2016 IEEE Symposium on Security and Privacy

Back in Black: Towards Formal, Black Box Analysis of Sanitizers and Filters

George Argyros
Columbia University
argyros@cs.columbia.edu

Ioannis Stais
University of Athens
i.stais@di.uoa.gr

Aggelos Kiayias
University of Athens
aggelos@di.uoa.gr

Angelos D. Keromytis
Columbia University
angelos@cs.columbia.edu

***Abstract*—We tackle the problem of analyzing filter and sanitizer programs remotely, i.e. given only the ability to query the targeted program and observe the output. We focus on two important and widely used program classes: regular expression (RE) filters and string sanitizers. We demonstrate that existing tools from machine learning that are available for analyzing RE filters, namely automata learning algorithms, require a very large number of queries in order to infer real life RE filters. Motivated by this, we develop the first algorithm that infers *symbolic* representations of automata in the standard membership/equivalence query model. We show that our algorithm provides an improvement of x15 times in the number of queries required to learn real life XSS and SQL filters of popular web application firewall systems such as mod-security and PHPIDS. Active learning algorithms require the usage of an equivalence oracle, i.e. an oracle that tests the equivalence of a hypothesis with the target machine. We show that when the goal is to audit a target filter with respect to a set of attack strings from a context free grammar, i.e. find an attack or infer that none exists, we can use the attack grammar to implement the equivalence oracle with a single query to the filter. Our construction finds on average 90% of the target filter states when no attack exists and is very effective in finding attacks when they are present.**

For the case of string sanitizers, we show that existing algorithms for inferring sanitizers modelled as Mealy Machines are not only inefficient, but lack the expressive power to be able to infer real life sanitizers. We design two novel extensions to existing algorithms that allow one to infer sanitizers represented as single-valued transducers. Our algorithms are able to infer many common sanitizer functions such as HTML encoders and decoders. Furthermore, we design an algorithm to convert the inferred models into BEK programs, which allows for further applications such as cross checking different sanitizer implementations and cross compiling sanitizers into different languages supported by the BEK backend. We showcase the power of our techniques by utilizing our black-box inference algorithms to perform an equivalence checking between different HTML encoders including the encoders from Twitter, Facebook and Microsoft Outlook email, for which no implementation is publicly available.

I. Introduction

Since the introduction and popularization of code injection vulnerabilities as major threats for computer systems, sanitization and filtering of unsafe user input is paramount to the design and implementation of a secure system. Unfortunately correctly implementing such functionalities is a very challenging task. There is a large literature on attacks and bypasses in implementations both of filter and sanitizer functions [1]–[3].

The importance of sanitizers and filters motivated the development of a number of algorithms and tools [4]–[7] to analyze such programs. More recently, the BEK language [8] was introduced. BEK is a Domain Specific Language(DSL) which allows developers to write string manipulating functions in a language which can then be compiled into symbolic finite state transducers(SFTs). This compilation enables various analysis algorithms for checking properties like commutativity, idempotence and reversibility. Moreover, one can efficiently check whether two BEK programs are equal and, in the opposite case to obtain a string in which the two programs differ.

The BEK language offers a promising direction for the future development of sanitizers where the programs developed for sanitization will be formally analyzed in order to verify that certain desired properties are present. However, the vast majority of code is still written in languages like PHP/Java and others. In order to convert the sanitizers from these languages to BEK programs a significant amount of manual effort is required. Even worst, BEK is completely unable to reason for sanitizers whose source code is not available. This significantly restricts the possibilities for applying BEK to find real life problems in deployed sanitizers.

In this paper we tackle the problem of black-box analysis of sanitizers and filters. We focus our analysis on *regular expression* filters and string sanitizers which are modelled as finite state transducers. Although regular expression filters are considered suboptimal choices for building robust filters [9], their simplicity and efficiency makes them a very popular option especially for the industry.

Our analysis is black-box, that is, without access to any sort of implementation or source code. We only assume the ability to query a filter/sanitizer and obtain the result. Performing a black-box analysis presents a number of advantages; firstly, our analysis is generic, i.e. indepedent of any programming language or system. Therefore, our system can be readily applied to any software, without the need for a large engineering effort to adjust the algorithms and implementation into a new programming language. This is especially important since in today's world, the number of programming languages used varies significantly. To give an example, there are over 15 different programming languages used in the backend of the 15 most popular websites [10].

The second advantage of performing a black-box analysis comes out of necessity rather than convience. Many times, access to the source code of the program to be analyzed is unavailable. There are multiple reasons this may happen; for one, the service might be reluctant to share the source code

DOI 10.1109/SP.2016.14

of its product website even with a trusted auditor. This is the reason, that a large percentage of penetration tests are performed in a black-box manner. Furthermore, websites such as the ones encountered in the deep web, for example TOR hidden services, are designed to remain as hidden as possible. Finally, software running in hardware systems such as smart cards is also predominately analyzed in a black-box manner.

Our algorithms come with a formal analysis; for every algorithm we develop, we provide a precise description of the conditions and assumptions under which the algorithm will work within a given time bound and provide a correct model of the target filter or sanitizer.

Our goal is to build algorithms that will make it easier for an auditor to understand the functionality of a filter or sanitizer program without access to its source code. We begin by evaluating the most common machine learning algorithms which can be used for this task. We find that these algorithms are not fit for learning filters and sanitizers for different reasons: The main problem in inferring regular expressions with classical automata inference algorithms is the explosion in the number of queries caused by the large alphabets over which the regular expressions are defined. This problem also occurs in the analysis of regular expressions in program analysis applications (whitebox analysis), which motivated the development of the class of symbolic finite automata which effectively handles these cases [11]. Motivated by these advances, we design the first algorithm that infers symbolic finite automata (SFA) in the standard active learning model of membership and equivalence queries. We evaluate our algorithm in 15 real life regular expression filters and show that our algorithm utilizes on average 15 times less queries than the traditional DFA learning algorithm in order to infer the target filter.

The astute reader will counter that an equivalence oracle (i.e., an oracle that one submits a hypothesized model and a counterexample is returned if there exists one) is not available in remote testing and thus it has to be simulated at potentially great cost in terms of number of queries. In order to address this we develop a structured approach to equivalence oracle simulation that is based on a given context free grammar G. Our learning algorithm will simulate equivalence queries by drawing a single random string w from $\mathcal{L}(G) \setminus \mathcal{L}(H)$ where $\mathcal{L}(H)$ is the language of the hypothesis. If w belongs to the target we have our counterexample, while if not, we have found a string w that is not recognized by the target. In our setting strings that are not recognized by the target filter can be very valuable: we set G to be a grammar of *attack strings* and we turn the failure of our equivalence oracle simulation to the discovery of a filter bypass! This also gives rise to what we call Grammar Oriented Filter Auditing (GOFA): our learning algorithm, equipped with a grammar of attack strings, can be used by a remote auditor of a filter to either find a vulnerability or obtain a model of the filter (in the form of an SFA) that can be used for further (whitebox) testing and analysis.

Turning our attention to sanitizers, we observe that inferring finite state transducers suffers from even more fundamental problems. Current learning algorithms infer models as Mealy machines, i.e. automata where at each transition one input symbol is consumed and one output symbol is produced. However, this model is very weak in capturing the behavior of real life sanitizers where for each symbol consumed multiple, or none, symbols are produced. Even worse, many modern sanitizers employ a “lookahead”, i.e. they read many symbols from the input before producing an output symbol. In order to model such behavior the inferred transducers must be *non deterministic*. To cope with these problems we make three contributions: First, we show how to improve the query complexity of the Shabaz-Groz algorithm [12] exponentially. Second, we design an extension of the Shabaz-Groz algorithm which is able to handle transducers which output multiple or no symbols in each transition. Finally, we develop a new algorithm, based on our previous extension, which is able to infer sanitizers that employ a lookahead, i.e., base their current output by reading ahead more than one symbol.

To enable more fine grained analysis of our inferred models we develop an algorithm to convert (symbolic) finite transducers with bounded lookahead into BEK programs. This algorithm enables an interesting application: In the original BEK paper [8] the authors manually converted different HTML encoder implementations into BEK programs and then used the BEK infrastructure to check equivalence and other properties. Our algorithms enable these experiments to be performed automatically, i.e. without manually converting each implementation to a BEK program and more importantly, being agnostic of the implementation details. In fact, we checked seven HTML encode implementations: three PHP implementations, one implementation from the AntiXSS library in .NET and we also included models infered from the HTML encoders used by the websites of Twitter and Facebook and by the Microsoft Outlook email service. We detected differences between many implementations and found that Twitter and Facebook’s HTML encoders match the `htmlspecialcharacters` function of PHP although the Outlook service encoder does not match the MS AntiXSS implementation in .NET. Moreover, we found that only one of these implementations is idempotent.

Finally, we point out that although our algorithms are focused on the analysis of sanitizers and filters they are general enough to potentially being applied in a number of different domains. For example, in appendix D, we show how one can use an SFA to model decision trees over the reals. In another application, Doupe et al. [13] create a state aware vulnerability scanner, where they model the different states of the application using a Mealy machine. In their paper they mention they considered utilizing inference techniques for Mealy machines but that this was infeasible, due to the large number of transitions. However, our symbolic learning algorithms are able to handle efficiently exactly those cases and thus, we believe several projects will be able to benefit from our techniques.

A. Limitations

Since the analysis we perform is black-box, all of our techniques are necessarily *incomplete*. Specifically, there might be some aspect of the target program that our algorithms will fail to discover. Our algorithms are not designed to find, for example, backdoors in filters and sanitizers where a “magic string” is causing the program to enter a hidden state. Such programs will necessarily require an exponential number of queries in the worst case in order to analyze completely. Moreover, our algorithms are not geared towards discovering new attacks for certain vulnerability classes. We assume that the description of the attack strings for a certain vulnerability class, for example XSS, is given in the form of a context free grammar.

B. Contributions

To summarize, our paper makes the following contributions:

Learning Algorithms: We present the first, to the best of our knowledge, algorithm that learns symbolic finite automata in the standard membership and equivalence query model. Furthermore, we improve the query complexity of the Shabaz-Groz algorithm [12], a popular Mealy machine learning algorithm and present an extension of the algorithm capable of handling Mealy Machines with ε-input transitions. Finally, we present a novel algorithm which is able to infer finite transducers with bounded lookahead. Our transducer learning algorithms can also be easily extended in the symbolic setting by expanding our SFA algorithm.

Equivalence Query Implementation: We present the Grammar Oriented Filter Auditing (GOFA) algorithm which implements an equivalence oracle with a single membership query for each equivalence query and demonstrate that it is capable to either detect a vulnerability in the filter if one is present or, if no vulnerability is present, to recover a good approximation of the target filter.

Conversion to BEK programs: We present, in appendix C an algorithm to convert our inferred models of sanitizers into BEK programs which can then be analyzed using the BEK infrastructure enabling further applications.

Applications/Evaluation: We showcase the wide applicability of our algorithms with a number of applications. Specifically, we perform a thorough evaluation of our SFA learning algorithm and demonstrate that it achieves a big performance increase on the total number of queries performed. We also evaluate our GOFA algorithm and demonstrate that it is able to either detect attacks when they are present or give a good approximation of the target filter. To showcase our transducer learning algorithms we infer models of several HTML encoders, convert them to BEK program and check them for equivalence.

We point out that, due to lack of space all proofs have been moved into the appendix.

II. Preliminaries

A. Background in Automata Theory

If M is a deterministic finite automaton (DFA) defined over alphabet Σ, we denote by $|M|$ the number of states of M and by $\mathcal{L}(M)$ the language that is accepted by M. For any k we denote by $[k]$ the set $\{1, \ldots, k\}$. We denote the set of states of M by Q_M. A certain subset F of Q_M is identified as the set of final states. We denote by $l : Q_M \to \{0, 1\}$ a function which identifies a state as final or non final. The program of the finite automaton M is determined by a transition function δ over $Q_M \times \Sigma \to Q_M$. For an automaton M we denote by $\neg M$ the automaton M with the final states inverted.

A push-down automaton (PDA) M extends a finite automaton with a stack. The stack accepts symbols over an alphabet Γ. The transition function is able to read the top of the stack. The transition function is over $Q_M \times \Sigma \times (\Gamma \cup \{\varepsilon\}) \to Q_M \times (\Gamma \cup \{\varepsilon\})$. A context-free grammar (CFG) G comprises a set of rules of the form $A \to w$ where $A \in V$ and $w \in (\Sigma \cup V)^*$ where V is a set of non-terminal symbols. The language defined by a CFG G is denoted by $\mathcal{L}(G)$.

A transducer T extends a finite automaton with an output tape. The automaton is capable of producing output in each transition that belongs to an alphabet Γ. The transition function is defined over $Q_M \times (\Sigma \cup \{\varepsilon\}) \to Q_M \times (\Gamma \cup \{\varepsilon\})$. A Mealy Machine M is a deterministic transducer without ε transitions where, in addition, all states are final. A non-deterministic transducer has a transition function which is a relation $\delta \subseteq Q_M \times (\Sigma \cup \{\varepsilon\}) \times Q_M \times (\Gamma \cup \{\varepsilon\})$. For general transducers (deterministic or not), following [8], we extend the definition of a transducer to produce output over Γ^*. A non-deterministic transducer is *single-valued* if it holds that for any $w \in \Sigma^*$ there exists at most one $\gamma \in \Gamma^*$ such that T on w outputs γ. A single-valued transducer T has the *bounded lookahead property* if there is a k such that any sequence of transitions involves at most k consecutive non-accepting states. We call such a sequence a *lookahead path* or *lookahead transition*. In a single valued transducer with bounded lookahead we will call the paths that start and finish in accepting states and involve only non-accepting states as lookahead paths. The path in its course consumes some input $w \in \Sigma^*$ and outputs some $\gamma \in \Gamma^*$. The bounded lookahead property definition is based on the one given by Veanes et al. [14] for Symbolic Transducers, however our definition better fits our terminology and the intuition behind our algorithms.

For a given automaton M, we denote by $M_q[s]$ the state reached when the automaton is executed from state q on input s. When the state q is omitted we assume that M is executed from the initial state. Let $l : Q \to \{0, 1\}$ be a function denoting whether a state is final. We define the transduction function $\mathcal{T}_M(u)$ as the output of a transducer/Mealy Machine M on input u omitting the subscript M when the context is clear. For transducers we will also use the notation $u[M]v$ to signify that $\mathcal{T}_M(u) = v$ for a transducer M.

For a string s, denote by s_i the i-th character of the string. In addition, we denote by $s_{>i}$ the substring s starting after s_i. The operators $s_{<i}, s_{\geq i}, s_{\leq i}$ are defined similarly. We denote by $\mathrm{suff}(s, k)$ the suffix of s of length k.

Given two DFA's M_1, M_2 it is possible to compute the intersection $M = M_1 \cap M_2$ of the two as follows. The set of states of M is the Cartesian product $Q_1 \times Q_2$ and the transition function combines the two individual transition functions to traverse over the pair of states simultaneously. The accepting states of Q_M are those that are simultaneously accepting for M_1, M_2. We can use exactly the same algorithm to obtain the intersection between a DFA M_1 and a PDA M_2. The resulting machine M is a PDA that inherits the stack operations of M_2. Moreover, one can trivially compute the completement of a DFA by switching all terminal states with non terminal and vice-versa.

Transducers are not closed under intersection and difference, and if the transducer is non-deterministic checking properties as simple as equality is undecidable. However, in the case the transducer is determinsitic or single valued then equality can be efficiently computed and in the case the transducers are not equal one can exhibit a string in which the two transducers are different efficiently [15].

B. Symbolic Finite State Automata

Symbolic Finite Automata (SFA) [16] extend classical automata by allowing transitions to be labelled with predicates rather than with concrete alphabet symbols. This allows for more compact representation of automata with large alphabets and it could allow automata that are impossible to model as DFAs when the alphabet size is infinite, as in the case where $\Sigma = \mathbb{Z}$. For the following we refer to a set of predicates $\mathcal{P}$ as a predicate family.

Definition 1. (Adapted from [16]) A symbolic finite automaton or SFA A is a tuple $(Q, q_0, F, \mathcal{P}, \Delta)$, where Q is a finite set of states, $q_0 \in Q$ the *initial state*, $F \subseteq Q$ is the set of *final states*, $\mathcal{P}$ is a predicate family and $\Delta \subseteq Q \times \mathcal{P} \times Q$ is the *move* relation.

A move $(p, \phi, q) \in \Delta$ is taken when ϕ is satisfied from the current symbol α. We will also use an alternative notation for a move (p, ϕ, q) as $p \xrightarrow{\phi} q$. We denote by $\mathrm{guard}(q)$ the set of predicate guards for the state q, in other words:

$$\mathrm{guard}(q) := \{\phi : \exists p \in Q, (q, \phi, p) \in \Delta\}$$

In this paper we are going to work with deterministic SFAs, which we define as follows:

Definition 2. A SFA A is deterministic if for all states $q \in Q$ and all distinct $\phi, \phi' \in \mathrm{guard}(q)$ we have that $\phi \wedge \phi'$ is unsatisfiable.

Finally, we also assume that for any state q and for any symbol a in the alphabet there exists $\phi \in \mathrm{guard}(q)$ such that $\phi(a)$ is true. We call such an SFA *complete*.

Finally, we define symbolic finite state transducers, the corresponding symbolic extension of transducers similarly to SFAs.

Definition 3. (Adapted from [15]) A symbolic finite transducer or SFT T is a tuple $(Q, q_0, F, \mathcal{P}, \Delta, \Gamma(x))$, where Q is a finite set of states, $q_0 \in Q$ the *initial state*, $F \subseteq Q$ is the set of *final states*, $\mathcal{P}$ is a predicate family, $\Gamma(x)$ is a set of terms representing functions over $\Sigma \to \Gamma$ and $\Delta \subseteq Q \times \mathcal{P} \times \Gamma(x) \times Q$ is the *move* relation.

C. Access and Distinguishing Strings

We will now define two sets of strings over an automaton that play a very important role in learning algorithms.

Access Strings: For an automaton M we define the set of access strings A as follows: For every state $q \in Q_M$, there is a string $s_q \in A$ such that $M[s_q] = q$. Given a DFA M, one can easily construct a minimal set of access strings by using a depth first search over the graph induced by M.

Distinguishing Strings: We define the set of distinguishing strings D for a minimal automaton M as follows: For any pair of states $q_i, q_j \in Q_M$, there exists a string $d_{i,j} \in D$ such that exactly one state of $M_{q_i}[d_{i,j}]$ and $M_{q_j}[d_{i,j}]$ is accepting. A set of distinguishing strings can be constructed using the Hopcroft algorithm for automata minimization [17].

The set of Access and Distinguishing strings play a central role in automata learning since learning algorithms try to construct these sets by querying the automaton. Once these sets are constructed then, as we will see, it is straightforward to reconstruct the automaton.

D. Learning Model

Our algorithms work in a model called **exact learning from membership and equivalence queries** [18], which is a form of active learning where the learning algorithm operates with oracle access to two types of queries:

- *Membership queries*: The algorithm is allowed to submit a string s and obtain whether $s \in \mathcal{L}(M)$.
- *Equivalence queries*: The algorithm is allowed to submit a hypothesis H which is a finite automaton and obtain either a confirmation that $\mathcal{L}(H) = \mathcal{L}(M)$ or a string z that is a *counterexample*, i.e., a string z that belongs to $\mathcal{L}(H) \triangle \mathcal{L}(M)$. [1]

The goal of the learning algorithm is to obtain an exact model of the unknown function. Note that, this model extends naturally to the case of deterministic Mealy machines and transducers by defining the membership queries to return the output of the transducer for the input string. We say that an algorithm gets *black box access* to an automaton/transducer when the algorithm is able to query the automaton with an input of his choice and obtain the result. No other information is obtained about the structure of the automaton.

III. Learning Algorithms

In this section we present two learning algorithms that form the basis of our constructions, Angluin's algorithm for DFA's [19] as optimized by Rivest and Schapire [20] and the Shabhaz-Groz (SG) algorithm for Mealy machines [12].

A. Angluin's Algorithm

Consider a finite automaton M. Angluin [19] suggested an algorithm (referred to as L^*) for learning M. The intuition behind the functionality of Angluin's algorithm is to construct the set of access and distinguishing strings given the two oracles available to it. Intuitively, the set of access strings will suggest the set of states of the reconstructed automaton. Furthermore, a transition from a state labeled with access string s to a state labelled with access string s' while consuming a symbol b will take place if and only if the string sb leads to a state that cannot be distinguished from s'.

In order to reconstruct the set of access and distinguishing strings the algorithm starts with the known set of access strings (initially just $\{\varepsilon\}$) and, using equivalence queries, expands the set of access and distinguishing strings until the whole automaton is reconstructed.

[1] We denote by $\triangle$ the symmetric difference operation.

Technical Description. The variant L^* we describe below is due to Rivest and Schapire [20]. The main data structure used by the L^* algorithm is the observation table.

Definition 4. An observation table OT with respect to an automaton M is a tuple $OT = (S, W, T)$ where

- $S \subseteq \Sigma^*$ is a set of access strings.
- $W \subseteq \Sigma^*$ is a set of distinguishing strings which we will also refer to as experiments.
- T is a partial function $T : \Sigma^* \times \Sigma^* \to \{0, 1\}$.

The function T maps strings into their respective state label in the target automaton, i.e., $T(s, d) = l(M[s \cdot d])$. We note here that T is defined only for those strings s, d such that $s \cdot d$ was queried using a membership query.

Next we define an equivalence relation between strings with respect to a set of strings and a finite automaton M.

Definition 5. (*Nerode Congruence*) Given a finite automaton M, for a set $W \subseteq \Sigma^*$ and two strings s_1, s_2 we say that

$$s_1 \equiv s_2 \bmod W$$

when for all $w \in W$ we have that $l(M[s_1 \cdot w]) = l(M[s_2 \cdot w])$.

Note that for any M there will be a finite number of different equivalence classes for any set W (this stems immediately from the fact that M is a finite automaton). This relates to the Myhill-Nerode theorem [21] that, for the above equivalence defined over a language L (i.e., requiring that either both $s_1 \cdot w, s_2 \cdot w \in L$ or none), it states that having a finite number of equivalence classes for L is equivalent to L being regular.

The observation table is going to give us a hypothesis automaton H when the property of *closedness* holds for the table.

Definition 6. Let $OT = (S, W, T)$ be an observation table. We say that OT is *closed* when, for all $t \in S \cdot \Sigma$, there exists $s \in S$ such that $t \equiv s \mod W$.

Given a closed observation table we can produce a hypothesis automaton as follows: For each string $s \in S$ we create a state q_s. The initial state is q_ε. For a state q_s and a symbol $b \in \Sigma$ we set $\delta(q_s, b) = q_t$ iff $s \cdot b \equiv t \mod W$. By the closedness property there will be always at least one such string. In the following, we will also see that by the way we fill the table that string will always be unique.

We are now ready to describe the algorithm: Initially we start with the observation table $OT = (S = \{\varepsilon\}, W = \{\varepsilon\}, T)$. The table T has $|\Sigma| + 1$ rows and is filled by querying an equal number of membership queries. The table is checked for closedness. If the table is not closed then let $t \in S \cdot \Sigma$ be a string such that for all $s \in S$, we have that $s \not\equiv t \mod W$. Then, we set $S = S \cup \{t\}$, complete remaining entries of the table via $|\Sigma|$ membership queries and we check again for closedness. Eventually the table becomes closed and we create a hypothesis automaton H. Observe that the number of times we will repeat the above process until we reach a closed table cannot exceed $|Q_M|$. A useful invariant in the above algorithmic process is the property of the observation table OT to be *reduced*: for all $s, s' \in S$ it holds that $s \neq s' \bmod W$. Observe that the initial OT is trivially reduced while augmenting the set S with a new state as described above preserves the property.

Now suppose that we have a hypothesis automaton H produced by a closed and reduced observation table. Given H, the algorithm makes an equivalence query and based on the outcome either the algorithm stops (no counterexample exists) or the counterexample z is processed and the set of distinguishing strings W is augmented by one element as shown below.

Processing a counterexample. For any $i \in \{0, \ldots, |z|\}$ define α_i to be the outcome (that is accept or reject) that is produced by processing the first i symbols of z with the hypothesis H and the remaining with M in the following manner. Given i we simulate H on the first i symbols of z to obtain a state $s_i \in S$. Let $z_{>i}$ be the suffix of z that is not processed yet; by submitting the membership query $s_i z_{>i}$ we obtain α_i. Observe that based on the fact that z is a counterexample it holds that $\alpha_0 \neq \alpha_{|z|}$. It follows that there exists some $i_0 \in \{0, \ldots, |z| - 1\}$ for which $\alpha_{i_0} \neq \alpha_{i_0+1}$. We can find such i_0 via a binary search using $O(\log |z|)$ membership queries. The new distinguishing string d will be defined as the suffix of $z_{>i_0}$ that excludes the first symbol b (denoted as $z_{>i_0+1}$). We observe the following: recall that α_{i_0} is the outcome of the membership query of $s_{i_0} z_{>i_0} = s_{i_0} b z_{>i_0+1}$ and α_{i_0+1} is the outcome of the membership query $s_{i_0+1} z_{>i_0+1}$. Furthermore, in H, s_{i_0} transitions to s_{i_0+1} by consuming b, hence we have that $s_{i_0} b \equiv s_{i_0+1} \bmod W$. By adding $d = z_{>i_0+1}$ to W we have that $T(s_{i_0} b, z_{>i_0+1}) \neq T(s_{i_0+1}, z_{>i_0+1})$ and hence the state s_{i_0+1} and the state that is derived by s_{i_0} consuming b should be distinct (while H pronounced them equal). We observe that the new observation table OT is not closed anymore: on the one hand, it holds that $s_{i_0} b \not\equiv s_{i_0+1} \bmod W \cup \{d\}$ (note that since $\varepsilon \in W$ it should be that $d \neq \varepsilon$), while if $s_{i_0} b \equiv s_j \bmod W \cup \{d\}$ for some $j \neq i_0 + 1$ this would imply that $s_{i_0} b \equiv s_j \bmod W$ and thus $s_{i_0+1} \equiv s_j \bmod W$ as well. This latter equality contradicts the property of the OT being reduced. Hence we conclude that the new OT is not closed and the algorithm continues as stated above (specifically it will introduce $s_{i_0} b$ as a new state in S and so on).

We remark that originally, L^* as described by Angluin added all prefixes of a counterexample in S and thus violated the reduced table invariant (something that lead to a sub-optimal number of membership queries). The variant of L^* we describe above due to [20] maintains the reduced invariant.

For a target automaton M with n states, the total number of membership queries required by the algorithm is bounded by $n^2(|\Sigma| + 1) + n \log m$ where m is the length of the longest counterexample.

B. The Shabhaz-Groz (SG) Algorithm

In [12], Shabhaz and Groz extended Angluin's algorithm to the setting of Mealy machines which are deterministic Transducers without ε-transitions.

The core of the algorithm remains the same: a table OT will be formed and as before will be based on rows corresponding to $S \cup S \times \Sigma$ and columns corresponding to distinguishing strings W. The table OT will not be a binary table in this case, but instead it will have values in Γ^*. Specifically, the partial function T in the SG observation table is defined as $T(s,d) = \mathrm{suff}(\mathcal{T}(sd), |d|)$. The rows of T satisfy the non-equivalence property, i.e., for any $s, s' \in S$ it holds that $s \not\equiv s' \mod W$, thus as in the Rivest-Schapire variant of L^* each access string corresponds to a unique state in the hypothesis automaton. Further, provided that $\Sigma \subseteq W$, we have for each $s \in S$, the availability of the output symbol produced when consuming any $b \in \Sigma$ is given by $T(s, b)$. In this way a hypothesis Mealy machine can be constructed in the same way as in the L^* algorithm. On the other hand, Shabhaz and Groz [12] contribute a new method for processing counterexamples described below.

Let z be a counterexample, i.e., it holds that the hypothesis machine H and the target machine produce a different output in Γ. Let s be the longest prefix of z that belongs to the access strings S. If $s \cdot d = z$, in [12] it is observed that they can add d as well as all of its suffixes as columns in OT. The idea is that at least one of the suffixes of d will contain a distinguishing string and thus it can be used to make the table not closed.In addition, this method of processing counterexamples makes the set W suffix closed. After adding all suffixes and making the corresponding membership queries, the algorithm proceeds like the L^* algorithm by checking the table for closedness. The overall query complexity of the algorithm is bounded by $O(|\Sigma|^2 n + |\Sigma| m n^2)$ queries, where n, m, Σ are defined as in the L^* algorithm.

IV. Learning Symbolic Automata

In this section we present our algorithm for learning symbolic finite automata for general predicate families. Then, we specialize our algorithm for the case of regular expression filters.

A. Main Algorithm

Symbolic finite automata extend classical finite automata by allowing transitions to be labelled by predicate formulas instead of single symbols. In this section we will describe the first, to the best of our knowledge, algorithm to infer SFAs from membership and equivalence queries. Our algorithm, contrary to previous efforts to infer symbolic automata [22] which required the counterexample to be of minimal length, works in the standard membership and equivalence query model under a natural assumption, that the guards themselves can be inferred using queries.

The main challenge in learning SFA's is that counterexamples may occur due to two distinct reasons: (i) a yet unlearned state in the target automaton (which is the only case in the L^* algorithm), (ii) a learned state with one of the guards being incorrect and thus, leading to a wrong transition into another already discovered state. Our main insight is that it is possible to distinguish between these two cases and suitably adjust either the guard or expand the hypothesis automaton with a new state.

Technical Description. The algorithm is parameterized by a predicate family $\mathcal{P}$ over Σ. The goal of the algorithm is to both infer the structure of the automaton and label each transition with the correct guard $\phi \in \mathcal{P}$. Compared to the L^* algorithm, our learning algorithm, on top of the ability to make membership and equivalence queries will also require that the guards come from a predicate family for which there exists a guard generator algorithm that we define below.

Definition 7. A guard generator algorithm $\mathrm{guardgen}()$ for a predicate family $\mathcal{P}$ over an alphabet Σ takes as input a sequence R of pairs (b, q) where $b \in \Sigma$ and q an arbitrary label and returns a set of pairs G of the form (ϕ, q) such that the following hold true:

- (Completeness) $\forall (b, q) \in R\ \exists \phi : (\phi, q) \in G \wedge \phi(b)$.
- (Uniqueness) $\forall \phi, \phi', q : (\phi, q), (\phi', q) \in G \rightarrow \phi = \phi'$.
- (Determinism) $\forall b \in \Sigma\ \exists! (\phi, q) \in G : \phi(b)$.

The algorithm fails if such set of pairs does not exist.

Given a predicate family $\mathcal{P}$ that is equipped with a guard generator algorithm, our SFA learning algorithm employs a special structure observation table $SOT = (S, W, \Lambda, T)$ so that the table T has labelled rows for each string in $S \cup \Lambda$ where $\Lambda \subseteq S \cdot \Sigma$. The initial table is $SOT = \{S = \{\varepsilon\}, W = \{\varepsilon\}, \Lambda = \emptyset, T\}$. Closedness of SOT is determined by checking that for all $s \in S$ it holds that $sb \in \Lambda \rightarrow \exists s' \in S : (sb \equiv s' \bmod W)$. Furthermore the table is reduced if and only if for all $s, s' \in S$ it holds that $s \not\equiv s' \bmod W$. Observe that the initial table is (trivially) closed and reduced.

Our algorithm operates as follows. At any given step, it will check T for closedness. If a table is not closed, i.e., there is a $sb \in \Lambda$ such that $sb \not\equiv s'$ for any $s' \in S$, the algorithm will add sb to the set of access strings S updating the table accordingly.

On the other hand, if the table is closed, a hypothesis SFA $H = (Q_H, q_\varepsilon, F, \mathcal{P}, \Delta)$ will be formed in the following way. For each $s \in S$ we define a state $q_s \in Q_H$. The initial state is q_ε. A state q_s is final iff $T(s, \varepsilon) = 1$. Next, we need to determine the move relation that contains triples of the form (q, ϕ, q') with $\phi \in \mathcal{P}$. The information provided by SOT for each q_s is the transitions determined by the rows $T(sb)$ for which it holds $sb \in \Lambda$. Using this we form the pairs $(b, q_{s'})$ such that $sb \equiv s' \bmod W$ (the existence of s' is guaranteed by the closedness property). We then feed those pairs to the $\mathrm{guardgen}()$ algorithm that returns a set G_{q_s} of pairs of the form (ϕ, q). We set $\mathrm{guard}(q_s) = \{\phi \mid (\phi, q) \in G_{q_s}\}$ and add the triple (q_s, ϕ, q) in Δ. Observe that by definition the above process when executed on the initial SOT returns as the hypothesis SFA a single state automaton with a self-loop marked with `true` as the single transition over the single state.

Processing Counterexamples. Assume now that we have a hypothesis SFA H which we submit to the equivalence oracle. In case H is correct we are done. Otherwise, we obtain a counterexample string z. First, as in the L^* algorithm, we perform a binary search that will identify some $i_0 \in \{0, 1, \ldots, |z| - 1\}$ for which the response of the target machine is different for the strings $s_{i_0} z_{>i_0}$ and $s_{i_0+1} z_{>i_0+1}$. This determines a new distinguishing string defined as $d = z_{>i_0+1}$. Notice that $s_{i_0} b \not\equiv s_{i_0+1} \bmod W \cup \{d\}$ something that reflects that s_{i_0} over b should not transition to s_{i_0+1} as the hypothesis has predicted. In case $s_{i_0} b \not\equiv s_j \bmod W \cup \{d\}$ for any j, the table will become not closed if augmented by d and thus the algorithm will proceed by adding d to W and update the table accordingly (this is the only case that occurs in the L^* algorithm). On the other hand, it may be the case that adding d to SOT preserves closedness as it may be that $s_{i_0}b \equiv s_j \bmod W \cup \{d\}$ for some $j \neq i_0 + 1$. This does not contradict the fact that the table prior to its augmentation was reduced, as in the case of the L^* algorithm, since the transition s_{i_0} to s_{i_0+1} when consuming b that is present in the hypothesis could have been the product of $\mathrm{guardgen}()$ and not an explicit transition defined in Λ. In such case Λ is augmented with $s_{i_0}b$ and the algorithm will issue another equivalence query, continuing in this fashion until the SOT becomes not closed or the hypothesis is correct.

The above state of affairs distinguishes our symbolic learning algorithm from learning via the L^* algorithm: not every equivalence query leads to the introduction of a new state. We observe though that some progress is still being made: if a new state is not discovered by an equivalence query, the set Λ will be augmented making a transition that was before implicit (defined via a predicate) now explicit. For suitable predicate families this augmentation will lead to more refined guard predicates which in turn will result to better hypothesis SFA's submitted to the equivalence oracle and ultimately to the reconstruction of an SFA for the target.

In order to establish formally the above we need to prove that the algorithm will converge to a correct SFA in a finite number of steps (note that the alphabet Σ may be infinite for a given target SFA and thus the expansion of Λ by each equivalence query is insufficient by itself to establish that the algorithm terminates).

Convergence can be shown for various combinations of predicate families $\mathcal{P}$ and $\mathrm{guardgen}()$ algorithms that relate to the ability of the $\mathrm{guardgen}()$ algorithm to learn guard predicates from the family $\mathcal{P}$. One such case is when $\mathrm{guardgen}()$ *learns predicates from $\mathcal{P}$ via counterexamples*. Let $\mathcal{G} \subseteq 2^{\mathcal{P}}$ a guard predicate family. Intuitively, the $\mathrm{guardgen}()$ algorithm operates on a training set containing actual transitions from a state that were previously discovered. Given the symbols labeling those transitions, the algorithm produces a candidate guard set for that state. If the training set is small the candidate guard set is bound to be wrong and a counterexample will exist. The $\mathrm{guardgen}()$ algorithm learns the guard set via counterexamples if by adding a counterexample in the training set in each iteration will eventually stabilize the output of the algorithm to the correct guard set. We will next define what a counterexample means with respect to the $\mathrm{guardgen}()$ algorithm, a set of predicates $\boldsymbol{\phi}$ and an input to $\mathrm{guardgen}()$ which is consistent with $\boldsymbol{\phi}$. Recall that inputs to $\mathrm{guardgen}()$ are sets R of the form (b, s_i) where b is a symbol and s_i is a label; a set R is consistent with $\boldsymbol{\phi}$ if it holds that $\phi_i(b)$ is true for all $(b, s_i) \in R$ (we assume a fixed correspondence between the labels s_i and the predicates ϕ_i of $\boldsymbol{\phi}$). A counterexample would be a pair (b^*, s^*) where s^* labels a predicate ϕ_j in $\boldsymbol{\phi}$ but the output predicate ϕ of $\mathrm{guardgen}()$ that is labelled by s_j disagrees with ϕ_j on symbol b^*. More formally we give the following definition.

Definition 8. For $k \in \mathbb{N}$, consider a set of predicates $\boldsymbol{\phi} = \{\phi_1, \ldots, \phi_k\} \in \mathcal{G}$ labelled by $\boldsymbol{s} = (s_1, \ldots, s_k)$ so that ϕ_i is labelled by s_i and a sequence of samples R containing pairs of the form (b, s_i) where $\phi_i(b)$ for some $i \in [k]$. A *counterexample* (b^*, s^*) for $(R, \boldsymbol{\phi}, \boldsymbol{s})$ w.r.t. $\mathrm{guardgen}()$ is a pair such that if $G = \mathrm{guardgen}(R)$ it holds that there is a $j \in \{1, \ldots, k\}$ with $s_j = s^*$, $(\phi, s_j) \in G$ and $\phi(b^*) \neq \phi_j(b^*)$.

Let t be a function of k. A guard predicate family $\mathcal{G}$ is t-learnable via counterexamples if it has a $\mathrm{guardgen}()$ algorithm such that for any $\boldsymbol{\phi} = (\phi_1, \ldots, \phi_k) \in \mathcal{G}$ labelled by $\boldsymbol{s} = (s_1, \ldots, s_k)$, it holds that the sequence $R_0 = \emptyset$, $R_i = A_i \cup R_{i-1}$ where A_i is a singleton containing a counterexample for $(R_{i-1}, \boldsymbol{\phi}, \boldsymbol{s})$ w.r.t. $\mathrm{guardgen}()$ (or empty if none exist), satisfies that $\mathrm{guardgen}(R_j) = \{(\phi_i, s_i) \mid i = 1, \ldots, k\}$ for any $j \geq t$. In other words, a guard predicate family is t-learnable if the $\mathrm{guardgen}()$ converges to the target guard set in t iterations when in each iteration the training set is augmented with a counterexample from the previous guard set.

We are now ready to prove the correctness of our SFA learning algorithm.

Theorem 1. *Consider a guard predicate family $\mathcal{G}$ that is t-learnable via counterexamples using a $\mathrm{guardgen}()$ algorithm. The class of deterministic symbolic finite state automata with guards from $\mathcal{G}$ can be learned in the membership and equivalence query model using at most $O(n(\log m + n)t(k))$ queries, where n is size of the minimal SFA for the target language, m is the maximum length of a counterexample, and k is the maximum outdegree of any state in the minimal SFA of the target language.*

In appendix D we describe an example of a $\mathrm{guardgen}()$ algorithm when SFAs are used to model decision trees.

B. A Learning Algorithm for RE Filters

Consider the SFA depicted in figure 1 for the regular expression $(.)^*$<a>$(.)^*$. This represents a typical regular expression filter automaton where a specific malicious string is matched and at that point any string containing that malicious substring is accepted and labeled as malicious. When testing regular expression filters many times we would have to test different character encodings. Thus, if we assume that the alphabet Σ is the set of two byte chatacter sequences as it would be in UTF-16, then each state would have 2^{16} different transitions, making traditional learning algorithms too inefficient, while we point out that the full unicode standard contains around 110000 characters.

We will now describe a guard generator algorithm and demonstrate that it efficiently learns predicates resulting from regular expressions. The predicate family used by our algorithm is $\mathcal{P} = 2^{\Sigma}$ where Σ is the alphabet of the automaton, for example UTF-16. The guard predicate family $\mathcal{G}_{l,k}$ is parameterized by integers l, k and contains vectors of the form $\langle \phi_1, \ldots, \phi_{k'} \rangle$ with $k' \leq k$, so that $\phi_i \in \mathcal{P}$ and[2] $|\phi_i| \leq l$ for any i, except for one, say j, for which it holds that $\phi_j = \neg(\vee_{i \neq j} \phi_i)$. The main intuition behind this algorithm is that, for each state all but one transitions contain a limited number of symbols, while the remaining symbols are grouped into a single (sink) transition.

In an SFA over $\mathcal{G}_{l,k}$, a transition (q, ϕ, q') is called *normal* if $|\phi| \leq l$. A transition that is not normal is called a *sink* transition. Our algorithm updates transitions *lazily* with new

[2] We use the notation $|\phi| = |\{b \mid \phi(b) = 1\}|$.

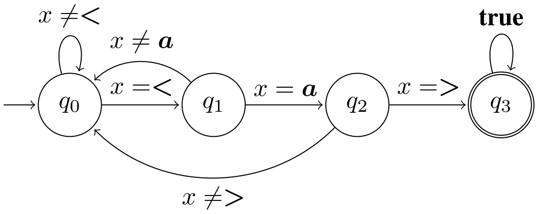

Fig. 1. SFA for regular expression (.)*<a>(.)*.

symbols whenever a counterexample shows that a symbol belongs to a different transition, while the transition with the largest size is assigned as the sink transition.

Consider R, an input sequence for the guard generator algorithm. We define $R_q = \{(b, q) \mid (b, q) \in R\}$. If $|R_q| \leq l$ then we define the predicate for R_q denoted by ϕ_q. Let q' be such that $|R_{q'}| \geq |R_q|$ for all q. We define $\sigma = \Sigma^* \setminus \cup_{q \neq q'} R_q$. The output is the set $G = \{(\phi_q, q) \mid q \neq q'\} \cup \{(\sigma, q')\}$. In case $R = \emptyset$ the algorithm returns Σ^* as the single predicate.

We observe now that $\mathcal{G}_{l,k}$ is t-learnable via counterexamples with $t = O(lk)$. Indeed, note that counterexamples will be augmenting the cardinality of the predicates that are constructed by the guard generator. At some point one predicate will exceed l elements and will correctly be identified as the sink transition. We conclude that the target SFA will be inferred using $O(nlk(\log m + n))$ queries.

V. Learning Transducers

In this section we present our learning algorithms for transducers. We start with our improved algorithm for Mealy machines and then we move to single-valued transducers with bounded lookahead. We conclude with how to extend our results to the symbolic transducer setting. To motivate this section we present in Figure 5 three examples of common string manipulating functions. For succinctness we present the symbolic versions of all three sanitizers. The first example is a typical `tolowercase` function which converts uppercase ascii letters to lowercase and leaves intact any other part of the input. The second example is a simplified HTML Encoder which only encodes the character "<". In this case, the transition reading the input symbol "<" needs to produce multiple output symbols that represent the encoded version of the symbol. An equivalent formulation of this property is to assume that the resulting Mealy machine is deterministic but allow ε-transitions. This transformation is not expressible with a Mealy machine which requires that only one output symbol will be produced for each input symbol consumed. Finally, the third sanitizer is a transformation function used by mod-security, a popular web application firewall, in order to remove comments from an SQL expression. This helps to deobfuscate the input before passing it through regular expression filters. In this case, to match the beggining of an SQL comment, i.e. the string "/*", the transducer need to employ an 1-lookahead. This transformation can only be modelled using non determinism in the resulting finite state transducer model. In the learning algorithms of this section, we will replace membership queries with transduction queries that output the result of the transduction of the input string.

A. *Improved learning of Mealy machines*

In this section we describe two improvements of the SG algorithm for Mealy machines. In the first one we provide an efficiency improvement over SG on the number of transduction queries required in order to learn a target Mealy machine of size n. Specifically we drop the counterexample processing complexity from $O(m \cdot n)$ to $O(m + \log n)$ where m is the length of the counterexample. Our main observation is that contrary to what is implied by Shabaz and Groz, processing Mealy machine counterexamples can take advantage of the binary-search counter example processing similar to Rivest-Schapire's version of the L^* algorithm something that leads to major improvements in the query complexity of the algorithm. In our second improvement we show how the learning algorithm can handle a more general class of Mealy Machines which are deterministic but also allow ε-transitions in the input. In practice, this modification allows for multiple symbols in the output to be produced for each single input symbol. This case is particularly relevant to our setting as such Mealy machines are very frequently encountered in practice notably as string encoders such *url* and *HTML encoders*, cf. Figure 5.

Improved Counterexample Processing: We now introduce a new way of handling counterexamples in the SG algorithm that is based on Rivest and Schapire's version of the L^* algorithm [20]. Recall that in the SG algorithm all the suffixes of a counterexample are added as new experiments in the table and therefore, in the worst case, $O(m \cdot n)$ new entries must be filled in the table using transduction queries where m is the length of the counterexample and n is the number of access strings.

Our improved counterexample processing operates as follows. Suppose that z is the given counterexample, i.e. it is a string where the target machine and the hypothesis disagree. Furthermore suppose that the hypothesis transducer is produced by a reduced observation table. We notice that even though the last state reached in the counterexample may be identical in both cases, we can find a point where a wrong state is traversed by the counterexample by inspecting the transduction of z. Indeed, there exists a (smallest) index i such that $\mathcal{T}_H(z)_i \neq \mathcal{T}_M(z)_i$. Therefore we can conclude that $z_{<i}$ reaches different states in the hypothesis and target machine. It follows we can trim the counterexample to $z' = z_{\leq i}$ and this way we know that the last symbol produced by the counterexample is wrong in the hypothesis automaton.

We now describe formally our improved counterexample processing algorithm. For any $j \in \{0, \ldots, |z'|\}$ let γ_j be a string that is produced as follows: first run the hypothesis H machine on $z'_{\leq j}$ to obtain γ^H_j; the hypothesis terminates on a state s_j; subsequently submit $s_j z'_{>j}$ to M in order to obtain a string γ^M_j. Let $\gamma_j = \gamma^H_j \cdot \mathrm{suff}(\gamma^M_j, |z'| - j)$ and observe that $\gamma_0 = \mathcal{T}_M(z')$, $\gamma_{|z'|} = \mathcal{T}_H(z')$ and $\gamma_0 \neq \gamma_{|z'|}$.

The binary search then is performed in this fashion. The initial range is $[0, |z'|]$ and the middle point is $j = \lceil |z'|/2 \rceil$. Given a range $[j_{\mathsf{left}}, j_{\mathsf{right}}]$ and a middle point position j, we check whether $\gamma_j = \gamma_0$; if this is the case we set the new range as $[j, j_{\mathsf{right}}]$ else we set the new range as $[j_{\mathsf{left}}, j - 1]$ and we continue recursively. The process finishes when the range is a singleton $[j_0, j_0]$ which is the output of the search.

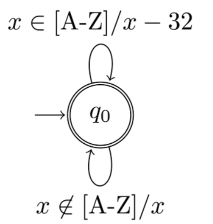

Fig. 2. `ToLowerCase` function. Mealy machine.

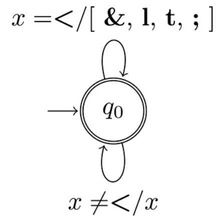

Fig. 3. Simplified version of HTML Encoder function. Deterministic Transducer with multiple output symbols per transition.

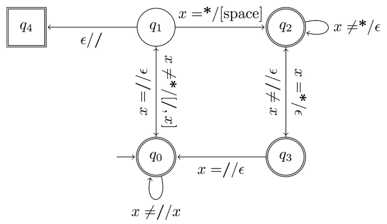

Fig. 4. `ReplaceComments` Mod-security transformation function. Non deterministic Transducer with ϵ transitions and 1-lookhead.

Fig. 5. Three different sanitizers implementing widely used functions and their respective features when modeled as transducers. Only the first sanitizer can be inferred using existing algorithms.

Theorem 2. *The binary search process described above returns $j_0 \in \{0, \ldots, |z'|-1\}$ such that $\gamma_{j_0} \neq \gamma_{j_0+1}$.*

Given such j_0, we observe that since the prefixes of $\gamma_{j_0}, \gamma_{j_0+1}$ that correspond to the processing of $z_{\leq j_0}$ are identical by definition, the difference between the strings should lie in their suffixes. Furthermore, $(\gamma_{j_0})_{j_0+1} = (\gamma_{j_0+1})_{j_0+1}$ since the former is the last output symbol produced by H when consuming $z_{\leq j_0}b$ and the latter is the last symbol produced by M when consuming $s_{j_0}b$, where $b = z'_{j_0+1}$ is the (j_0+1)-th symbol of the counterexample. As a result the difference of $\gamma_{j_0}, \gamma_{j_0+1}$ is in their $(|z'|-j_0-1)$-suffixes that by definition are equal to the same length suffixes of $\gamma^M_{j_0}, \gamma^M_{j_0+1}$. This implies that $j_0 < |z'|-1$ and thus we can define a new distinguishing string $d = z'_{>j_0+1}$. The observation table augmented by this new string d is not closed any more: the string $s_{j_0}bd = s_{j_0}z'_{>j_0}$ when queried to M produces the string $\gamma^M_{j_0}$ which disagrees in its $|d|$-suffix with the string $\gamma^M_{j_0+1}$ produced by M on input $s_{j_0+1}d$. Closing the table will now introduce the new access string s_jb and hence the algorithm continues by expanding the hypothesis machine.

The approach we outlined above offers a significant efficiency improvement over the SG algorithm. Performing the binary search detailed above requires merely $O(\log m)$ queries where m is the length of the counterexample. This gives a total of $O(n + \log m)$ queries for processing a counterexample as opposed to the $O(n \cdot m)$ of the SG algorithm where n is the number of access strings in the observation table.

Handling ε-transitions: We next show how to tackle the problem of a Mealy machine that takes ε-transitions but still is deterministic in its output. The effect of such ε-transitions is that many or no output symbols may be generated due to a single input symbol. Even though this is a small generalization it complicates the learning process. First, if more than one output symbols are produced for each input symbol our counterexample processing method will fail because the breakpoint output symbol $(\mathcal{T}_M(z))_i$ may be produced by less than i symbols of z. Further, in the observation table, bookkeeping will be inaccurate since, if we keep only the $\text{suff}(\mathcal{T}_M(sd), |d|)$ string in each table entry, then this might not correspond to the output symbols that correspond to last d symbols of the input string.

We show next how to suitably modify our bookkeeping and counterexample processing so that Mealy machines with ε-transitions are handled.

- Instead of keeping in each table entry the string $\text{suff}(\mathcal{T}_M(sd), |d|)$ we only keep the output that corresponds to the experiment d. While in standard Mealy machines this is simply $\text{suff}(\mathcal{T}_M(sd), |d|)$, when ε-transitions are used the output may be longer or shorter. Therefore, we compute the output of the experiment as the substring of $\mathcal{T}_M(sd)$ when we subtract the longest common prefix with the string $\mathcal{T}_M(s)$. Intuitively, we keep only the part of the output that is produced by the experiment d. Given that we do not know the length of that output we subtract the output produced by the access string s. Notice that, because the observation table is prefix closed, we can obtain the output $\mathcal{T}_M(s)$ without making an additional transduction query to the target M.
- When processing a counterexample, the method we outlined above can still be used. However, as we mentioned, the index i where the output of the hypothesis and the target machine differ may not be the correct index in which we must trim the input at. Specifically, if $\mathcal{T}_H(z)$ and $\mathcal{T}_M(z)$ differ in position i (and i is the smallest such position), then we are looking for an index $i' \leq i$ such that $\mathcal{T}_M(z_{\leq i'}) = \mathcal{T}_M(z)_{\leq i}$. Given i, such a position i' can be found with $\log |z|$ queries using a binary search on the length of the output of each substring of z. We will then define $z' = z_{\leq i'}$.

Given the above modifications we will seek j_0 via a binary search as in Theorem 2 but using the strings γ_j that are defined as $\gamma^H_j \cdot \text{suff}(\gamma^M_j, |\gamma^M_j| - j')$ where $j' = |\mathcal{T}_M(s_j)|$ for $j = 0, \ldots, |z'|$. Then, the same proof as in Theorem 2 applies. Further, using a similar logic as before we argue that the string $d = z_{>j_0+1}$ is non-empty and it can be used as a new distinguishing string. The asymptotic complexity of the algorithm will remain the same.

B. Learning Transducers with Bounded Lookahead

It is easy to see that if the target machine is a single-valued non-deterministic transducer with the bounded lookahead property the algorithm of the previous section fails. In fact the algorithm may not even perform any progress beyond the initial single state hypothesis even if the number of states of the target is unbounded; for instance, consider a transducer that modifies only a certain input symbol sequence w (say by redacting its first symbol) while leaving the remaining input intact. The algorithm of the previous section will form a hypothesis that models the identity function and obtain from the equivalence oracle, say, the string w as the counterexample (any string containing w would be a counterexample, but w is the shortest one). The binary search process will identify $j_0 = 0$ (it is the only possibility) and will lead the algorithm to the adoption of $d = w_{>1}$ as the distinguishing string. However, $\mathcal{T}_M(s_{j_0}bd) = \mathcal{T}_M(w) = w_{>1}$, and also $\mathcal{T}_M(s_{j_0+1}d) = w_{>1}$ hence d is not distinguishing: $s_{j_0}b \equiv s_{j_0+1} \bmod W \cup \{d\}$. At this moment the algorithm is stuck: the table remains closed and no progress can be made. For the following we assume that the domain of the target transducer is Σ^*, i.e. for every string $\alpha \in \Sigma^*$ there exists *exactly* one $\gamma \in \Gamma^*$ such that $\mathcal{T}_M(\alpha) = \gamma$.

Technical Description. The algorithm we present builds on our algorithm of the previous section for Mealy Machines with ε-transitions. Our algorithm views the single-valued transducer as a Mealy Machine with ε-transitions augmented with certain lookahead paths. As in the previous section we use an observation table OT that has rows on $S \cup S \times \Sigma$ and columns corresponding to the distinguishing strings W. In addition our algorithm holds a *l*ookahead list L of quadraples $(src, dst, \alpha, \gamma)$ where src, dst are index numbers of rows in the OT, $\alpha \in \Sigma^*$ is the input string consumed by the lookahead path, while $\gamma \in \Gamma^*$ is the output produced by the lookahead path. Whenever a lookahead path is detected, it is added in the lookahead transition list L. Our algorithm will also utilize the concept of a prefix-closed membership query: In a prefix closed membership query, the input is a string s and the result is the set of membership queries for all the prefixes of s. Thus, if O is the membership oracle, then a prefix-closed membership query on input a string s will return $\{O(s_{\leq 1}), \dots, O(s)\}$. We will now describe the necessary modifications in order to detect and process lookahead transitions.

Detecting and Processing lookahead transitions. Observe that in a deterministic transducer the result of a prefix-closed query on a string s would be a prefix closed set $r_1, \dots, r_t$. The existence of $i_0 \in \{1, \dots, t\}$ with r_{i_0} *not a strict prefix* of r_{i_0+1} suggests that a lookahead transition was followed. Let r_{j_0} be the longest common prefix of $r_1, \dots, r_{i_0+1}$. The state $src = s_{j_0}$ that corresponds to q_{j_0} is the state that the lookahead path commences while the state $dst = s_{i_0+1}$ that corresponds to input q_{i_0+1} is the state the path terminates. The path consumes the string α that is determined by the suffix of q_{i_0+1} starting at the (j_0+1)-position. The output of the path is $\gamma = \text{suff}(r_{i_0+1}, |r_{i_0+1}| - |r_{j_0}|)$.

The algorithm proceeds like the algorithm for Mealy machines with ε-transitions. However, all membership queries are replaced with prefix-closed membership queries. Every query is checked for a lookahead transition. In case a lookahead transition is found, it is checked if it is already in the list L. In the opposite case the quadraple $(src, dst, \alpha, \gamma)$ is added in L and all suffixes of α are added as columns in the observation table. The reason for the last step is that every lookahead path of length m defines $m - 2$ final states in the single-valued transducer. The suffixes of α can be used to distinguish these states. Finally, when the table is closed, a hypothesis is generated as before taking care to add the respective lookahead transitions, removing any other transitions which would break the single-valuedness of the transducer.

Processing Counterexamples. For simplicity, in this algorith we utilize the Shabaz-Groz counterexample processing method. We leave the adjustment of our previous binary search counterexample method as future work. Notice that, a counterexample may occur either due to a hidden state or due to a yet undiscovered lookahead transition. We process a counterexample string as follows: We follow the counterexample processing method of Shabaz Groz and we add all the suffixes of the counterexample string as columns in the OT. Since the SG method already adds all suffixes, this also covers our lookahead path processing. In case we detect a lookahead we also take care to add the respective transition in the lookahead list L. Notice that, following the same argument as in the analysis of the SG algorithm, one of the suffixes will be distinguishing, thus the table will become not closed and progress will be made.

Regarding the correctness and complexity of our algorithm we prove the following theorem.

Theorem 3. *The class of non-deterministic single-valued transducers with the bounded lookahead property and domain Σ^* can be learned in the membership and equivalence query model using at most $O(|\Sigma|n(mn+|\Sigma|+kn)(n+\max\{m,n\}))$ membership queries and at most $n + k$ equivalence queries where m is the length of the longest counterexample, n is the number of states and k is the number of lookahead paths in the target transducer.*

C. Learning Symbolic Finite Transducers

The algorithm for inferring SFAs can be extended naturally in order to infer SFTs. Due to space constraints we won't describe the full algorithm here rather sketch certain aspects of the algorithm.

The main difference between the SFA algorithm and the SFT algorithm is that on top of inferring predicates guards, the learning algorithm for SFTs need to also infer the term functions that are used to generate the output of each transition. This implies that there might be more than one transition from a state s_i to a state s_j due to differences in the term functions of each transition. This scenario never occurs in the case of SFAs. Thus, the `guardgen()` algorithm on an SFT inference algorithm should also employ a `termgen()` algorithm which will work as a submodule of `guardgen()` in order to generate the term functions for each transition and possibly split a predicate guard into more.

Finally, we point out that in our implementation we utilized a simple SFT learning algorithm which is a direct extension of our RE filter learning algorithm in the sense that we generalize the pair (predicate, term) with the most members to become the sink transition for each state.

VI. Implementing an Equivalence Oracle

In practice a membership oracle is usually easy to obtain as the only requirement is to be able to query the target filter or sanitizer and inspect the output. However, simulating an equivalence oracle is not trivial. A straightforward approach is to perform random testing in order to find a counterexample and declare the machines equal if a counterexample is not found after a number of queries. Although this is a feasible approach, it requires a very large number of membership queries.

Taking advantage of our setting, in this section we will introduce an alternative approach where an equivalence oracle is implemented using just a single membership query. To illustrate our method consider a scenario where an auditor is remotely testing a filter or a sanitizer. For that purpose the auditor is in possession of a set of attack strings given as a context free grammar (CFG).

The goal of the auditor is to either find an attack-string bypassing the filter or declare that no such string exists and obtain a model of the filter for further analysis. In the latter case, the auditor may work in a whitebox fashion and find new attack-strings bypassing the inferred filter, which can be used to either obtain a counterexample and further refine the model of the filter or actually produce an attack. Since performing whitebox testing on a filter is much easier than black-box, even if no attack is found the auditor has obtained information on the structure of the filter.

Formally, we define the problem of Grammar Oriented Filter Auditing as follows:

Definition 9. In the grammar oriented filter auditing problem (GOFA), the input is a context free grammar G and a membership oracle for a target DFA F. The goal is to find $s \in G$, such that $s \notin F$ or determine that no such s exists.

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. [23].

Our algorithm for the GOFA problem uses a learning algorithm for SFAs utilizing Algorithm 1 as an equivalence oracle. 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 $\mathcal{L}(G \cap \neg H) = \emptyset$, this does not imply that $\mathcal{L}(G \cap \neg F) = \emptyset$.

Algorithm 1 GOFA Algorithm

```
Require: Context Free Grammar $G$, membership oracle $O$

  function EQUIVALENCE ORACLE($H$)
      $G_A \leftarrow G \cap \neg H$
      if $\mathcal{L}(G_A) = \emptyset$ then
          return $Done$
      else
          $s \leftarrow \mathcal{L}(G_A)$
          if $O(s) = True$ then
              return $Counterexample, s$
          else
              return $Attack, s$
          end if
      end if
  end function
```

	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 I. SFA VS. DFA LEARNING

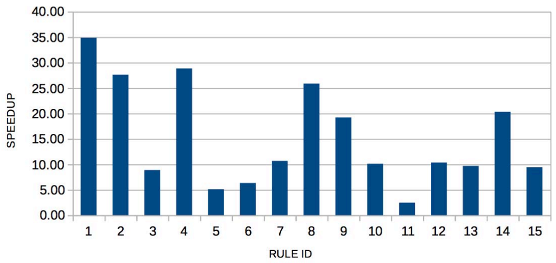

Fig. 6. Speedup of SFA vs. DFA learning.

Adaptation to sanitizers. The technique above can be generilized 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.

In appendix A, B we describe a comparison of the GOFA algorithm with random testing as well as ways in which an complete equivalence oracle may be implemented.

VII. EVALUATION

A. Implementation

We have implemented all the algorithms described in the previous sections. In order to evaluate our DFA/SFA learning algorithms in the standard membership/equivalence query model we implemented an equivalence oracle by computing

	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 II. SFA VS. DFA LEARNING + GOFA

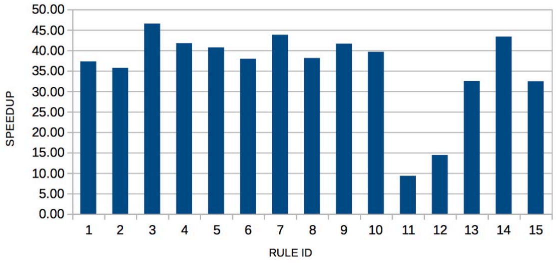

Fig. 7. Speedup of SFA vs. DFA learning with GOFA.

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 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 [24] 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 [25]. In order to convert the inferred models to BEK programs we used the algorithm described in appendix C.

B. 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 [26] and PHPIDS [27] web application firewalls. These systems contain well designed, complex regular expressions rulesets that attempt to protect against vulnerability classes 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 targetting XSS and SQL injection vulnerabilities. We chose the filter 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. Our sanitizer testbed is described in detail in section VII-E. Finally, for testing our GOFA and filter fingerprinting algorithms we also incorporated two additional WAF implementations, Web Knight and Web Castelum and Microsoft's urlscan with a popular set of SQL Injection rules [28]. 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.

C. Evaluation of DFA/SFA Learning algorithms

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 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 I 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 showin in Figure 6. 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.

D. Evaluation of 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 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 separetely.

DFA model generation evaluation. Intuitevely, the GOFA algorithm is efficient in recovering a model for the target filter if the algorithm is in possesion 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 II we present the numerical results of our experiments over the same set of filters used in the experiments of Section VII-C. 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 7. We also point out the even though the DFA algorithm checks all transitions of the automaton explicitily (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 more expressive 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.

For our evaluation we targetted SQL Injection vulnerabilities. In our first experiment we utilized five well known web application firewalls and used as an input grammar an SQL grammar from the yaxx project [29]. In this experiment the input filter was running on live firewall installations rather than on the extracted rules. We checked whether there were valid SQL statements that one could pass through the web application firewalls.

The results of this experiment can be found in table IV. We found that in all cases a user can craft a valid SQL statement that will bypass the rules of all five firewalls. For the first 4 products where more complex rules are used the simple statement “open a” is not flagged as malicious. This statement allows the execution of statements saved in the database system before using a “DECLARE CURSOR” statement. Thus, these attacks could be part of an attack which reexecutes a statement already in the database in a return oriented programming manner.

The open statement was flagged malicious by urlscan, in which case GOFA succesfully detected that and found an alternative vector, “replace”. We also notice, that using GOFA with the SFA learning algorithm makes a minimum number of queries since our SFA algorithm adds new edges to the automaton only lazily to update the previous models, thus making GOFA a compelling option to use in practice.

In the second experiment we performed we tested what will happen if we have a much more constrained grammar against the composition of two rules targetting SQL Injection attacks from PHPIDS. In order to achieve that we started with a small grammar which contains the combination of some attack vectors and, whenever a vector is identified bypassing the filter, 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. The succssive vectors we used as input grammar can be found in full version of the paper. The results of the experiment can be found in table IV. To check whether a vulnerability exists in the filter we computed the symmetric difference between the input grammar and the targetted 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 as well, GOFA was succesfull 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 succesfull attack vector was detected. Against we notice that the speedup of using the SFA algorithm is huge.

To conclude with the evaluation of the GOFA algorithm, although as we already discussed in section VI, the GOFA algorithm is necessarily either incomplete or inefficient in the worst case, it performs well in practice detecting both vulnerabilities when they exist and inferring a large part of the targetted filter when it is not able to detect a vulnerability.

E. 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 [8]. 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 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 [30] 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 infrastrucure. 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 input string.

In our algorithm, we used a simple form of symbolic transducer learning, as sketched in section V-C, where we gen-

	GRAMMAR		DFA LEARNING			SFA LEARNING				VULNERABILITY	
ID	STATES	ARCS	FOUND STATES	MEMBERSHIP	EQUIVALENCE	FOUND STATES	MEMBERSHIP	EQUIVALENCE	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 III. Bypasses detected by succesively reducing the attack grammar size for RE rules PHPIDS 76 & 52 composed

WAF	DFA LEARNING			SFA LEARNING				VULNERABILITY	
Target	FOUND STATES	MEMBERSHIP	EQUIVALENCE	FOUND STATES	MEMBERSHIP	EQUIVALENCE	SPEEDUP	EXISTS	FOUND
PHPIDS 0.7	2	186	1	0	3	1	46.75	TRUE	open a
MODSECURITY 2.2.9	1	186	1	0	3	1	46.75	TRUE	open a
WEBCASTELLUM 1.8.3	1	94	1	0	3	1	23.75	TRUE	open a
WEBKNIGHT 4.2	1	94	1	0	3	1	23.75	TRUE	open a
URLSCAN Common Rules	4	1835	2	5	40	2	43.73	TRUE	rollback work
						AVG=	36.94		

TABLE IV. Running the GOFA algorithm with an SQL grammar on common web applications firewalls

eralized the most commonly seen output term to all alphabet members not explicitily checked.

As an alphabet, we used a subset of characters including standard characters that should be encoded under the HTML stnadard 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 8 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 representation &#8249;. 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 rencode 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.

Another suprising 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 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 the two sanitizers 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.

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

Fig. 8. Equivalence Checking of HTML encoder implementations.

VIII. Related work

Our work is mainly motivated by recent advances in the analysis of sanitizers and regular expressions, a line of work which was initiated with the introduction of symbolic automata [11], although similar constructions were suggested much earlier [31]. The BEK language was introduced by Hooimeijer et al. [8] and the theory behind symbolic finite state transducers was extended in a follow up paper [15]. Symbolic automata, transducers and the BEK language is a very active area of research [14], [32]–[35] and we expect that BEK programs will get more widespread adoption in the near future. In the inference of symbolic automata and transducers there are two relevant recent works. Botincan and Babic [36] 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 Veanes [37] shows that equivalence of SLTs is in fact undecidable. Moreover, although [36] implements a symbolic version of Angluin's algorithm, in their system the predicates are obtained through symbolic execution, and therefore, there is no need to infer the predicate guards or infer the correct transitions for each state. Since their system is using the Shabaz-Groz algorithm, our improved counterexample processing would provide an exponentially faster way to handle counterexamples in their case too.

The second closely related work in the inference of symbolic automata was done by Maller and Mens [22].They describe an algorithm to infer automata over ordered alphabets which is a specific instantiation of symbolic automata. However, in order to correctly infer such an automaton the authors assume that the counterexample given by the equivalence oracle is of minimal length and this assumption is used in order to distinguish between a wrong transition in the hypothesis or a hidden state. Unfortunately, verifying that a counterexample is minimal requires an exponential number of queries and thus this assumption does not lead to a practical algorithm for inferring symbolic automata. On the other hand, our algorithm is more general, as it works for any kind of predicate guards as long as they are learnable, and moreover does not assume a minimal length counterexample making the algorithm practical.

The work on active learning of DFAs was initiated by Angluin [19] after a negative result of Gold [38] who showed that it is NP-Hard to infer the minimal automaton consistent with a set of samples. After its introduction, Anlguin's algorithm was improved and many variatons were introduced; Rivest and Schapire [20] showed how to improve the query complexity of the algorithm and introduced the binary search method for processing counterexamples. Balcazar et al. [39] describe a general approach to view the different variations of Angluin's algorithm.

Shabaz and Groz [12] extended Angluin's algorithm to handle Mealy Machines and introduced the counterexamlpe processing we discussed above. Their approach was then extended by Khalili and Tacchella [40] to handle non deterministic Mealy Machines. However, as we point out above mealy machines in general are not expressive enough to model complex sanitization functions. Moreover, the algorithm by Khalili and Tacchella uses the Shabaz-Groz counterexample processing thus it can be improved using our method. Since Shabaz-Groz is used in many contexts including the reverse engineering of Command and Control servers of botnets [41], we believe that our improved counterexample processing method will find many applications. Lately, inference techniques were developed for more complex classes of automata such as register automata [42]. These automata are allowed to use a finite number of registers [43]. Since registers were also used in some case during the analysis of sanitizer functions [15], and specifically decoders, we believe that expanding our work to handle register versions of symbolic automata and transducers is a very interesting direction for future work.

The implementation of our equivalence oracle is inspired by the work of Peled et al. [23]. In their work, a similar equivalence oracle implementation is described for checking Buichi automata, however, their implentation also utilizes the Vasileski-Chow algorithm [44], 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 inpractical for real applications. On the other hand, we demonstrate that our GOFA algorithm is able to infer 90% of the states of the target filter on average.

The algorithm for initializing the observation table was first described by Groce et al. [45]. In their paper they describe the initialization procedure and prove two lemmas regarding the efficiency of the procedure in the context of their model checking algorithm. However, the lemma proved just shows convergence and they are not concerned with the reduction of equivalence queries as we prove.

There is a large body of work regarding whitebox program analysis techniques that aim at validating the security of sanitizer code. The SANER [4] project uses static and dynamic analysis to create finite state transducers which are overapproximations of the sanitizer functions of programs. Minamide [5] 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 [6] attempts to generate high coverage test cases with symbolic execution for systems that use complex regular expressions. Wasserman and Su [7] 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.

IX. Conclusions and Future Work

Clearly, we are light of need for robust and complete black-box analysis algorithms for filter programs. In this paper we presented a first set of algorithms which could be utilized to analyze such programs. However, the space for research in this area is still vast. We believe that our algorithms can be further tuned in order to achieve an even larger performance increase. Moreover, more complex automata model which are currently being used [14], [43] can be also utilized to further reduce the number of queries required to infer a sanitizer model. Finally, we point out that totally different models might be necessary to handle other types of filters programs which are based on big data analytics or on the analysis of network protocols. Thus, to conclude we believe that black-box analysis of filters and sanitizers presents a fruitful research area which deserves more attention due to both scientific interest and practical applications.

Acknowledgements

This work was supported by the Office of Naval Research (ONR) through contract N00014-12-1-0166. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government or ONR.

References

[1] D. L. Eduardo Vela, “Our favorite xss filters/ids and how to attack them,” in *Black Hat Briefings*, 2009.

[2] D. Evteev, “Methods to bypass a web application methods to bypass a web application firewall.” http://ptsecurity.com/download/PT-devteev-CC-WAF-ENG.pdf.

[3] S. Esser, "Web application firewall bypasses and php exploits -rss'09 november 2009." http://www.suspekt.org/downloads/RSS09-WebApplicationFirewallBypassesAndPHPExploits.pdf.

[4] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Saner: Composing static and dynamic analysis to validate sanitization in web applications," in *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pp. 387–401, IEEE, 2008.

[5] Y. Minamide, "Static approximation of dynamically generated web pages," in *Proceedings of the 14th international conference on World Wide Web*, pp. 432–441, ACM, 2005.

[6] N. Li, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, "Reggae: Automated test generation for programs using complex regular expressions," in *Automated Software Engineering, 2009. ASE'09. 24th IEEE/ACM International Conference on*, pp. 515–519, IEEE, 2009.

[7] G. Wassermann and Z. Su, "Sound and precise analysis of web applications for injection vulnerabilities," in *ACM Sigplan Notices*, vol. 42, pp. 32–41, ACM, 2007.

[8] P. Hooimeijer, P. Saxena, B. Livshits, M. Veanes, and D. Molnar, "Fast and precise sanitizer analysis with bek," in *In 20th USENIX Security Symposium*, 2011.

[9] D. Bates, A. Barth, and C. Jackson, "Regular expressions considered harmful in client-side xss filters," in *Proceedings of the 19th international conference on World wide web*, pp. 91–100, ACM, 2010.

[10] "Programming languages used in most popular websites." https://en.wikipedia.org/wiki/Programming_languages_used_in_most_popular_websites. Accessed: 2015-11-10.

[11] M. Veanes, P. d. Halleux, and N. Tillmann, "Rex: Symbolic regular expression explorer," in *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ICST '10, (Washington, DC, USA), pp. 498–507, IEEE Computer Society, 2010.

[12] M. Shahbaz and R. Groz, "Inferring mealy machines," in *Proceedings of the 2Nd World Congress on Formal Methods*, FM '09, (Berlin, Heidelberg), pp. 207–222, Springer-Verlag, 2009.

[13] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna, "Enemy of the state: A state-aware black-box web vulnerability scanner.," in *USENIX Security Symposium*, pp. 523–538, 2012.

[14] M. Veanes, T. Mytkowicz, D. Molnar, and B. Livshits, "Data-parallel string-manipulating programs," in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 139–152, ACM, 2015.

[15] N. Bjorner, P. Hooimeijer, B. Livshits, D. Molnar, and M. Veanes, "Symbolic finite state transducers, algorithms, and applications," in *IN: PROC. 39TH ACM SYMPOSIUM ON POPL.*, 2012.

[16] M. Veanes, P. De Halleux, and N. Tillmann, "Rex: Symbolic regular expression explorer," in *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pp. 498–507, IEEE, 2010.

[17] J. Hopcroft, "An n log n algorithm for minimizing states in a finite automaton," tech. rep., DTIC Document, 1971.

[18] M. J. Kearns and U. V. Vazirani, *An introduction to computational learning theory*. MIT press, 1994.

[19] D. Angluin, "Learning regular sets from queries and counterexamples," *Information and computation*, vol. 75, no. 2, pp. 87–106, 1987.

[20] R. L. Rivest and R. E. Schapire, "Inference of finite automata using homing sequences," *Information and Computation*, vol. 103, no. 2, pp. 299–347, 1993.

[21] J. E. Hopcroft, *Introduction to automata theory, languages, and computation*. Pearson Education India, 1979.

[22] O. Maler and I.-E. Mens, "Learning regular languages over large alphabets," in *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 485–499, Springer, 2014.

[23] D. Peled, M. Y. Vardi, and M. Yannakakis, "Black box checking," in *Formal Methods for Protocol Engineering and Distributed Systems*, pp. 225–240, Springer, 1999.

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

[25] A. Carayol and M. Hague, "Saturation algorithms for model-checking pushdown systems," *EPTCS*, vol. 151, pp. 1–24, 2014.

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

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

[28] "How to configure urlscan 3.0 to mitigate sql injection attacks." http://goo.gl/cmU0ze. Accessed: 2015-11-10.

[29] "Yaxx project." https://code.google.com/p/yaxx/. Accessed: 2015-11-10.

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

[31] B. W. Watson, "Implementing and using finite automata toolkits," *Natural Language Engineering*, vol. 2, no. 04, pp. 295–302, 1996.

[32] L. D'Antoni and M. Veanes, "Minimization of symbolic automata," in *ACM SIGPLAN Notices*, vol. 49, pp. 541–553, ACM, 2014.

[33] L. DAntoni and M. Veanes, "Equivalence of extended symbolic finite transducers," in *Computer Aided Verification*, pp. 624–639, Springer, 2013.

[34] M. Veanes, "Symbolic string transformations with regular lookahead and rollback," in *Perspectives of System Informatics*, pp. 335–350, Springer, 2014.

[35] R. A. Cochran, L. D'Antoni, B. Livshits, D. Molnar, and M. Veanes, "Program boosting: Program synthesis via crowd-sourcing," in *ACM SIGPLAN Notices*, vol. 50, pp. 677–688, ACM, 2015.

[36] M. Botinčan and D. Babić, "Sigma*: symbolic learning of input-output specifications," *ACM SIGPLAN Notices*, vol. 48, no. 1, pp. 443–456, 2013.

[37] L. DAntoni and M. Veanes, "Extended symbolic finite automata and transducers," *Formal Methods in System Design*, July 2015.

[38] E. M. Gold, "Complexity of automaton identification from given data," *Information and control*, vol. 37, no. 3, pp. 302–320, 1978.

[39] J. L. Balcázar, J. Díaz, R. Gavalda, and O. Watanabe, *Algorithms for learning finite automata from queries: A unified view*. Springer, 1997.

[40] A. Khalili and A. Tacchella, "Learning nondeterministic mealy machines," in *Proceedings of the 12th International Conference on Grammatical Inference, ICGI 2014, Kyoto, Japan, September 17-19, 2014.*, pp. 109–123, 2014.

[41] C. Y. Cho, D. Babic, E. C. R. Shin, and D. Song, "Inference and analysis of formal models of botnet command and control protocols," in *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*, pp. 426–439, 2010.

[42] F. Howar, B. Steffen, B. Jonsson, and S. Cassel, "Inferring canonical register automata," in *Verification, Model Checking, and Abstract Interpretation*, pp. 251–266, Springer, 2012.

[43] S. Cassel, F. Howar, B. Jonsson, M. Merten, and B. Steffen, "A succinct canonical register automaton model," *Journal of Logical and Algebraic Methods in Programming*, vol. 84, no. 1, pp. 54–66, 2015.

[44] T. S. Chow, "Testing software design modeled by finite-state machines," *IEEE transactions on software engineering*, no. 3, pp. 178–187, 1978.

[45] A. Groce, D. Peled, and M. Yannakakis, "Adaptive model checking," in *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 357–370, Springer, 2002.

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

[47] L. Pitt and M. K. Warmuth, "The minimum consistent dfa problem cannot be approximated within any polynomial," *Journal of the ACM (JACM)*, vol. 40, no. 1, pp. 95–142, 1993.

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

[49] Y. Freund and R. E. Schapire, "Large margin classification using the perceptron algorithm," *Mach. Learn.*, vol. 37, pp. 277–296, Dec. 1999.

Appendix

A. Comparison of GOFA algorith 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. [46]).
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 [47]. 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.

B. 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 [44] 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 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 [7] 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 fingerprint algorithm.

```
program name(input){
     return iter(c in input)[registers]
     {cases}end{cases};
}
```

Fig. 9. General structure of a BEK program.

C. Converting Transducers to BEK Programs

In this section we will describe our algorithm to convert finite state transducers into BEK programs. The assumptions we have is that the transducers given to our algorithm are single-valued transducers with bounded lookahead and domain Σ^*. Due to lack of space, we won't describe here the full specification of the BEK language. We urge the interested reader to refer to the original BEK paper [8] as well as to the online tutorial [48].

Figure 9 presents the general template of a BEK program. In a nutshell the BEK language allows one to define an iterator over the input string. In addition, a predefined number of registers taking integer values can be used. Inside the iterator loop an outer switch-case statement is placed, with guards defined by the programmer. Inside each case loop the programmer is allowed to place an if-then-else statement with an arbitrary number of else-if statements and a final else statement. In order to produce an output symbol the `yield` statement is used, which can also produce multiple output symbols. After the main iteration over the input is over, a BEK program can have a final series of case statements which will be evaluated over the register variables defined on the program after exiting the input iteration. We call these statements the `end` part of the iterator.

The overall construction is straightforward in the case the transducer is determinstic: We define a register s which at each point of the computation holds the current state of the transducer. The outer case loop of the program checks the state number while, an internal if-then-else chain matches the current input character and afterwards, sets the next state and yields the corresponding symbol of the transition, if any.

Unfortunately, when a bounded lookahead is present a more complicated situtation arises, because the BEK language cannot process more than one input characters at each iteration. Thus, the program needs to manually store a buffer and keep track of all the alternative states the transducer might be in until a lookahead is matched or discarded.

In fact, as we demonstrate in appendix E, this complexity can easily lead to errors in BEK programs. Indeed, we found a problem in an HTML decoder program which was given as an example in the BEK tutorial. The problem occured because the BEK program was not taking into account all possibilities when a lookahead string was partially matched and then discarded.

The overall structure of a BEK program with lookahead transitions is similar with the basic structure. However, we add additional guards in all states that can be part of a lookahead transition as follows:

Consider each path starting in a final state q_{src} and ending in a final state q_{dst} through a path of non final states, while consuming an input string r, $|r| = k$ and generating an output o. In other words this path is a lookahead transition which consumes the input string r and produces the string o. Then we perform the following:

1) For each prefix of r, r_i for all $i < k$ compute the set of states S_i which are accesible from state q_{src} with the string r_i. Since the transducer is single-valued this set contains exactly one final state. The set S_i of accesible states can be easily computed using a BFS search. Moreover, let o_i be the output of the transducer on string r_i from state q_{src}. We save for each prefix i the triple (r_i, o_i, S_i).
2) Let s_i be the non final state reached by r_i if the suffix following r_i is the remaining symbols of r. Then, for every state $s \in S_i$ add inside the case statement containing the guards of s_i the guards of each $s \in S$ ordered in a way such that the unique final state in S_i is checked last.
3) In the `end` part of the iterator, add for each prefix i a case guard asserting that if the computation ended in state s_i then the program must yield the string o_i. These statements handle the case where the input is finished while processing a lookahead transition.

As soon as we add these additional guards for every lookahead transition the BEK program is completed.

D. Decision trees as SFA

Although are main focus in developing a learning algorithm for SFAs lies in the inference of regular expression filters, SFAs is a very general computation model which allow us to represent various data structures. In figure 10 we show the representation of a decision tree over the real numbers, as a SFA. The predicate family here is the set of linear inequalities of one variable over the real numbers. If we restrict the alphabet Σ to an, infinite, subset of the real numbers such that $\max_{w\in\Sigma}|w| = R$ and moreover, there is a margin γ for every predicate guard [3], then, predicate guards of size k will be $O(kR^2/\gamma^2)$-learnable [49] and thus the overall decision tree can be efficiently inferred using our algorithm.

E. 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 [48]. 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; }
```

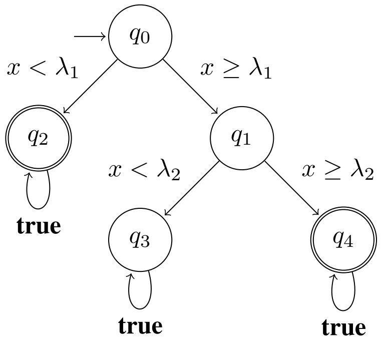

Fig. 10. SFA model for a decision tree over the reals.

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 "&lt;" or "&gt;". 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 "&&lt;".

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 VII-E. 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.

F. Proofs of Theorems and Lemmas

Proof: (of Theorem 1) We need to show that the algorithm does progress towards the discovery of a correct hypothesis. Recall that the algorithm starts with an SOT that is closed and reduced. Each time the algorithm has an SOT that satisfies these properties an equivalence query is issued resulting either in termination or in a counterexample. Processing the counterexample will require $O(\log m + n)$ membership queries. The counterexample will either make the SOT not closed (in which case a new state is introduced) or it will lead to the introduction of an element $s_{i_0}b$ in Λ. A pair of access

[3] A margin γ for a linear inequality $\sum_i a_i\chi_i \geq \theta$ means that, for all $\vec{\chi} \in \Sigma$ $|\sum_i a_i\chi_i + \theta| > \gamma$

strings (s, s') will be called completed if it holds that the guard predicate ϕ in the transition (s, ϕ, s') of the hypothesis is logically equivalent to the predicate ϕ that is in the transition between states q_s and $q_{s'}$ in the target SFA. We will show that for the new element $s_{i_0}b$ that is added in Λ it holds that it corresponds to an s' for which (s_{i_0}, s') is not yet completed. For the sake of contradiction suppose the opposite is true, i.e., that $s_{i_0}b \equiv s' \bmod W \cup \{d\}$ for some s' for which (s, s') is completed. It follows that the the transition $(q_{s_{i_0}}, \phi, q_{s'})$ found in the Hypothesis SFA is correct and it will hold that $\phi(b)$ and also $s_{i_0}b \equiv s' \bmod W \cup \{d\}$. In turn this means that $s_{i_0}b \equiv s' \bmod W$ and as a result $s_{i_0+1} \equiv s' \bmod W$. Because the hypothesis SFA is reduced we obtain $s' = s_{i_0+1}$ which is a contradiction since $s_{i_0}b \not\equiv s_{i_0+1} \bmod W \cup \{d\}$. It follows that $s_{i_0}b \equiv s_j \bmod W \cup \{d\}$ for some j, $j \neq i_0 + 1$ and the pair (s_{i_0}, s_j) is not yet completed. We conclude that (b, s_j) is a counterexample w.r.t. $(R, \boldsymbol{\phi}, \boldsymbol{s})$ where R was the input to the $\mathrm{guardgen}()$ algorithm for the construction of the guard of state s_{i_0} in the hypothesis and $\boldsymbol{\phi}$ is the predicate guard of the state $q_{s_{i_0}}$ in the target automaton. Indeed, (ϕ, s_{i_0+1}) is in the output of $\mathrm{guardgen}()$ and it holds that $\phi(b) = 1$, while $\phi_{i_0+1}(b) = 0$ as $j \neq i_0 + 1$ and $\phi_j(b) = 1$. Using the above, the equivalence queries that result in closed SOT tables cannot exceed $nt(k)$. On the other hand, if an equivalence query results in an SOT that is not closed this results in the introduction of a new state; no membership queries will be needed in this case as the row $s_{i_0}b$ is already determined with respect to $W \cup \{d\}$. The statement of the theorem follows. ■

Proof: (of Theorem 2) First of all observe that there is at least one index $j^* \in \{0, \ldots, |z'| - 1\}$ with the property that $\gamma_{j^*} \neq \gamma_{j^*+1}$. Indeed if the negation of this statement holds it will contradict with the statement that $\gamma_0 \neq \gamma_{|z'|}$. Let $\mathcal{J}^*$ be the set of all such indices. The proof of the theorem is by induction using the previous observation as basis. Suppose that the given range $[j_{\mathsf{left}}, j_{\mathsf{right}}]$ satisfies the property that it intersects with $\mathcal{J}^*$. We will prove that the next range selected by the binary search process as described above preserves the property and it also intersects with $\mathcal{J}^*$. Suppose that j is the middle point of $[j_{\mathsf{left}}, j_{\mathsf{right}}]$ and $\gamma_j = \gamma_0$. The search process selects $[j, j_{\mathsf{right}}]$ as the next range. Suppose for the sake of contradiction that $[j, j_{\mathsf{right}}]$ has no intersection with $\mathcal{J}^*$; this implies $\gamma_{j_{\mathsf{right}}} = \gamma_0$. In case $j_{\mathsf{right}} = |z'|$ this leads immediately to a contradiction. On the other hand, if $j_{\mathsf{right}} < |z'|$ this means that at a previous stage $j_{\mathsf{right}} + 1$ was a middle point and the binary search process decided to choose the left sub-range. By definition this implies that $\gamma_{j_{\mathsf{right}}+1} \neq \gamma_0$. As a result, since $\gamma_{j_{\mathsf{right}}} = \gamma_0$ we obtain that $j_{\mathsf{right}} \in \mathcal{J}^*$ which is again a contradiction. For the second case, suppose that $\gamma_j \neq \gamma_0$ and thus the search process selects $[j_{\mathsf{left}}, j - 1]$ as the next range. Suppose, for the sake of contradiction that $[j_{\mathsf{left}}, j - 1]$ has no intersection with $\mathcal{J}^*$. In case $j_{\mathsf{left}} = 0$ then $\gamma_{j-1} = \gamma_0$ and since $\gamma_j \neq \gamma_0$ we have that $j - 1 \in \mathcal{J}^*$ hence a contradiction. On the other hand, if $j_{\mathsf{left}} > 0$ this means that at a previous stage of the binary search process, j_{left} was a middle point and a decision to go right was made. In turn this implies that $\gamma_{j_{\mathsf{left}}} = \gamma_0$. However by assumption $\gamma_j \neq \gamma_0$ and thus there must be an index in $[j_{\mathsf{left}}, j - 1]$ that belongs to $\mathcal{J}^*$, a contradiction. ■

Proof: (Sketch) (of Theorem 3) The algorithm starts with the empty string as the sole access string and attempts to close the observation table by issuing transduction queries. Eventually the table will become closed, possibly with the addition of certain lookahead transitions in the list L with the respective columns in the observation table. Now it is easy to notice that the SG counterexample processing method will add a distinguishing suffix if the counterexample is due to a hidden state while the prefix-closed queries will detect and process any undiscovered lookahead transition, thus the algorithm will eventually terminate with a correct hypothesis.

Regarding the complexity of the algorithm, notice that the algorithm will issue a prefix-closed query only in order to fill certain entries in the observation table. Therefore, it suffices to bound the size of the rows and columns of the table. The rows of the table remain the same as in the Shabaz-Groz algorithm and therefore, we have at most $(|\Sigma| + 1)n$ rows. The table is initialized with $|\Sigma|$ columns corresponding to each symbol of the alphabet. A column is added either when we process a counterexample due to a hidden state or an undiscovered lookahead transition. We distinguish between the two cases:

- In case the counterexample is due to a hidden state, then at most m columns are added. Since there are at most n counterexamples due to hidden states the total number of columns added can be at most mn.
- In case the counterexample is due to an undiscovered lookahead transition, we notice that the length of the path can be at most n, since we have a bounded lookahead, and therefore at most n columns will be added. Thus, since there is a total of k lookahead transitions at most kn columns will be added.

We notice that each prefix-closed membership query can be implemented with at most $n + \max\{n, m\}$ membership queries, since the longest column is of length $\max\{n, m\}$ and the longest row is of length n. Finally, since a counterexample will be either due to a hidden state or an undiscovered lookahead transition it follows that we can have at most $n + k$ equivalence queries.

■