorithm in a reasonable level. In order to implement an equivalence oracle, we used a manually created set of test cases designed to cover all different behaviors of each benchmark and manually inspected each learned model. Implementing, or approximating, a correct equivalence oracle is an important aspect in order to utilize query learning algorithms in practice. We discuss available choices in the related work section. Finally, we implemented two basic

	Benchmark	States	Lookahead	Total Queries	Cached Queries	Equivalence	Cached Equivalence
Encoders	1. E-1	2	0	105	452	1	0
	2. E-2	2	0	126	683	1	0
	3. E-3	24	6	11519	35201	4	9
Browser Filters	4. B-1	25	∞	14023	39812	2	10
	5. B-2	13	∞	5736	15032	2	6
	6. B-3	17	∞	7168	21425	2	7
	7. B-4	17	∞	7524	21538	2	7
	8. B-5	29	∞	20425	56251	2	13
	9. B-6	17	∞	7498	21325	2	7
	10. B-7	13	∞	3092	11399	2	4
	11. B-8	34	∞	23428	46150	3	10
	12. B-9	23	∞	1Z289	31541	3	7
	13. B-10	15	9	6650	18275	2	7
	14. B-11	9	∞	2858	7209	1	4
	15. B-12	13	∞	4461	13954	2	5
	16. B-13	23	∞	10273	30499	2	7
Mutations	17. M-1	6	∞	1101	3471	2	1
	18. M-2	9	∞	2128	7126	2	3
	19. M-3	28	∞	17951	50225	5	10
	20. M-4	29	∞	17785	52499	5	10
	21. M-5	12	∞	4820	13810	2	5
Sanitizers	22. S-1	20	∞	6703	20820	3	5
	23. S-2	8	∞	1742	5823	2	2
	24. S-3	12	∞	2205	8531	2	2
	25. S-4	12	∞	2951	9951	2	3
	26. S-5	11	∞	4521	12421	2	5
	27. S-6	44	∞	37852	104521	6	15
	28. S-7	4	∞	561	2521	2	0

Table 10.1: Performance of SVND learning algorithm.

query reduction optimizations: Firstly, whenever an output query is asked by the algorithm we cache the result for future usage. Second, we implement a similar caching optimization for equivalence queries: Whenever an equivalence query is performed we check that the supplied model is behaving correctly on the last counterexample, otherwise we resupply the same counterexample.

Overall results. Table 10.1 presents the results of running our SVND learning algorithm in the set of experiments. We found that that the filters, despite being simple in their description they are non-trivial in terms of number states, ranging from 2 to 44 states. Notice that, caching the membership queries resulted in reusing a large part of the output queries previously made by the algorithm. Finally, we notice that most benchmarks have unbounded lookahead, a fact suggests that non-deterministic models are necessary in order to model real-life programs as transducers, at least in the domain of string sanitizer programs.

Equivalence queries and counterexamples. When examining the number of equivalence queries made by the learning algorithm we notice that despite the fact that the algorithm may perform up to $|Q|$ equivalence queries the number of actual equivalence queries performed was usually a small fraction of $|Q|$ and many times, a single equivalence query was made. We attribute this behavior in two facts: First, many counterexamples were generated by resupplying the same counterexample as before. This behavior is expected in L^* style algorithms since each addition of a distin-

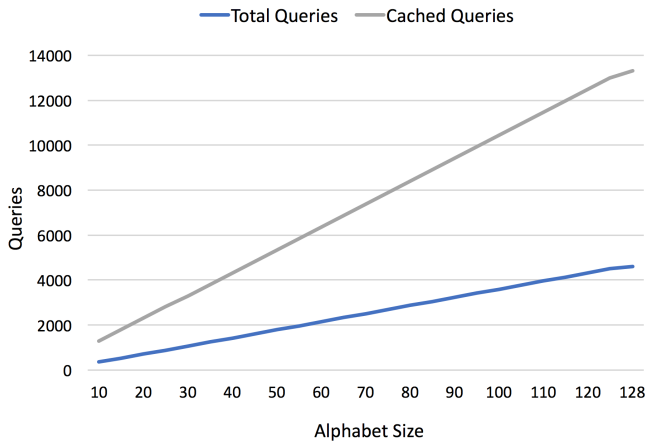

Figure 10-1: Total number of output queries made by the learning algorithm for different alphabet sizes when learning the IE Anti-XSS Form filter (no. 14).

guishing condition will usually discover one additional state and therefore, discovering all undiscovered states which were accessed by the counterexample may require many iterations. Moreover, a large number of counterexamples in each benchmark was detected by checking the models for ambiguity.

Effect of the alphabet size on performance. As we mention in theorem 8, the number of output queries performed by the algorithm is correlated linearly with the size of the alphabet. In figure 10-1 we verify this relation by examining the number of output queries required in order to learn a correct model of the IE Anti-XSS Form (no. 14) filter using alphabets of different sizes. Notice, that even though the correlation is indeed linear, the actual number of queries tends to increase significantly (in concrete numbers) as the alphabet grows. Indeed, learning the filter using a small alphabet of 10 symbols requires just 350 queries while learning over an alphabet of 128 symbols requires more 4500 queries. To mitigate this problem, we are currently working on adapting our algorithms on the symbolic setting where independent synthesis algorithms are used in order to infer the set of output labels from a few examples [17, 36].

10.1.3 Black-box testing of sanitizer robustness

Learning algorithms can be used to develop testing frameworks. In the context of testing string sanitizers, one such testing method is the GOFA algorithm [17]. In a nutshell, the algorithm accepts a set of attack vectors in the form of a context-free grammar (CFG) G, a learning algorithm for transducers, and checks, in a black-box manner, whether any string from G can bypass a target string sanitizer. The interested reader can find more details on the algorithm in the appendix.

Case study. We will now demonstrate that using our SVND transducer learning algorithm, the GOFA algorithm can uncover sophisticated attacks against sanitizers. The following example is taken from a vulnerability found in an older version of the Taskfreak [12] task management application which is affected by an XSS vulnerability. The taskfreak application used, among others, the following string transformation in order to remove malicious part of the user input:

$$\text{“<script[^>]*>[^<]+</script[^>]*>”} \rightarrow \epsilon$$

Using the GOFA algorithm we evaluated whether it is possible to bypass the filtering and insert a script tag into the response of the application. To do so, we generated a grammar containing the string `<script>s()</script>`. Afterwards, we ran the GOFA algorithm using our VND transducer learning algorithm.

Results. The GOFA algorithm quickly converged to a vulnerability after our learning algorithm initially inferred a model computing the identity function for which a candidate attack was produced. The counterexample uncovered the non-deterministic path that removed suspicious input and after 33 states were added to the model, the algorithm uncovered the following attack:

“<sc<script>s</script>ript>s()</script>”

The reason this attack works is the fact the sanitization routine is not applied recursively and therefore, once the inner malicious payload is removed another malicious payload is created. The algorithm converged to this vulnerability after making about 2000 output queries and 1 equivalence query (without counting cached queries). The

alphabet used in this experiment was restricted on the characters used in our attack vector. Notice that the discovered attack is non-trivial: The attack requires to break a valid attack vector using another valid attack vector in a specific index. Discovering this attack using random testing or grammar-based testing is unlikely because such strings have a very small probability of occurring randomly even if sampling from a grammar.

10.2 MAT^* Evaluation

We have implemented MAT^* in the open-source `symbolicautomata` library [3], as well as the learning algorithms for boolean algebras over finite domains, equality algebras and BDD algebras as discussed in Section 8.5. Our implementation is fully modular: Once an algebra learning algorithm is defined in our library, it can be seamlessly plugged in as a guard learning algorithm for s-FAs. Since MAT^* is also an algebra learning algorithm, this allows us to easily learn automata over automata. All experiments were ran in a Macbook air with an 1.8 GHz Intel Core i5 and 8 GiB of memory. The goal of our evaluation is to answer the following research questions:

Q1: How does MAT^* perform on automata over large finite alphabets? (§ 10.2.1)

Q2: How does MAT^* perform on automata over algebras that require both membership and equivalence queries? (§ 10.2.2)

Q3: How does the size of predicates affect the performance of MAT^*? (§ 10.2.3)

10.2.1 Equality Algebra Learning

In this experiment, we use MAT^* to learn s-FAs obtained from 15 regular expressions drawn from 3 domains: (1) Regular expressions used in web application sanitization frameworks such as in the CodeIgniter framework, (2) Regular expressions drawn from popular web application firewall ModSecurity and finally (3) Regular expressions from [58]. For this set of experiments we utilize as alphabet the entire UTF-16 (2^{16}

Table 10.2: Evaluation of MAT^* on regular expressions.

ID	$\|Q\|$	$\|\Delta\|$	Memb	Equiv	R-CE	GU	D-CE	C-CE
RE.1	11	35	653	17	19	25	106	78
RE.2	24	113	7203	66	45	87	565	479
RE.3	11	15	483	11	16	16	59	45
RE.4	18	40	1745	17	33	32	188	164
RE.5	25	55	3180	22	48	45	244	211
RE.6	52	155	43737	588	104	640	3102	2953
RE.7	179	658	66477	1486	91	1398	7748	6540
RE.8	115	175	929261	299	206	390	28606	28354
RE.9	144	369	844213	699	261	817	30485	30135
RE.10	175	551	3228102	5346	286	5457	172180	170483
RE.11	6	9	3409	281	14	289	723	710
RE.12	10	14	1367	88	8	86	314	291
RE.13	29	46	20903	743	49	764	2637	2550
RE.14	8	13	5949	365	24	381	854	836
RE.15	8	15	661	82	2	76	228	198

characters) and used the equality algebra to represent predicates. Since the alphabet is finite, we also tried learning the same automata using TTT [51], the most efficient algorithm for learning finite automata over finite alphabets.

Results Table 10.2 presents the results of MAT^*. The **Memb** and **Equiv** columns present the number of distinct membership and equivalence queries respectively. The **R-CE** column shows how many times a counterexample was reused, while the **GU** column shows the number of counterexamples that were used to update an underlying predicate (as opposed to adding a new state in the s-FA). Finally, **D-CE** shows the number of counterexamples provided to an underlying algebra learner due to failed determinism checks, while **C-CE** shows the number of counterexamples due to failed completeness checks. Note that these counterexamples did not require invoking the equivalence oracle.

Given the large alphabet sizes, TTT runs out of memory on all our benchmarks. This is not surprising since the number of queries required by TTT just to construct the *correct* model for a DFA with $128 = 2^7$ states is at least $|\Sigma||Q| \log |Q| = 2^{16} * 2^7 * 7 \approx 2^{26}$. We point out that a corresponding lower bound of $\Omega(|Q| \log |Q||\Sigma|)$ exists for the number of queries any DFA algorithm may perform and therefore, the size of

the alphabet provides a fundamental limitation for any such algorithm.

Analysis. First, we observe that the performance of the algorithm is not always monotone in the number of states or transitions of the s-FA. For example, RE.10 requires more than 10x more membership and equivalence queries than RE.7 despite the fact that both the number of states and transitions of RE.10 are smaller. In this case, RE.10 has fewer transitions, but they contain predicates that are harder to learn—e.g., large character classes. Second, the completeness check and the corresponding counterexamples are not only useful to ensure that the generated guards form a partition but also to restore predicates after new states are discovered. Recall that, once we discover (split) a new state, a number of learning instances is discarded. Usually, the newly created learning instances will simply output $\bot$ as the initial hypothesis. At this point, completeness counterexamples are used to update the newly created hypothesis accordingly and thus save the MAT^* from having to rerun a large number of equivalence queries. Finally, we point out that the equality algebra learner made no special assumptions on the structure of the predicates such as recognizing character classes which are used in regular expressions and others. We expect that providing such heuristics can greatly improve the performance MAT^* in these benchmarks.

10.2.2 BDD Algebra Learning

In this experiment, we use MAT^* to learn s-FAs over a BDD algebra. We run MAT^* on 1,500 automata obtained by transforming Linear Temporal Logic over finite traces into s-FAs [30]. The formulas have 4 atomic propositions and the height in each BDD used by the s-FAs is four. To learn the underlying BDDs we use MAT^* with the learning algorithm for algebras over finite domains (see section 8.5) since ordered BDDs can be seen as s-FAs over $\boldsymbol{dom} = \{0, 1\}$.

Figure 10-2 shows the number of membership (top left) and equivalence (top right) queries performed by MAT^* for s-FAs with different number of states. For this s-FAs, MAT^* is highly efficient with respect to both the number of membership and equivalence queries, scaling linearly with the number of states. Moreover, we

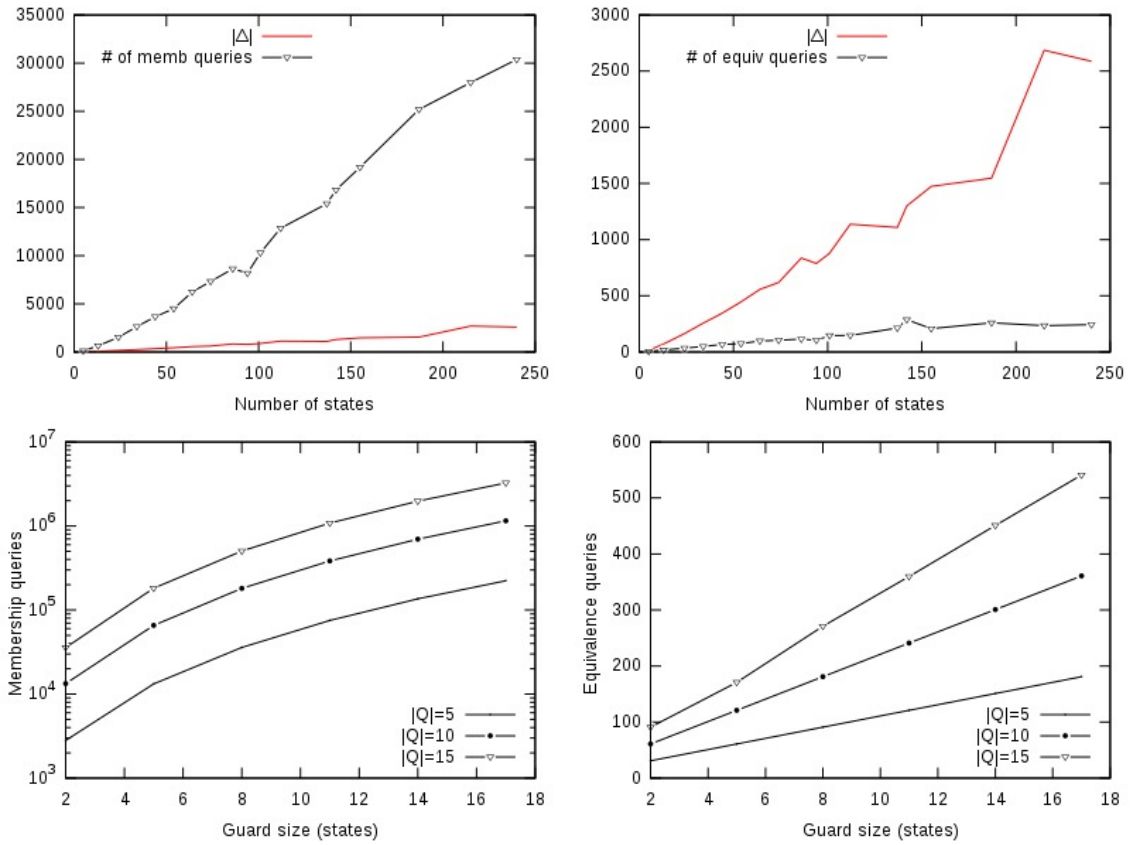

Figure 10-2: (Top) Evaluation of MAT^* on s-FAs over a BDD algebra. (Bottom) Evaluation of MAT^* on s-FAs over an s-FA algebra. For an s-FA $\mathcal{M}_{m,n}$, the x-axis denotes the values of n. Different lines correspond to different values of m.

note that the size of the set of transitions $|\Delta|$ does not drastically affect the overall performance of the algorithm. This is in agreement with the results presented in the previous section, where we argued that the difficulty of the underlying predicates and not their number is the primary factor affecting performance.

10.2.3 s-FA Algebra Learning

In this experiment, we use MAT^* to learn 18 s-FAs over s-FAs, which accept strings of strings. We evaluate the scalability of our algorithms when the difficulty of learning the underlying predicates increases. The possible internal s-FAs, which we will use as predicates, operate over the equality algebra and are denoted as I_k (where $2 \leq k \leq 17$). Each s-FA I_k accepts exactly one word $a \cdots a$ of length k and has $k+1$ states and $2k+1$ transitions. The external s-FAs are denoted as $\mathcal{M}_{m,n}$ (where $m \in \{5, 10, 15\}$

and $2 \leq n \leq 17$). Each s-FA $\mathcal{M}_{m,n}$ accepts exactly one word $s \cdots s$ of length m where each s is accepted by I_n.

Analysis. For simplicity, let's assume that we have the s-FA $\mathcal{M}_{n,n}$. Consider a membership query performed by one of the underlying algebra learning instances. Answering the membership query requires sifting a sequence in the classification tree of height at most n which requires $O(n)$ membership queries. Therefore, the number of membership queries required to learn each individual predicate is increased by a factor of $O(n)$. Moreover, for each equivalence query performed by an algebra learning instance, the s-FA learning algorithm has to pinpoint the counterexample to the specific algebra learning instance, a process which requires $\log m$ membership queries, where m is the length of the counterexample. Therefore, we conclude that each underlying guard with n states will require a number of membership queries which is of the order of $O(n^3)$ at the worst and $O(n^2 \log n)$ queries at the best (since the CT has height $\Omega(\log n)$), ignoring the queries required for counterexample processing.

Figure 10-2 shows the number of membership (bottom left) and equivalence (bottom right) queries, which verify the theoretical analysis presented in the previous paragraph. Indeed, we see that in terms of membership queries, we have a very sharp increase in the number of membership queries which is in fact about quadratic in the number of states in the underlying guards. On the other hand, equivalence queries are not affected so drastically, and only increase linearly.

10.3 GOFA Algorithm Evaluation

10.3.1 Implementation

We have implemented all the algorithms described in the previous sections. In order to evaluate our SFA learning algorithm in the standard membership/equivalence query model we implemented a complete equivalence oracle by computing the symmetric difference of each hypothesis automaton with the target filter. In order to evaluate regular expression filters we used the flex regular expression parser to generate a

IDS RULES			DFA LEARNING		SFA LEARNING		
ID	STATES	ARCS	MEMBER	EQUIV	MEMBER	EQUIV	SPEEDUP
1	7	13	4389	3	118	8	34.86
2	16	35	21720	3	763	24	27.60
3	25	33	56834	6	6200	208	8.87
4	33	38	102169	7	3499	45	28.83
5	52	155	193109	6	37020	818	5.10
6	60	113	250014	7	38821	732	6.32
7	66	82	378654	14	35057	435	10.67
8	70	99	445949	15	17133	115	25.86
9	86	123	665282	27	34393	249	19.21
10	115	175	1150938	31	113102	819	10.10
11	135	339	1077315	24	433177	4595	2.46
12	139	964	1670331	29	160488	959	10.35
13	146	380	1539764	28	157947	1069	9.68
14	164	191	2417741	29	118611	429	20.31
15	179	658	770237	14	80283	1408	9.43
						AVG=	15.31

Table 10.3: SFA vs. DFA Learning

DFA from the regular expressions and then parsed the code generated by flex to extract the automaton. In order to implement the GOFA algorithm we used the FAdo library [2] to convert a CFG into Chomsky Normal Form(CNF) and then we convert from CNF to a PDA. In order to compute the intersection we implemented the product construction for pushdown automata and then directly checked the emptiness of the resulting language, without converting the PDA back to CNF, using a dynamic programming algorithm [24].

10.3.2 Testbed

Since our focus is on security related applications, in order to evaluate our SFA learning and GOFA algorithms we looked for state-of-the-art regular expression filters used in security applications. We chose filters used by Mod-Security [5] and PHPIDS [7] web application firewalls (WAFs). These systems contain well designed and very complex regular expressions rule sets that attempt to protect against vulnerability classes

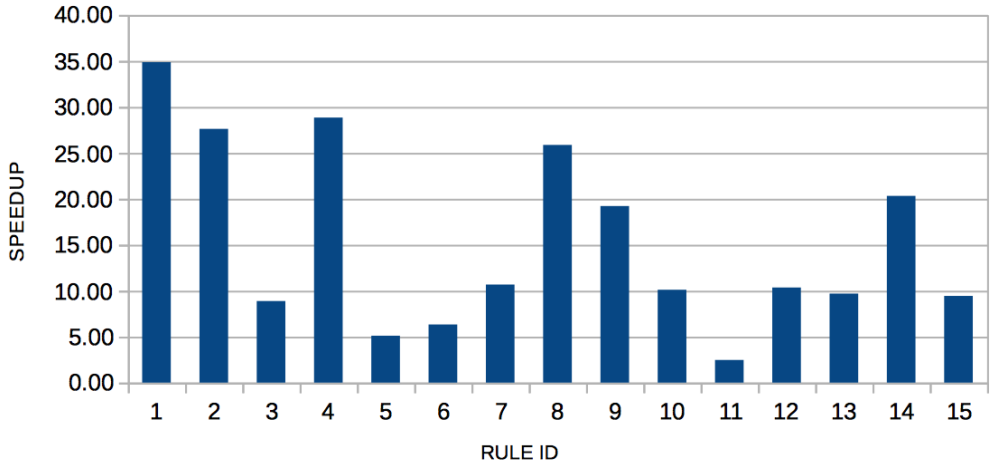

Figure 10-3: Speedup of SFA vs. DFA learning.

such as SQL Injection and XSS, while minimizing the number of false positives. For our evaluation we chose 15 different regular expression filters from both systems targeting XSS and SQL injection vulnerabilities. We chose the filters in a way that they will cover a number of different sizes when they are represented as DFAs. Indeed, our testbed contains filters with sizes ranging from 7 to 179 states. Using the identifiers from the figure, one can retrieve the rules from the source code of the systems. Our sanitizer testbed is described in detail in section 10.3.5.

For the evaluation of our SFA and DFA learning algorithms we used an alphabet of 92 ASCII characters. We believe that this is an alphabet size which is very reasonable for our domain. It contains all printable characters and in addition some non printable ones. Since many attacks contain Unicode characters we believe that alphabets will only tend to grow larger as the attack and defense technologies progress.

10.3.3 SFA Learning Algorithm Evaluation

We first evaluate the performance of our SFA learning algorithm using the L^* algorithm as the baseline. We implemented the algorithms as we described them in the paper using only an additional optimization both in the DFA and SFA case: we

	DFA LEARNING			SFA LEARNING			
ID	MEMBER	EQUIV	LEARNED	MEMBER	EQUIV	LEARNED	SPEEDUP
1	3203	2	100.00%	81	5	100.00%	37.27
2	18986	2	100.00%	521	11	100.00%	35.69
3	52373	5	100.00%	1119	7	96.00%	46.52
4	90335	5	96.97%	2155	10	96.97%	41.73
5	176539	4	98.08%	4301	38	80.77%	40.69
6	227162	5	96.67%	5959	32	96.67%	37.92
7	355458	12	98.48%	8103	17	98.48%	43.78
8	420829	13	98.57%	11013	34	98.57%	38.10
9	634518	25	98.84%	15221	30	98.84%	41.61
10	1110346	29	99.13%	27972	54	99.13%	39.62
11	944058	19	94.81%	100522	955	93.33%	9.30
12	1645751	28	100.00%	113714	662	96.40%	14.39
13	1482134	26	97.95%	45494	143	93.15%	32.48
14	1993469	24	90.85%	45973	32	90.85%	43.33
15	14586	5	8.94%	428	22	8.94%	32.42
		AVG=	91.95		**AVG=**	89.87%	35.66

Table 10.4: SFA vs. DFA Learning + GOFA

cached each query result both for membership and equivalence queries. Therefore, whenever we count a new query we verify that this query wasn't asked before. In the case of equivalence queries, we check that the automaton complies with all the previous counterexamples before issuing a new equivalence query.

In table 10.3 we present numerical results from our experiments that reveal a significant advantage for our SFA learning over DFA: it is approximately 15 times faster on the average. The speedup as the ratio between the DFA and the SFA number of queries is presented in Figure 10-3. An interesting observation here is that the speedup does not seem to be a simple function of the size of the automaton and it possibly depends on many aspects of the automaton. An important aspect is the size of the sink transition in each state of the SFA. Since our algorithm learns lazily the transitions, if the SFA incorporates many transitions with large size, then the speedup will be less than what it would be in SFAs were the sink transition is the only one with big size.

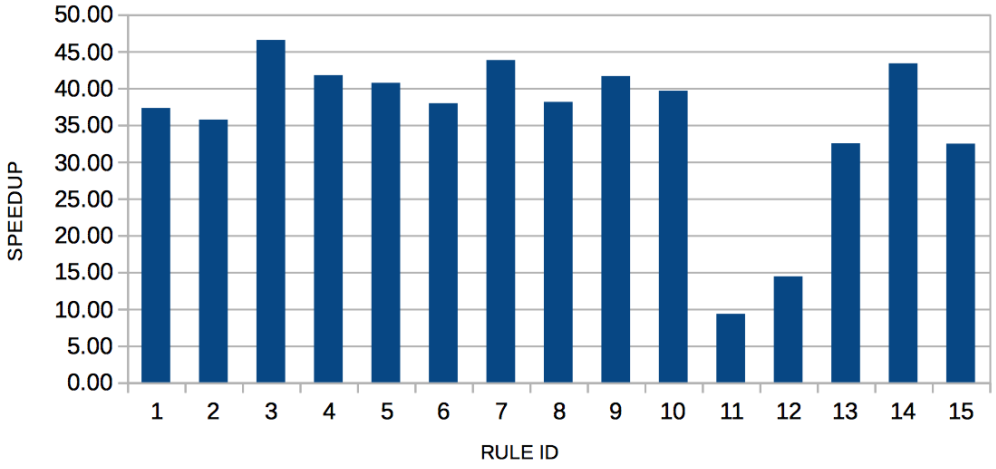

Figure 10-4: Speedup of SFA vs. DFA learning with GOFA.

Finally, we conducted we evaluated the overall speedup of the SFA algorithm in different alphabet sizes. Specifically, we found the minimal alphabet size such that all transitions of the filters tested are present; this requires an alphabet with 34 symbols. Then, we run the DFA and SFA algorithm with different alphabet sizes starting from 34 symbols up to our full alphabet of 92 symbols. The results of the experiments are shown in figure 10-5. We notice that the speedup is a monotonically increasing function as the alphabet size gets larger.

10.3.4 GOFA algorithm

In this section we evaluate the efficiency of our GOFA algorithm. In our evaluation we used both the DFA and the SFA algorithms. Since our SFA algorithm uses significantly more equivalence queries than the L^* algorithm, we need to evaluate whether this additional queries would influence the accuracy of the GOFA algorithm. Specifically, we would like to answer the following questions:

1. How good is the model inferred by the GOFA algorithm when no attack string exists in the input CFG?

2. Is the GOFA algorithm able to detect a vulnerability in the target filter if one

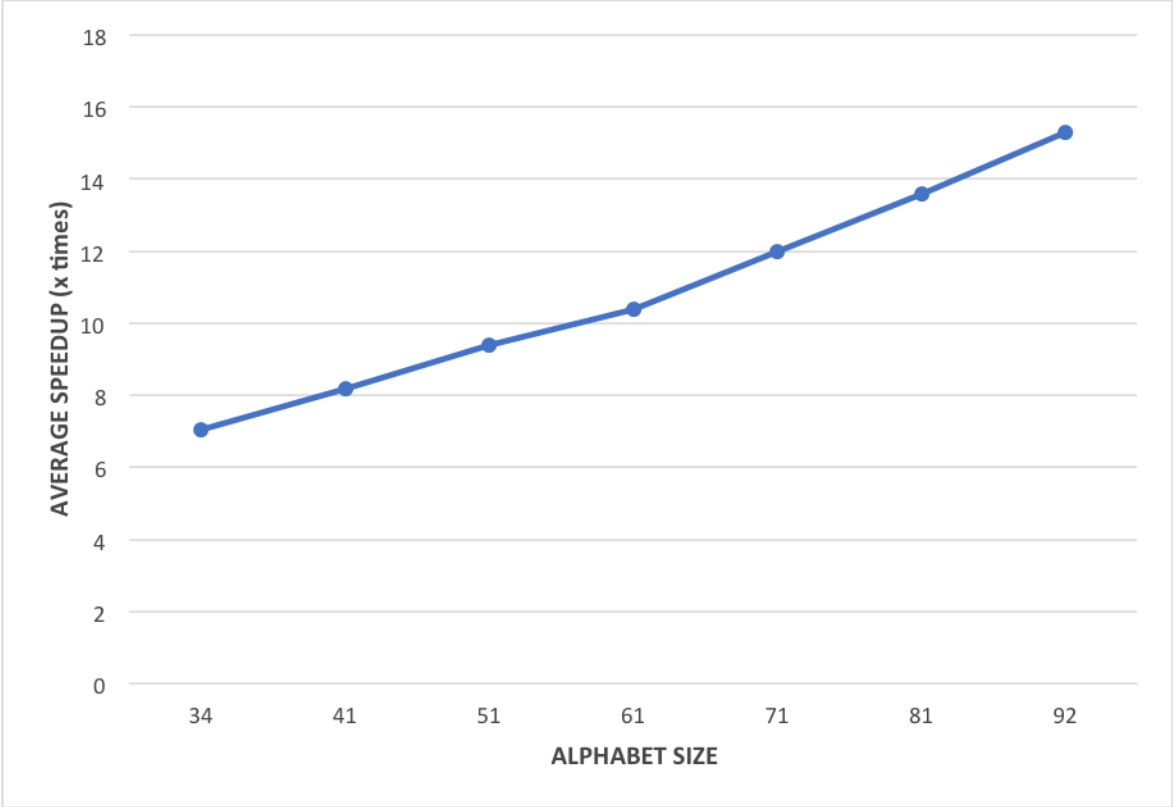

Figure 10-5: Speedup of SFA vs DFA algorithms for different alphabet sizes.

exists in the input CFG?

Making an objective evaluation on the effectiveness of the GOFA algorithm in these two questions is tricky due to the fact that the performance of the algorithm depends largely on the input grammar provided by the user. If the grammar is too expressive then a bypass will be trivially found. On the other hand if no bypass exists and moreover, the grammar represents a very small set of strings, then the algorithm is condemned to make a very inaccurate model of the target filter. Next, we tackle the problem of evaluating the two questions about the algorithm separately.

DFA model generation evaluation. Intuitively, the GOFA algorithm is efficient in recovering a model for the target filter if the algorithm is in possession of the necessary information in order to recover the filter in the input CFG and is able to do so. Therefore, in order to evaluate experimentally the accuracy of our algorithm in producing a correct model for the target filter independently of the choice of the grammar we used as input grammar the target filter itself. This choice is justified

as setting as input grammar the target filter itself we have that a grammar that, intuitively , is a maximal set without any vulnerability.

In table 10.4 we present the numerical results of our experiments over the same set of filters used in the experiments of Section 10.3.3. The learning percentage of both DFA and SFA with simulated equivalence oracle via GOFA is quite high (close to 90% for both cases). The performance benefit from our SFA learning is even more dramatic in this case reaching an average of ≈ 35 times faster than DFA. The speedup is also pictorially presented in Figure 10-4. We also point out the even though the DFA algorithm checks all transitions of the automaton explicitly (which is the main source of overhead), the loss in accuracy between the L^* algorithm and our SFA algorithm is only 2%, for a speedup gain of approximately x35.

Vulnerability detection evaluation. In evaluating the vulnerability detection capabilities of our GOFA algorithm we ran into the same problem as with the model generation evaluation; namely, the efficiency of the algorithm depends largely on the input grammar given by the user. If the grammar is much wider than the targeted filter then a bypass can be trivially found. On the other hand if it is too restrictive maybe no bypass will exist at all.

In our first experiment we test the GOFA algorithm using a regular grammar against the union of two rules targeting SQL Injection attacks from PHPIDS. Specifically, we start with a small grammar which contains the combination of some attack vectors and, whenever a vector bypassing the filter is found, we remove the vector from the grammar and rerun it with a smaller grammar until no attack is possible. Here we would like to find out whether the GOFA algorithm can operate under restricted grammars that require many updates on the hypothesis automaton. To check whether a vulnerability exists in the filter we computed the symmetric difference between the input grammar and the targeted filters. We note that this step is the reason we did not perform the same experiment on live WAF installations, since we do not have the full specification as a regular expression and thus cannot check if a bypass exists in an attack grammar.

We notice that in this case, GOFA was successful in updating the attack vectors

in order to generate new attacks bypassing the filter. However, in this case the GOFA algorithm generated as many as 61 states of the filter in the DFA case and 31 states in the SFA case until a successful attack vector was detected. Against we notice that the speedup of using the SFA algorithm is huge.

For our second experiment we used the GOFA algorithm against a live WAF installation. We utilized a handcrafted grammar containing valid suffixes for SQL statements. This grammar is efficiently modeling many different types of SQL injection attacks. For our target we use the latest version of Mod-Security, version 3.0.0. Table 10.6 presents the results from our experiment. We found many, previously unknown, SQL Injection attacks that could be used for evading Mod-Security while allowing the attacker to bypass authentication or retrieve data from the backend database. In order to find different attacks we use the same technique as in the previous experiment, i.e. we successively remove attacks found from the grammar. We notice that many vulnerabilities were found without discovering any state of the WAF. This fact suggests that state of the art WAFs are still not mature enough in order to defend even against popular vulnerability classes such as SQL injection. Furthermore, we notice that after detecting some vulnerabilities for which no rules exist, the algorithm discovered more complex vulnerabilities which required to iterate a few failed attack attempts from the input grammar.

To conclude with the evaluation of the GOFA algorithm, even though any GOFA algorithm is necessarily either incomplete or inefficient in the worst case, our algorithm performs well in practice detecting both vulnerabilities when they exist and inferring a large part of the targeted filter when it is not able to detect a vulnerability.

10.3.5 Cross Checking HTML Encoder implementations

To demonstrate the wide applicability of our sanitizer inference algorithms we reconsider the experiment performed in the original BEK paper [48]. The authors, payed a number of freelancer developers to develop HTML encoders. Then they took these HTML encoders, along with some other existing implementations and manually converted them to BEK programs. Then, using BEK the authors were able to find

	GRAMMAR		DFA LEARNING				SFA LEARNING				VULNERABILITY	
ID	STATES	ARCS	FOUND STATES	MEMB	EQUIV	FOUND STATES	MEMB	EQUIV	SPEEDUP	EXISTS	FOUND
1	128	175	61	155765	3	31	1856	8	83.56	TRUE	`union select load_file('0\0\0')`
2	111	146	61	155765	3	31	1811	7	85.68	TRUE	`union select 0 into outfile '0\0\0'`
3	92	120	61	155765	3	31	1793	6	86.58	TRUE	`union select case when (select user_name()) then 0 else 1 end`
4	43	54	61	155764	3	31	1770	7	87.65	FALSE	None
								AVG=	85.87		

Table 10.5: Attacks found by succesively reducing the attack grammar rules PHPIDS 76 & 52 composed

STATES	MEMB	EQUIV	VULNERABILITY
0	3	1	`a for update`
0	3	1	`a limit 1`
0	3	1	`a ; select a`
0	3	1	`a join a on a`
10	118	2	`a group by a desc`
0	3	1	`a procedure a ( a )`
0	3	1	`a and exists select a`
7	67	2	`a and a > any select a`
7	90	2	`a and a like 1`

Table 10.6: Vulnerabilities discovered using the GOFA algorithm on Mod-Security 3.0.0.

differences in the sanitizers and check properties such as idempotence.

Using our learning algorithms we are able to perform a similar experiment but this time completely automated and in fact, without any access to source code of the implementation. For our experiments we used 3 different encoders from the PHP language, the HTML encoder from the .net AntiXSS library [4] and then, we also inferred models for the HTML encoders used by Twitter, Facebook and Microsoft Outlook email service.

We used our transducer learning algorithms in order to infer models for each of the sanitizers which we then converted to BEK programs and checked for equivalence and idempotence using the BEK infrastructure. A function f is idempotent if $\forall x$, $f(x) = f(f(x))$ or in other words, reapplying the sanitizer to a string which was already

sanitized won't change the resulting string. This is a nice property for sanitizers because it means that we easily reapply sanitization without worrying about breaking the correct semantics of the string.

In our algorithm, we used a simple form of symbolic transducer learning where we generalized the most commonly seen output term to all alphabet members not explicitly checked. As an alphabet, we used a subset of characters including standard characters that should be encoded under the HTML standard and moreover, a set of other characters, including Unicode characters, to provide completeness against different implementations. For the simulation of the equivalence oracle we produced random strings from a predefined grammar including all the characters of the alphabet and in addition many encoded HTML character sequences. The last part is important for detecting if the encoder is idempotent.

Figure 10-6 shows the results of our experiment. We found that most sanitizers are different and only one sanitizer is idempotent. All the entries of the figure represent the character or string that the two sanitizers are different or a tick if they are equal. One exception is the entries labelled with u8249 which denotes the Unicode character with decimal representations ‹. We included the decimal representation in the table to avoid confusion with the "<" symbol. The idempotent sanitizer is a version of `htmlspecialcharacters` function with a special flag disabled, that instructs the function not to re-encode already encoded html entities. We would like to point out that although in general html encoders can be represented by single state transducers, making the encoder idempotent requires a large amount of lookahead symbols to detect whether the current character is part of an already encoded HTML entity. This adds a significant amount of complexity to the encoder which is a possible reason for not being used in practice.

Another surprising result is that the .net HTML encode function did not match the one in the MS Outlook email service. The encoder in the outlook email seems to match an older encoder of the AntiXSS library which was encoding all HTML entities in their decimal representations. For example, this encoder is the only one encoding the semicolon symbol. On the other hand the .net AntiXSS implementation

	PHP1	PHP2	PHP3	.NET	TW	FB	MS	Idempotent
PHP1	✓	u8249	&	u8429	✓	✓	;	✗
PHP2		✓	u8249	u8294	u8429	u8429	;	✗
PHP3			✓	&	&	&	;	✓
.NET				✓	u8429	u8429	;	✗
TW					✓	✓	;	✗
FB						✓	;	✗
MS							✓	✗

Figure 10-6: Equivalence Checking of HTML encoder implementations.

will encode Unicode characters in their decimal representations but will skip encoding the semicolon, as did every other sanitizer that we tested.

At this point, we would like to stress that our results are not conclusive. For example, the fact that we found that the Twitter and Facebook encoders are equal does not mean that there is no string in which these two functions differ. This is fundamental limitation of all black-box testing algorithms. In fact, even the results on differences between sanitizers might be incorrect in principle. However, in this case we can easily verify the differences and, if necessary, update the corresponding models for the encoders.

Black-box Cross Compilation: Once we have obtained models of our encoders and converted them to BEK programs, we can use the BEK backend to compile the generated models in a different language. For example, since twitter and facebook seem to use the `htmlspecialcharacters` fuction, we show in the appendix the corresponding BEK program inferred by our algorithm and the generated JavaScript code produce by BEK.

10.3.6 Bug in BEK HTML Decoder Example

While developing and debugging our implementation we found a bug in an example implementation of a simplified HTML decoder in the online BEK tutorial. The program in question is the program named **decode** from the second part of the BEK tutorial [1]. We won't present the whole program here due to space constraints, but the problem occurs in the following case:

```
case (s == 1) :                  //memorized &
  if (c == '&') { yield ('&'); }
  else if (c == 'l') { s := 2; }
  else if (c == 'g') { s := 3; }
  else { yield ('&',c); s := 0; }
```

Here as the comments suggests, the transducer has already processed the letter "&" and checks if any of the letter "l" or "t" follows which would complete the html entities "<" or ">". In the opposite case that no match with these two characters is found, the memorized symbol is being added to the output along with the current symbol. Unfortunately, if the new character is also part of an HTML entity, for example "&", then the program will fail to start scanning for the next symbols of the entity, rather it will just output the same character and return to initial state. Therefore, the program will fail to correctly decode sequences such as "&<".

We detected this bug during the development of our lookahead learning algorithm and our conversion algorithm to BEK programs. Specifically, we coded an HTML decoder like the decode BEK program and used the equivalence checking function of BEK in order to check whether the inferred BEK programs we were producing were correct. At some point, we detected the bug we described as a counterexample to the equivalence of the two implementations.

We believe that this bug demonstrates the complexity of writing sanitizers that make heavy use of lookahead transitions in BEK. One should implement a large number of nested if-then-else statements, like we describe in our conversion algorithm in section 10.3.5. We believe that the BEK language could become much simpler with the introduction of a string compare function to allow the programmers to easily handle lookaheads. This may require extra work on the backend of the BEK compiler, however we believe that this is a feasible task, that will greatly simplify the language.

IDS Rules	Without Init		With Init		Learned States	Init Filter States	States Diff	Member Overhead	Equiv Speedup
	Member	Equiv	Member	Equiv					
MODSEC 973323	2367	97	2400	2	25	25	0	1.01	48.50
MODSEC 973324	768	55	892	19	15	12	3	1.16	2.89
MODSEC 973330	887	62	941	21	15	12	3	1.06	2.95
PHPIDS 22	17195	252	17330	105	70	45	25	1.01	2.40
PHPIDS 27	144759	2618	149159	437	66	59	7	1.03	5.99
PHPIDS 40	11119	337	11152	68	35	25	10	1.00	4.96
PHPIDS 41	6635	318	8535	137	25	21	4	1.29	2.32
PHPIDS 50	6206	255	9829	1	25	27	-2	1.58	255.00
PHPIDS 56	38768	840	46732	7	60	62	-2	1.21	120.00
		Avg= 537.11×		Avg= 88.56×				Avg= 1.15×	Avg= 49.45×

Figure 10-7: The performance (no. of equivalence and membership queries) of the SFA learning algorithm with and without initialization for different rules from two WAFs (ModSecurity OWASP CRS and PHPIDS).

10.4 SFADiff Evaluation

10.4.1 Initialization evaluation

Our first goal is to evaluate the efficiency of our observation table initialization algorithm as a method to reduce the number of equivalence queries while inferring similar models. The experimental setup is motivated by our assumptions that the initialization model and the target model would be similar. For that purpose, we utilized 9 regular expression filters from two different versions of ModSecurity (versions 3.0.0 and 2.2.7) and PHPIDS WAFs (versions 0.7.0 and 0.6.3). The filters in the newer versions of the systems have been refined to either patch evasions or possibly to reduce false positive rate.

For our first experiment we used an alphabet of 92 symbols, the same one used in our next experiments, which contains most printable ASCII characters. Since, in this experiment, we would like to measure the reduction offered by our initialization algorithm in terms of equivalence queries, we simulated a complete equivalence oracle by comparing each inferred model with the target regular expression.

Results. Table 10-7 shows the results of our experiments. First, notice that in most cases the updated filters contain more states than their previous versions. This is expected, since most of the times the filters are patched to cover additional attacks, which requires the addition of more states for covering these extra cases. We can see

OS	States	Queries
OSX Yosemite (version 14.5.0)	7	858
Debian Linux (Kernel v3.2.0)	9	1100
FreeBSD 10.3	9	1100

Table 10.7: Results for different TCP implementations: Number of states in each model and number of membership queries required to infer the model.

that, in general, our algorithm offers a massive reduction of approximately 50× in the number of equivalence queries utilized in order to infer a correct model. This comes with a trade-off since the number of membership queries are increased by a factor of 1.15×, on average. However, equivalence queries are usually orders of magnitude slower than membership queries. Therefore, the initialization algorithm results in significant overall performance gain. We notice that 2/3 cases where we observed a large increase (more than 1.2×) in membership queries (filters PHPIDS 50 & PHPIDS 56) are filters for which states were removed in the new version of the system. This is expected since, in that case, SFADiff makes redundant queries for an entry in the observation table that does not correspond to an access string. Another possible reason for an increase in the number of the membership queries is the chance that the distinguishing set obtained by the SFA learning algorithm is smaller than the one obtained by the initialization algorithm which is always of size $n - 1$ where n is the number of states in a filter. Exploring ways to obtain a distinguishing set of minimum size is an interesting direction in order to further develop our initialization algorithm. Nevertheless, in all cases, the new versions of the filters were similar in structure with the older versions and thus, our initialization algorithm was able to reconstruct a large part of the filter and massively reduce the number of equivalence queries required to obtain the correct model.

10.4.2 TCP state machines

For our experiments with TCP state machines, we run a simple TCP server on the test machine while the learning algorithm runs as a client on another machine in the same LAN. Because the TCP protocol will, possibly, emit output for each packet sent, the ASKK algorithm is not suited for this case. Thus, we used our algorithm for

Input	Linux	OSX	FreeBSD
S, S	SA, RA	SA, RA, RA	SA
S, A, F	SA, A, FA	SA	SA
S, RA, A	SA, R	SA, R	SA

Table 10.8: Some example fingerprinting packet sequences found by SFADiff across different TCP implementations. The TCP flags that are set for the input packets are abbreviated as follows: SYN(S), ACK(A), FIN(F), and RST(R).

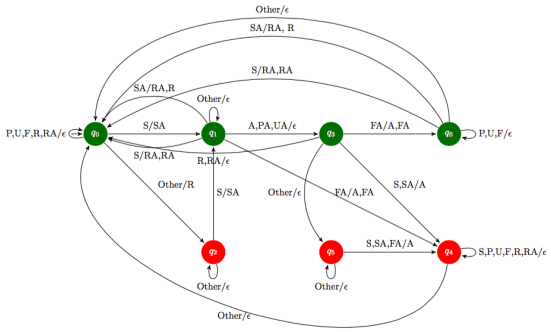

Figure 10-8: State machine inferred by SFADiff for Mac OSX TCP implementation. The TCP flags that are set for the input packets are abbreviated as follows: SYN(S), ACK(A), FIN(F), PSH(P), URG(U), and RST(R).

learning deterministic transducers in order to infer models of the TCP state machines.

Alphabet. For this set of experiments, we focus on the effect of TCP flags on the TCP protocol state transitions. Specifically, we select an alphabet with 11 symbols including 6 TCP flags: SYN(S), ACK(A), FIN(F), PSH(P), URG(U), and RST(R) along with all possible combinations of these flags with the ACK flag, i.e., SA, FA, PA, UA, and RA.

Membership queries. Once our learning algorithm formulates a membership query, our client implementation creates a sequence of TCP packets corresponding to the symbols and sends them to the server.

Our server module is a simple python script which works as follows: The script is listening for new connections on a predefined port. Once a connection is established our server module makes a single `recv` call and then actively close the connection. In addition, for each different membership query we spawn a new server process on a different port to ensure that packets belonging to different membership queries will not be mixed together.

The learning algorithm handles the sequence and acknowledgement numbers in

the outgoing TCP packets in the following way: a random sequence number is used as long as no SYN packet is part of a membership query; otherwise, after sending a SYN packet we set the sequence and acknowledgement numbers of the following packets in manner consistent with the TCP protocol specification. In case the learning algorithm receives a RST packet during the execution of a membership query, we also reset the state of the sequence numbers, i.e. we start sending random sequence numbers again until the next SYN packet is send.

After sending each packet from a membership query, the learning algorithm waits for the response for each packet using a time window. If the learning algorithm receives any re-transmitted packets during that time, it ignores those packets. We detect re-transmitted packets by checking for duplicate sequence/acknowledgement numbers. Ignoring the re-transmitted packets is crucial for the convergence of the learning algorithm as it helps us avoid any non-determinism caused by the timing of the packets.

Initialization. As TCP membership queries usually outputs more information in terms of packets than one bit, our algorithm worked efficiently for the TCP implementations even without any initialization. Thus, for this experiment, we start the learning algorithm without any initial model.

Results. We used SFADIFF in order to infer models for the TCP implementations of three different operating systems: Debian Linux, Mac OSX and FreeBSD. The inferred models contain all state transitions that are necessary to capture a full TCP session. Figure 10-8 shows the inferred state machine for Mac OSX. States in green color are part of a normal TCP session while states in red color are reached when an invalid TCP packet sequence is sent by the client. The path $q_0 \rightarrow q_1 \rightarrow q_3$ is where the TCP three-way handshake takes place and it is leading to state q_3 where the connection is established, while the path $q_3 \rightarrow q_6 \rightarrow q_0$ close the connection and returns to the initial state (q_0). Table 10.7 shows that the inferred model for Mac OSX contain fewer states than the respective FreeBSD and Linux models. Manual inspection of the models revealed that these additional states are due to different handling of invalid TCP packet sequences. Finally, in Table 10.8, we present some

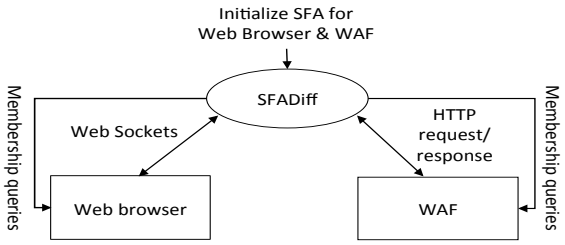

Figure 10-9: The setup for SFADiff finding differences between the HTML/-JavaScript parsing in Web browsers and WAFs.

sample differences found by SFADiff. Note that, even though the state machines of Linux and FreeBSD contain the same number of states, they are not equivalent, as we can see in Table 10.8, since the two implementations produce different outputs for all three inputs.

10.4.3 Web Application Firewalls and Browsers

In this setting, we perform two sets of experiments: (i) we use SFADiff to explore differences in HTML/JavaScript signatures used by different WAFs for detecting XSS attacks; and (ii) we use SFADiff to find differences in the JavaScript parsing implementation of the browsers and the WAFs that can be exploited to launch XSS attacks while bypassing the WAFs.

For these tests, we configure the WAFs to run as a server and the learning algorithm executes as a client on the same machine. The browser instance is also running on the same machine. The learning algorithm communicates with the browser instance through WebSockets. The learning algorithm can test whether an HTML page with some JavaScript code is correctly parsed by the browser and if the embedded JavaScript is executed or not by exchanging messages with the browser instance. The overall setup is shown in Figure 10-9.

Alphabet. We used an alphabet of 92 symbols containing most printable ASCII characters. This allows us to encode a wide range of Javscript attack vectors.

Membership queries to the browser. In order to allow the learning algorithm to drive the browser, we make the browser connect to a web server controlled by

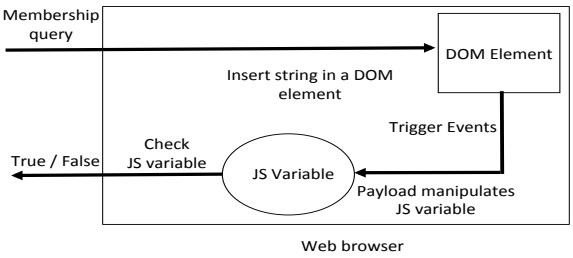

Figure 10-10: The implementation of membership queries for Web browsers.

the learning algorithm. Next, the learning algorithm sends a message to the browser over WebSockets with the HTML/JavaScript content corresponding to a membership query as the message's payload. Upon receiving such a message, the browser sets the query payload as the `innerHTML` of a DOM element and waits for the DOM element to be loaded. The user's browser dispatches a number of events (such as "click") on the DOM element and examines if the provided string led to JavaScript execution. These events are necessary for triggering the JavaScript execution in certain payloads. In order to examine if the JavaScript execution was successful, the browser monitors for any change in the value of a JavaScript variable located in the page. The payload, when executed, changes the variable value in order to notify that the execution was successful. Furthermore, in order to cover more cases of JavaScript execution, the user's browser also monitors for any JavaScript errors that indicate JavaScript execution. After testing the provided string, the user's browser sends back a response message containing a boolean value that indicates the result. The results of the membership queries are cached by the learning algorithm in order to be reused in the future. The details of our implementation of membership queries for the browsers is shown in Figure 3.

Membership queries to the WAF. SFADiff sends an HTTP request to the WAFs containing the corresponding HTML/JavaScript string as payload to perform a membership request, The WAF analyzes the request, decides whether to allow/block the payload, and communicates the decision back to SFADiff. SFADiff caches the results of the membership queries in order to be reused in the future.

Equivalence queries. We perform equivalence queries in two ways: first, when-

ever an equivalence query is sent either to the browser or to a WAF, we check that the model complies to the answers of all membership queries made so far. This ensures that simple model errors will be corrected before we perform more expensive operations such as cross-checking the two models against each other. Afterwards, we proceed to collect candidate differences and verify them against the actual test programs.

Initialization. We initialize the observation tables for both the browser and the WAF using a small subset of filters that come bundled with PHPIDS and ModSecurity, two open-source WAFs in our test set. However, in the case of the browser we slightly modify the filters in order to execute our JavaScript function call if they are successfully parsed by the browser.

Fingerprinting WAFs. In order to evaluate the efficiency of our fingerprint generation algorithm we selected 4 different WAFs. Furthermore, To demonstrate the ability of our system to generate fine-grained fingerprints we also include 4 different versions of PHPIDS in our test set. As an additional way to avoid blowup in the fingerprint tree size we employ the following optimization: Whenever a fingerprint is found for a pair of firewalls, we check whether this fingerprint is able to distinguish any other firewalls in the set and thus further reduce the remaining possibilities. This simple heuristic significantly reduces the size of the tree: Our basic algorithm creates a full binary tree of height 8 while this heuristic reduced the size of the tree to just 4 levels.

Figure 10.4.3 presents the results of our experiment. The resulting fingerprinting tree also provides hints on how restrictive each firewall is compared to the others. An interesting observation is that we see the different versions of PHPIDS to be increasingly restrictive in newer versions, by rejecting more of the generated fingerprint strings. This is natural since newer versions are usually patching vulnerabilities in the older filters. Finally, we would like to point out that some of the fingerprints are also suggesting potential vulnerabilities in some filters. For example, the top level string, *union select from*, is accepted by all versions of PHPIDS up to 0.6.5, while being rejected by all other filters. This may raise suspicion since this string can be

easily extended into a full SQL injection attack.

Evading WAFs through browser parser inference. For our last experiment we considered the setting of evaluating the robustness of WAFs against evasion attacks. Recall, that, in the context of XSS attacks, WAFs are attempting to reimplement the parsing logic of a browser in order to detect inputs that will trigger JavaScript execution. Thus, finding discrepancies between the browser parser and the WAF parser allows us to effectively construct XSS attacks that will bypass the WAF. In order to accomplish that, we used the setup described previously. However, instead of cross-checking the WAFs against each other, we cross-checked WAFs against the web browser in order to detect inputs which are successfully executing JavaScript in the browser, however they are not considered malicious by the WAF.

Table 10.9 shows the result of a sample execution of our system in the setting of detecting evasions. The execution time of our algorithm was about 6 minutes, in which 53 states were discovered in the browser parser and 36 states in PHPIDS. Our system converged fast into a vulnerability after improving the generated SFA models using the cached membership queries. This optimization was very important in order to correct invalid transitions generated by the learning algorithm in the inferred models. The number of invalid attacks that were attempted was 4. Each failed attack led to the refinement of the SFA models and the generation of new candidate differences. At some point the vector " " was reported as a difference by SFADiff.

We were able to detect the same vulnerability using all major browsers and furthermore, the same problem was found to affect the continuation of PHPIDS, the Expose WAF. Finally, we point out that our algorithm also found three more variations of the same attack vector, using the characters "!", and ";".

Evasion analysis. Figures 10-11 and 10-12 shows simplified models of the parser implemented by the WAF and the browser respectively. These models contain a minimal number of states in order to demonstrate the aforementioned evasion attack. Notice that, intuitively, the cause for the vulnerability is the fact that from state q_1^p the parser of PHPIDS will return to the initial state with any non alphanumeric input,

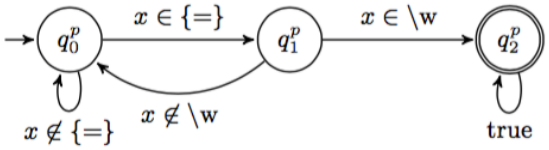

Figure 10-11: PHPIDS 0.7 parser (simplified version).

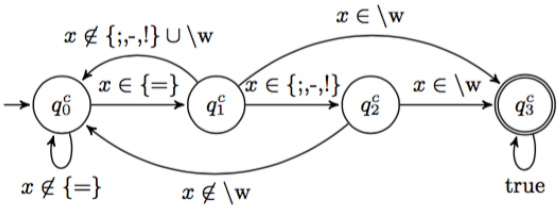

Figure 10-12: Google Chrome parser (simplified version).

while the Google Chrome parser has the choice to first transition to q_2^c and then to an accepting state q_3^c using any alphanumeric character. For example, with an input "=!a" the product automaton will reach the point of exposure (q_0^p, q_3^c). Furthermore, using our root cause analysis, all different evasions we detected are grouped under a single root cause. This is intuitively correct, since a patch, which adds the missing state in the PHPIDS parser will address all evasion attacks at once.

10.4.4 Comparison with black-box fuzzing

To the best of our knowledge there is no publicly available black-box system which is capable of performing black-box differential testing like SFADiff. A straightforward approach would be to use a black-box fuzzer (e.g. the PEACH fuzzing platform [6]) and send each input generated by the fuzzer to both programs. Afterwards, the outputs from both programs are compared to detect any differences. Note that, like SFADiff, fuzzers also start with some initial inputs (seeds) which they mutate in order to generate more inputs for the target program. We argue that our approach is more effective in discovering differences for two reasons:

Adaptive input generation. Fuzzers incorporate a number of different strategies in order to mutate previous inputs and generate new ones. For example, PEACH

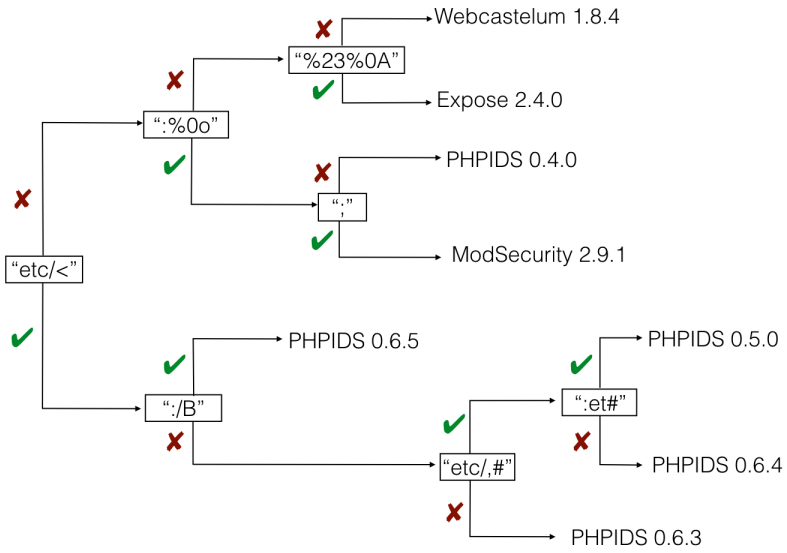

Figure 10-13: Fingerprint tree for different web application firewalls.

supports more than 20 different strategies for mutating an input. However, assuming that a new input does not cause a difference, no further information is extracted from it; the next inputs are unrelated to the previous ones. On the contrary, each input submitted by SFADiff to the target program provides more information about the structure of the program and its output determines the next input that will be tested. For example, in the execution shown in table 10.9, SFADiff utilized the initialization model and detected the additional state in Chrome's parser (cf. figures 10-11, 10-12). Notice that, the additional state in Chrome's parser was not part of the model used for initialization. This allowed SFADiff to quickly discover an evasion attack after a few refinements in the generated models. Each refinement discarded a number of candidate differences and drove the generation of new inputs based on the output of previous ones.

Root cause analysis. In the presence of a large number of differences, black-box fuzzers are unable to categorize the differences without some form of white-box access to the program (e.g. crash dumps). On the other hand, as demonstrated in the evasion analysis paragraph of section 10.4.3, our root cause analysis algorithm

Attributes	Browser Model	WAF Model
Membership	6672	4241
Cached Membership	448	780
Equivalence	0	3
Cached Equivalence	40	106
Learned States	53	36
Cross-Check Times	4	4
Provided Browser Model	(<(p|div|form|input) onclick=a()>) (</(p|div|form|input)>)	
Vulnerability Discovered		
Execution Time	382.12 seconds	

Table 10.9: A sample execution that found an evasion attack for PHPIDS 0.7 and Google Chrome on MAC OSX.

provides a meaningful categorization of the differences based on the execution path they follow in the generated models.

Chapter 11

Conclusions

In this thesis we studied the idea of extracting formal models from programs using novel automata and transducer learning algorithms with the goal of checking security properties such as robustness against code injection attacks. In order to achieve this goal, we have developed a number of novel learning algorithms, namely the MAT^* algorithm for learning symbolic automata and new algorithms for learning partial and non-deterministic functional transducers. In terms of applications we presented the GOFA algorithm for evaluating the robustness of filters and sanitizers against code injection attacks and differential automata learning and `sfadiff` a technique and tool which can be used when the specification which is required by the GOFA algorithm is either unavailable or imprecise. We evaluated our algorithms against state of the art web application firewalls and found a large number of new attacks.

In terms of future work, we need to consider both short term as well as long term goals: Short terms goals include finding more applications for our techniques as well as extending our current learning algorithms into more expressive models to allow us to check if more complicated filter and sanitizer routines. A concrete application which requires more expressive models is path normalization functions which usually require pushdown transducers in order to model effectively.

Finally, in terms of long term research goals, we believe that the problem of learning the proper abstractions will play a significant role in order to develop effective program analysis systems. As we demonstrated in our evaluation it is common to find

vulnerabilities were common heuristics such as code and basic block coverage will fail to uncover the underlying problems and addressing the problems in the right abstraction layer (in our case using transducers) is of fundamental importance. Therefore, we believe that developing systems which are capable of abstracting away irrelevant information and reason about the parts of code which are relevant to the security property to be checked is very important. Taking this concept further, we believe that an important long term goal is to develop systems that can infer proper abstractions which can be used in order to reason about different systems. While this goal seems out of reach using current technologies we believe that working towards this direction may bring important new insights in the field of program analysis.

Bibliography

[1] Bek guide. http://www.rise4fun.com/Bek/tutorial/guide2. Accessed: 2015-11-10.

[2] Fado library. https://pypi.python.org/pypi/FAdo. Accessed: 2015-11-10.

[3] lorisdanto/symbolicautomata: Library for symbolic automata and symbolic visibly pushdown automata. https://github.com/lorisdanto/symbolicautomata/. (Accessed on 01/29/2018).

[4] Microsoft antixss library. https://msdn.microsoft.com/en-us/security/aa973814.aspx. Accessed: 2015-11-10.

[5] Mod-security. https://www.modsecurity.org/. Accessed: 2015-11-10.

[6] Peach fuzzer. http://www.peachfuzzer.com/. (Accessed on 08/10/2016).

[7] Phpids source code. https://github.com/PHPIDS/PHPIDS. Accessed: 2015-11-10.

[8] Xss cheat sheet. https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet. Accessed: 2016-01-10.

[9] Codeigniter web framework. https://codeigniter.com/, 2018. (Accessed on 06/16/2018).

[10] Data validation - wordpress codex. https://codex.wordpress.org/Data_Validation, 2018. (Accessed on 06/16/2018).

[11] syspass :: Systems password manager. http://syspass.org/index-en.html, 2018. (Accessed on 11/16/2017).

[12] Taskfreak! web based task manager and todo list, project management made easy. http://www.taskfreak.com/, 2018. (Accessed on 16/16/2018).

[13] Fides Aarts, Paul Fiterau-Brostean, Harco Kuppens, and Frits Vaandrager. Learning register automata with fresh value generation. In *International Colloquium on Theoretical Aspects of Computing*, pages 165–183. Springer, 2015.

[14] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987.

[15] Dana Angluin, Sarah Eisenstat, and Dana Fisman. Learning regular languages via alternating automata. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, pages 3308–3314. AAAI Press, 2015.

[16] Dana Angluin and Michael Kharitonov. When wonâĂš t membership queries help? *Journal of Computer and System Sciences*, 50(2):336–355, 1995.

[17] G. Argyros, I. Stais, A. Keromytis, and A. Kiayias. Back in black: Towards formal, black-box analysis of sanitizers and filters. In *Security and privacy (S&P), 2016 IEEE symposium on*, 2016.

[18] José L Balcázar, Josep Díaz, Ricard Gavalda, and Osamu Watanabe. Algorithms for learning finite automata from queries: A unified view. In *Advances in Algorithms, Languages, and Complexity*, pages 53–72. Springer, 1997.

[19] Davide Balzarotti, Marco Cova, Vika Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 387–401. IEEE, 2008.

[20] Therese Berg, Bengt Jonsson, and Harald Raffelt. Regular inference for state machines with parameters. In *International conference on fundamental approaches to software engineering*, pages 107–121. Springer, 2006.

[21] Nikolaj Bjorner, Pieter Hooimeijer, Ben Livshits, David Molnar, and Margus Veanes. Symbolic finite state transducers, algorithms, and applications. In *IN: PROC. 39TH ACM SYMPOSIUM ON POPL.*, 2012.

[22] Benedikt Bollig, Peter Habermehl, Carsten Kern, and Martin Leucker. Angluin-style learning of nfa. In *Proceedings of the 21st International Jont Conference on Artifical Intelligence*, IJCAI'09, pages 1004–1009, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[23] Matko Botinčan and Domagoj Babić. Sigma*: symbolic learning of input-output specifications. In *ACM SIGPLAN Notices*, volume 48, pages 443–456. ACM, 2013.

[24] Arnaud Carayol and Matthew Hague. Saturation algorithms for model-checking pushdown systems. *EPTCS*, 151:1–24, 2014.

[25] Sofia Cassel, Falk Howar, Bengt Jonsson, Maik Merten, and Bernhard Steffen. A succinct canonical register automaton model. *Journal of Logical and Algebraic Methods in Programming*, 84(1):54–66, 2015.

[26] Chia Yuan Cho, Eui Chul Richard Shin, Dawn Song, et al. Inference and analysis of formal models of botnet command and control protocols. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 426–439. ACM, 2010.

[27] Tsun S. Chow. Testing software design modeled by finite-state machines. *IEEE transactions on software engineering*, (3):178–187, 1978.

[28] Corinna Cortes, Leonid Kontorovich, and Mehryar Mohri. Learning languages with rational kernels. In *International Conference on Computational Learning Theory*, pages 349–364. Springer, 2007.

[29] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[30] Loris D'Antoni, Zachary Kincaid, and Fang Wang. A symbolic decision procedure for symbolic alternating finite automata. *arXiv preprint arXiv:1610.01722*, 2016.

[31] Loris D'Antoni and Margus Veanes. Minimization of symbolic automata. In *ACM SIGPLAN Notices*, volume 49, pages 541–553. ACM, 2014.

[32] Loris D'Antoni, Margus Veanes, Benjamin Livshits, and David Molnar. Fast: A transducer-based language for tree manipulation. In *ACM SIGPLAN Notices*, volume 49, pages 384–394. ACM, 2014.

[33] Colin De la Higuera. *Grammatical inference: learning automata and grammars*. Cambridge University Press, 2010.

[34] Joeri De Ruiter and Erik Poll. Protocol state fuzzing of tls implementations. In *USENIX Security Symposium*, pages 193–206, 2015.

[35] François Denis, Aurélien Lemay, and Alain Terlutte. Residual finite state automata. *Fundamenta Informaticae*, 51(4):339–368, 2002.

[36] Samuel Drews and Loris DâĂŹAntoni. Learning symbolic automata. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 173–189. Springer, 2017.

[37] Loris DâĂŹAntoni and Margus Veanes. Equivalence of extended symbolic finite transducers. In *Computer Aided Verification*, pages 624–639. Springer, 2013.

[38] Loris DâĂŹAntoni and Margus Veanes. Extended symbolic finite automata and transducers. *Formal Methods in System Design*, July 2015.

[39] Loris DâĂŹAntoni and Margus Veanes. The power of symbolic automata and transducers. In *International Conference on Computer Aided Verification*, pages 47–67. Springer, 2017.

[40] Paul Fiterău-Broştean, Toon Lenaerts, Erik Poll, Joeri de Ruiter, Frits Vaandrager, and Patrick Verleg. Model learning and model checking of ssh implementations. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, pages 142–151. ACM, 2017.

[41] Zoltán Fülöp and Heiko Vogler. *Syntax-directed semantics: Formal models based on tree transducers*. Springer Science & Business Media, 2012.

[42] E Mark Gold. Complexity of automaton identification from given data. *Information and control*, 37(3):302–320, 1978.

[43] Alex Groce, Doron Peled, and Mihalis Yannakakis. Adaptive model checking. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 357–370. Springer, 2002.

[44] Amaury Habrard and Jose Oncina. Learning multiplicity tree automata. In *International Colloquium on Grammatical Inference*, pages 268–280. Springer, 2006.

[45] Mario Heiderich. *Web Application Obfuscation:-/WAFs.. Evasion.. Filters//alert (/Obfuscation/)-*. Elsevier, 2011.

[46] Mario Heiderich, Jörg Schwenk, Tilman Frosch, Jonas Magazinius, and Edward Z Yang. mxss attacks: Attacking well-secured web-applications by using innerhtml mutations. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 777–788. ACM, 2013.

[47] Pieter Hooimeijer, Benjamin Livshits, David Molnar, Prateek Saxena, and Margus Veanes. Fast and precise sanitizer analysis with bek. In *Proceedings of the 20th USENIX conference on Security*, pages 1–1. USENIX Association, 2011.

[48] Pieter Hooimeijer, Prateek Saxena, Benjamin Livshits, Margus Veanes, and David Molnar. Fast and precise sanitizer analysis with bek. In *In 20th USENIX Security Symposium*, 2011.

[49] Falk Howar, Bernhard Steffen, Bengt Jonsson, and Sofia Cassel. Inferring canonical register automata. In *VMCAI*, volume 7148, pages 251–266. Springer, 2012.

[50] Falk Howar, Bernhard Steffen, and Maik Merten. Automata learning with automated alphabet abstraction refinement. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 263–277. Springer, 2011.

[51] Malte Isberner, Falk Howar, and Bernhard Steffen. The ttt algorithm: A redundancy-free approach to active automata learning. In *RV*, pages 307–322, 2014.

[52] Michael Kearns and Leslie Valiant. Cryptographic limitations on learning boolean formulae and finite automata. *Journal of the ACM (JACM)*, 41(1):67–95, 1994.

[53] Michael J Kearns and Umesh Virkumar Vazirani. *An introduction to computational learning theory*. MIT press, 1994.

[54] Ali Khalili and Armando Tacchella. Learning nondeterministic mealy machines. In *International Conference on Grammatical Inference*, pages 109–123, 2014.

[55] Leonid Kontorovich. A universal kernel for learning regular languages. In *MLG*, 2007.

[56] D. Kozen. Lower bounds for natural proof systems. In *FOCS*, 1977.

[57] Harry R Lewis and Christos H Papadimitriou. *Elements of the Theory of Computation*. Prentice Hall PTR, 1997.

[58] Nuo Li, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. Reggae: Automated test generation for programs using complex regular expressions. In *Automated Software Engineering, 2009. ASE'09. 24th IEEE/ACM International Conference on*, pages 515–519. IEEE, 2009.

[59] Anthony W Lin and Pablo Barceló. String solving with word equations and transducers: towards a logic for analysing mutation xss. In *ACM SIGPLAN Notices*, volume 51, pages 123–136. ACM, 2016.

[60] Oded Maler and Irini-Eleftheria Mens. Learning regular languages over large alphabets. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 485–499. Springer, 2014.

[61] Konstantinos Mamouras, Mukund Raghothaman, Rajeev Alur, Zachary G Ives, and Sanjeev Khanna. Streamqre: modular specification and efficient evaluation of quantitative queries over streaming data. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 693–708. ACM, 2017.

[62] Yasuhiko Minamide. Static approximation of dynamically generated web pages. In *Proceedings of the 14th international conference on World Wide Web*, pages 432–441. ACM, 2005.

[63] Joshua Moerman, Matteo Sammartino, Alexandra Silva, Bartek Klin, and Michal Szynwelski. Learning nominal automata. In *Proceedings of the 44th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2017.

[64] Mehryar Mohri. Finite-state transducers in language and speech processing. *Computational linguistics*, 23(2):269–311, 1997.

[65] Mehryar Mohri. Minimization algorithms for sequential transducers. *Theoretical Computer Science*, 234(1-2):177–201, 2000.

[66] Atsuyoshi Nakamura. An efficient query learning algorithm for ordered binary decision diagrams. *Information and Computation*, 201(2):178–198, 2005.

[67] José Oncina and Pedro Garcia. Identifying regular languages in polynomial time. In *Advances in Structural and Syntactic Pattern Recognition*, pages 99–108. World Scientific, 1992.

[68] José Oncina, Pedro García, and Enrique Vidal. Learning subsequential transducers for pattern recognition interpretation tasks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(5):448–458, 1993.

[69] OWASP. Top ten 2017. `https://www.owasp.org/index.php/Top_10_2017-Top_10`. (Accessed on 11/15/2017).

[70] Doron Peled, Moshe Y Vardi, and Mihalis Yannakakis. Black box checking. In *Formal Methods for Protocol Engineering and Distributed Systems*, pages 225–240. Springer, 1999.

[71] PHP. preg_replace - manual. `http://php.net/manual/en/function.preg-replace.php`, 2018. (Accessed on 07/11/2018).

[72] Leonard Pitt and Manfred K Warmuth. The minimum consistent dfa problem cannot be approximated within any polynomial. *Journal of the ACM (JACM)*, 40(1):95–142, 1993.

[73] Harald Raffelt, Bernhard Steffen, and Therese Berg. Learnlib: A library for automata learning and experimentation. In *Proceedings of the 10th international workshop on Formal methods for industrial critical systems*, pages 62–71. ACM, 2005.

[74] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[75] Ronald L Rivest and Robert E Schapire. Inference of finite automata using homing sequences. *Information and Computation*, 103(2):299–347, 1993.

[76] Olli Saarikivi and Margus Veanes. Minimization of symbolic transducers. In *International Conference on Computer Aided Verification*, pages 176–196. Springer, 2017.

[77] Muzammil Shahbaz and Roland Groz. Inferring mealy machines. *FM*, 9:207–222, 2009.

[78] Michael Sipser. *Introduction to the Theory of Computation*, volume 2. Thomson Course Technology Boston, 2006.

[79] Suphannee Sivakorn, George Argyros, Kexin Pei, Angelos D Keromytis, and Suman Jana. Hvlearn: Automated black-box analysis of hostname verification in ssl/tls implementations. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 521–538. IEEE, 2017.

[80] Frits Vaandrager. Model learning. *Commun. ACM*, 60(2):86–95, January 2017.

[81] Margus Veanes. Symbolic string transformations with regular lookahead and rollback. In *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, pages 335–350. Springer, 2014.

[82] Margus Veanes, Peli de Halleux, and Nikolai Tillmann. Rex: Symbolic regular expression explorer. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ICST '10, pages 498–507, Washington, DC, USA, 2010. IEEE Computer Society.

[83] Margus Veanes, Todd Mytkowicz, David Molnar, and Benjamin Livshits. Data-parallel string-manipulating programs. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 139–152. ACM, 2015.

[84] Juan Miguel Vilar. Query learning of subsequential transducers. In *International Colloquium on Grammatical Inference*, pages 72–83. Springer, 1996.

[85] Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In *ACM Sigplan Notices*, volume 42, pages 32–41. ACM, 2007.

[86] Bruce W Watson. Implementing and using finite automata toolkits. *Natural Language Engineering*, 2(4):295–302, 1996.