A Grammatical Inference Approach to Language-Based Anomaly Detection in XML

Harald Lampesberger
Christian Doppler Laboratory for Client-Centric Cloud Computing,
Softwarepark 21, 4232 Hagenberg, Austria
Email: h.lampesberger@cdcc.faw.jku.at

***Abstract*—False-positives are a problem in anomaly-based intrusion detection systems. To counter this issue, we discuss anomaly detection for the eXtensible Markup Language (XML) in a language-theoretic view. We argue that many XML-based attacks target the syntactic level, i.e. the tree structure or element content, and syntax validation of XML documents reduces the attack surface. XML offers so-called schemas for validation, but in real world, schemas are often unavailable, ignored or too general. In this work-in-progress paper we describe a grammatical inference approach to learn an automaton from example XML documents for detecting documents with anomalous syntax.**

We discuss properties and expressiveness of XML to understand limits of learnability. Our contributions are an XML Schema compatible lexical datatype system to abstract content in XML and an algorithm to learn visibly pushdown automata (VPA) directly from a set of examples. The proposed algorithm does not require the tree representation of XML, so it can process large documents or streams. The resulting deterministic VPA then allows stream validation of documents to recognize deviations in the underlying tree structure or datatypes.

***Keywords*-intrusion detection; anomaly detection; XML; grammatical inference**

I. Introduction

Detecting attacks against software is the research field of intrusion detection systems (IDS). We distinguish IDS techniques into *misuse-* and *anomaly-based* detection for hosts and networks: A misuse-based IDS matches *signatures* in a stream of events or network traffic. Contrary, an anomaly-based IDS isolates events or network packets that deviate from *normal behavior*. While signatures represent known patterns of misbehavior, normality for anomaly detection is usually approximated from observations by machine learning or stochastic methods.

In theory, anomaly detection has the advantage of recognizing yet unknown (*zero-day*) or *targeted attacks* that are specifically designed to evade signatures. Bilge and Dumitras [1] show that zero-day attacks are actually frequent and targeted attacks like Stuxnet [2] will likely reoccur in the future. Anomaly detection seems like a perfect solution but suffers from severe practical problems.

False-positives and the high costs associated with them are one major problem [3]. We don't know beforehand how often attacks occur, so the ratio of normal to abnormal events can be heavily skewed: Even a system with low false-positive rate could generate an unacceptable number of false-positives. Sommer and Paxson [4] identify further issues why anomaly detection is not adopted outside academia: It is hard to understand semantics of a detected anomaly and the notion of normality is unstable, especially in networks. Commercial antivirus and network IDS software still relies on signature-based techniques and anomaly detection only plays a supporting role in products that offer behavioral analysis.

The goal of this paper is a more promising anomaly detection technique for the *eXtensible Markup Language* (XML). XML is a platform-independent language for semi-structured data and a pillar of today's Web. Reducing the attack surface therefore makes sense. For our approach we resort to *formal language theory* and *grammatical inference* to understand language-theoretic properties and learnability of XML. We believe, a detection technique can only guarantee low false-positive and high detection rates if it respects these properties.

A. Problem Definition

We consider anomaly detection similar to grammatical inference: Learning a *representation* of a language, e.g. a grammar or automaton, from the *presentation* of a language, e.g. from examples, counter-examples or an oracle. Grammatical inference assumes that there is some hidden *target* representation to be discovered, where language class and type of presentation influence successfulness of learning [5, pp. 141–172]. A learning algorithm is said to converge if the hidden representation is uncovered.

We define our problem as follows: Given a set of example *XML documents*, a learner returns an automaton that allows *validation* of *syntactic structure* and *datatypes* to decide normality of future documents.

While XML is exchanged as document, the underlying logical model is a *tree*. Processing the tree as *Document Object Model* (DOM) [6] requires all the information in memory and this becomes harder with increasing size. We require both automaton and learner to operate in a *streaming fashion*, where memory and time for processing is limited. The *Simple API for XML* (SAX) [7] is our streaming interface to documents.

We approach the problem by first discussing expressiveness of XML. For that, we introduce a formal abstraction of practical schema languages and show that language representation through *visibly pushdown automata* (VPA) is equivalently expressive. VPA are an executable model

This is a preprint version of the paper accepted for publication at the *First Int. Workshop on Emerging Cyberthreats and Countermeasures*, September 2-6, 2013, Regensburg, Germany.

capable of stream processing documents and satisfy our requirement. We then characterize an XML language class that can be efficiently learned from a set of example documents. Content in XML is from an unknown language class in general. We therefore introduce a datatype system for abstracting content of possibly infinite nature into a finite set of datatypes. The contributions are an XML Schema compatible lexical datatype system and a state-merging algorithm for learning VPA. An inferred VPA can validate future documents to recognize anomalous syntax.

The paper is structured as follows: In the remaining introduction we discuss vulnerabilities, XML-based attacks and introduce our learning setting. In Section II we define notations, schema languages and analyze XML expressiveness and its limits for stream validation. VPA are introduced in Section III. Section IV presents our datatype system and learning algorithm. Related work is listed in Section V and Section VI concludes this paper.

B. A Language-Theoretic View on Security

Sassaman et al. [8] analyze the software vulnerability problem using formal language theory. Modularization and composition is an important process in software engineering but implicitly requires *interfaces* and *protocols* between components. A protocol basically specifies the syntax and semantics of a formal language for encoding information, e.g. a file format or network message.

When two components R and S interact, the sender S encodes information w.r.t. the protocol as transportable object, e.g. a network message or file. The receiver R decodes (*parses*) this object according to the protocol and R's internal state is updated in the process. Unfortunately, protocols in the real world are often ambiguous, underspecified or implementations have errors [8]. Sender S might be able to craft a special object such that R moves into an unexpected or insecure state upon parsing. This object is then called *exploit* because it bends or breaks the original intention of the protocol; We say S abuses a *vulnerability* in the protocol to attack R.

An unambiguous and precise protocol specification is required to resolve vulnerabilities such that the receiving component can reject malformed entities [8]. This is exactly the *membership decision problem* in formal languages and it may be intractable or undecidable depending on the language class. Another difficulty is that protocols are often *layered* such that several languages are embedded within each other, e.g. TCP/IP or content in an XML document.

Today's IDS are typically engineered around a specific language class, where computational complexity is tractable. Nevertheless, their goal is to detect exploits in a possibly larger language class or across several layers of embedded languages. False-positives and false-negatives are a direct consequence of mismatching language classes. For example, misuse-based IDS are often restricted to the class of *regular word languages* ($\mathcal{REG}$). If the class of the observed protocol is greater than $\mathcal{REG}$ and there is a vulnerability, there might be infinite variants of exploits that *evade* signatures over $\mathcal{REG}$. Understanding the language-theoretic problems is therefore important.

```
<transaction>
  <total>1000.00<total>
  <cc>
      1234
  </cc>
</transaction>
```

(a) Expected format, attacker controls credit card number [9].

```
<transaction>
  <total>1000.00<total>
  <cc>
      1234' or '1'='1
  </cc>
</transaction>
```

(b) SQL-injection attack.

```
<transaction>
  <total>1000.00<total>
  <cc>
      1234</cc><total>1.00</total><cc>1234
  </cc>
</transaction>
```

(c) XML injection attack for DOM parsers [9].

Figure 1. XML-based attacks.

C. Why Secure XML Processing Matters

XML takes the role of the protocol in Web browsers, mobile applications and Web services. The logical tree structure allows high expressiveness but correct processing becomes more complex and vulnerabilities arise. DOM parsers are vulnerable to Denial-of-Service (DoS) attacks that exhaust time and memory, for example by *overlong element names* or *oversized payload*. A *coercive parsing* attack causes DoS by nesting a vast amount of tags [9].

If an XML parser respects the *Document Type Definition* (DTD) in the preamble of a document, several DoS attacks based on *entity expansion* become a threat. Furthermore, the XML parser could expose confidential information if *external entity references* enable local file import [9].

XML injection is a large class of XML attacks, where the attacker controls parts of a document. Figure 1 describes a fictional transaction document, where a monetary amount is given and the user provides a credit card number. In Figure 1c, the attacker manipulates the transaction value in the DOM tree when a DOM parser is in place [9]. XML injection affects SAX parsers too if the parser state is not propagated correctly. Cross-Site Scripting in the Web is also a form of injection, where a script or Iframe is embedded. Classic attacks like *SQL-*, *command-* or *XPATH-injection* are also a threat if the application that utilizes the XML parser is vulnerable.

Note that all the presented example attacks change the *expected syntax* of a document. Unexpected tree structure or wrong datatypes could lead to harmful interpretation in the XML processing component. Falkenberg et al. [9] and Jensen et al. [10] recommend *strict validation* of XML documents to mitigate attacks but validation requires a language representation, i.e. a *schema*.

Unfortunately, validation is not common. Only 8.9% of XML documents in the Web refer validate to a schema [11]. Also, Web paradigms like *Asynchronous JavaScript and XML* (AJAX) [12] do not enforce schemas or validation, so developers are misled to ad-hoc design. This

motivates learning a language representation from effectively communicated XML for later validation.

D. Learning in the Limit

We consider Gold's *learning in the limit from positive examples* [13] as our grammatical inference setting. The target class, a language class $\mathcal{L}$ expressible by a class of language describing devices $\mathcal{A}$, is *identifiable* in the limit if there exists a learner I with the following properties: Learner I receives as input enumerated examples $E(1), E(2), \ldots$ of some language $L \in \mathcal{L}$, where $E\colon \mathbb{N} \to L$ is an enumeration of L, and examples may be in arbitrary order with possible repetitions. With every input, I returns the current hypothesis $A_i \in \mathcal{A}$, e.g. a grammar or automaton, and there is a point of convergence $N(E)$: For all $j \geq N(E)$, $A_j = A_{N(E)}$ and the language of $A_{N(E)}$ is L. We call I a learner for target class $\mathcal{L}$ if there is convergence for all $L \in \mathcal{L}$. A sample set $S_+ \subseteq L$ is called *characteristic* if learning converges when S_+ is enumerated to I [14].

Unfortunately, grammatical inference is hard and even the class $\mathcal{REG}$ is not learnable in the limit from positive examples only [13]. Learning from XML documents is even harder because it is a *context-free word language*. Ignorance of learnability properties reflects in bad practical performance of anomaly-based IDS. We therefore approach the problem more formally and present a learner for a restricted class of XML in Section IV.

II. XML

The logical structure of XML is a tree, where Σ always denotes the alphabet of element names. We encode attributes as elements with a leading @-character and namespaces as part of the element name. An encoding example is in Figure 2. We disregard identifiers and references because they change the logical structure.

The structure without element content or attribute values is characterized by Σ-trees [15]. The inductive definition of $\mathcal{T}_\Sigma$, the set of all Σ-trees, is: (1) every $c \in \Sigma$ is a Σ-tree; (2) if $c \in \Sigma$ and $t_1, \ldots, t_n \in \mathcal{T}_\Sigma, n \geq 1$ then $c(t_1, \ldots, t_n)$ is a Σ-tree. Σ-trees are *unranked* such that every node can have an arbitrary number of children.

The set of nodes of tree $t \in \mathcal{T}_\Sigma$ is $Dom(t) \subseteq \mathbb{N}^*$ and defined as follows: If $t = c(t_1 \cdots t_n)$ with $c \in \Sigma$, $n \geq 0$ and $t_1, \ldots, t_n \in \mathcal{T}_\Sigma$, then $Dom(t) = \{\epsilon\} \cup \{i.u \mid i \in \{1, \ldots, n\}, u \in Dom(t_i)\}$. Symbol ϵ, the empty word, is the root of the tree and node $v.j$ is the j-th child of node v. The label of v in t is $lab^t(v)$. A *tree language* L over Σ is then a set of trees such that $L \subseteq \mathcal{T}_\Sigma$.

A document is a Σ-tree encoded with tags. For notational convenience, we strip angled brackets such that the set of open-tags is Σ and the set of close-tags becomes $\overline{\Sigma} = \{\overline{c} \mid c \in \Sigma\}$. We use variables $c, c_1, c_2, \ldots, c_i \in \Sigma$ for open-tags and $\overline{c}, \overline{c_1}, \overline{c_2}, \ldots, \overline{c_i} \in \overline{\Sigma}$ for the according close-tags. XML documents without content are words over $(\Sigma \cup \overline{\Sigma})$ and *well-matched* if they obey the grammar $W ::= WW \mid cW\overline{c} \mid \epsilon$.

A tree is translated into a document by pre-order traversal (document order) and we denote the function doc as the bijection between trees and documents.

A. Schemas and Types

A schema is a tree grammar that restricts expressible XML over elements Σ and implicitly gives meaning to structure. DTDs are the simplest form of schemas:

Definition 1 (DTD [16]): A DTD is a triple (Σ, d, s_d), where production rules $d\colon \Sigma \to \mathcal{REG}(\Sigma)$ map element names to *regular expressions* over Σ and s_d is the distinguished start element. The right-hand side of production rules is called *content model* and $L(d)$ is the set of trees that satisfy d.

The expressible language class $\mathcal{DTD}$ is rather limited and practical schema languages like XML Schema (XSD) [17] or Relax NG [18] offer *types* to increase expressiveness. Types are from a finite set, each type is associated with a unique element name and the start element has exactly one type [16]. Variables m, m_0, n in this paper always denote types. As a formal abstraction of practical schema languages we recall the definition of *extended DTD* (EDTD):

Definition 2 (EDTD [16]): An EDTD D is a tuple $D = (\Sigma, M, d, m_0, \mu)$, where M is a set of types, $\mu\colon M \to \Sigma$ is a surjection from types onto element names and (M, d, m_0) is a DTD over types. A tree t satisfies D if $t = \mu(t')$ for some $t' \in L(d)$, where μ ranges over trees. Tree t' is called *witness* for t, $L(D)$ denotes the set of trees and $L^w(D)$ denotes the documents that satisfy D. The language class $\mathcal{EDTD}$ expressible by EDTDs is equivalent to the *regular tree languages* [15].

B. Stream Validation and Expressiveness

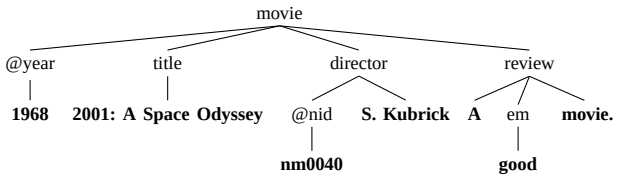
The stream validation or type-checking problem is to decide whether a document is in the language of a given schema within a single pass. Typing a document w is to assign every position i (every tag) some type. A document is *valid* w.r.t. a schema if such an assignment is possible for all positions. Note that function μ is surjective and a position can have multiple types in general.

Martens et al. [16] and Murata et al. [19] discuss *ambiguity* and *determinism* of schemas: A schema is ambiguous if there is a document in the language with multiple types at some position. A schema is *deterministic* if for all described documents at all positions the choice is limited to a single type. In other words, a schema is deterministic if every type assignment is clear when an open-tag is read. Note that ambiguity always implies nondeterminism. Martens et al. [16] introduce the *1-pass pre-order typed* (1PPT) property for EDTDs that are deterministic and therefore allow efficient stream validation.

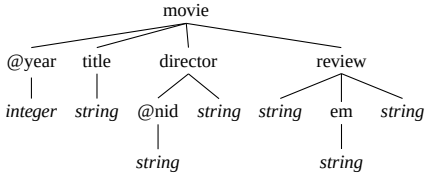
Note that determinism is also important for efficient processing. This is one factor why content models in practical schema languages like DTD and XSD have restrictions to enforce determinism [17]. We direct the reader to Martens et al. [16] for a thorough analysis of expressiveness of schemas.

```
<movie year="1968">
  <title>2001: A Space Odyssey</title>
  <director nid="nm0040">S. Kubrick</director>
  <review>A <em>good</em> movie.</review>
</movie>
```

(a) XML document with attributes, data and mixed content.



(b) Tree with data nodes.



(c) Datatyped tree.

Figure 2. Example XML document and it's tree representation.

We have $\mathcal{DTD} \subsetneq \mathcal{EDTD}^{st} \subsetneq \mathcal{EDTD}^{rc} \subsetneq \mathcal{EDTD}$, where $\mathcal{EDTD}^{st}$ is the class of schemas that satisfy the restrictions of XSD and $\mathcal{EDTD}^{rc}$ is the class of deterministic schemas, where the 1PPT property holds [16].

C. Datatypes and Mixed Content

XML documents carry data as element contents or attribute values and the tag encoding of documents guarantees that tags and data are not confused. We denote data in a document as words over alphabet U, i.e. Unicode, and variables $r, s \in U^*$ always represent data. But data could be from any language class, for example natural language or program code. As an abstraction of data, we introduce *datatypes*:

Definition 3: A so-called lexical *datatype system* is a tuple (Δ, U, ϕ), where Δ is a finite set of datatypes and $\phi\colon \Delta \to \mathcal{P}(U^*)$ is a surjection that assigns every datatype its *lexical space* as some language over U. Because it is surjective, a datum r may have several matching datatypes. Variables $a, b \in \Delta$ always denote datatypes in this paper.

A datatype system has functions $types\colon U^* \to \mathcal{P}(\Delta)$ and $firstType\colon U^* \to \Delta$. While $types(r)$ returns all matching datatypes for some r, $firstType(r)$ chooses one matching datatype to reduce arbitrary data to datatypes.

With respect to the tree structure, content r between tags $cr\overline{c}$ is encoded into a single child node $v.j$ representing the datatype, where $lab^t(v) = c$ and $lab^t(v.j) = r$. We call such a node *data node* for short. Data nodes are always tree leafs and we now refine EDTDs with datatypes:

Definition 4: A *datatype extended DTD* (Δ-EDTD) is a tuple $D = (\Sigma, M, d, m_0, \mu, \Delta, U, \phi)$, where elements (Σ, M, d, m_0, μ) form an EDTD and (Δ, U, ϕ) is a datatype system. As an extension to EDTDs, production rules $d\colon M \to \mathcal{REG}(M \cup \Delta)$ assign every type a content model as regular expression over both types and datatypes. Production rules only allow expressions, where a datatype is followed by either a type or ϵ, but never a subsequent datatype. Lexical spaces $\phi\colon \Delta \to \mathcal{REG}(U)$ are restricted to regular expressions over U.

A *datatyped tree* t' over $(\Sigma \cup \Delta)$ satisfies D if $t' = \mu(t'')$ for some $t'' \in L(d)$, where μ applies only to elements. We denote $L_\Delta(D)$ as the set of datatyped trees that satisfy D. A tree with data nodes t satisfies D if there is a datatyped tree $t' \in L_\Delta(D)$ such that $lab^t(u) \in \phi(lab^{t'}(u))$ holds for all data nodes u and $lab^t(v) = lab^{t'}(v)$ holds for all nodes v with $lab^{t'}(v) \in \Sigma$. Language $L(D)$ denotes the set of trees with data nodes that satisfy D.

Accordingly, we define the word languages generated by Δ-EDTD D. Suppose that bijection doc transforms trees with data nodes and datatyped trees into documents like in Figure 2. Then $L^w_\Delta(D) = \{doc(t) \mid t \in L_\Delta(D)\}$ is the *datatyped word language* and the *document (word) language* generated by D is $L^w(D) = \{doc(t) \mid t \in L(D)\}$.

Δ-EDTDs allow so-called *mixed content*, where between two tags both data and other nested tags are allowed. Mixed content typically appears in markup languages, e.g. the XML Hypertext Markup Language (XHTML). Regarding structural expressiveness, the language classes of EDTDs also translate to our definition of Δ-EDTDs. We now have an abstraction of schema languages that captures attributes, datatypes and mixed-content on a syntactic level.

III. Visibly Pushdown Automata for XML

The well-matched tags in documents induce a visible nesting relation. In fact, XML is a *visibly pushdown language* (VPL) [20] and Kumar et al. [21] show that every EDTD-definable document language is a VPL. This property holds for our definition of Δ-EDTDs because tags are still well-matched and we encode attributes as nested elements. VPLs are accepted by *visibly pushdown automata* (VPA), a restricted form of pushdown automata, where the input symbol determines the stack action.

Definition 5 (VPA [20]): $A = (\tilde{\Sigma}, Q, q_0, Q^F, \Gamma, \delta)$ is a VPA, where $\tilde{\Sigma} = (\Sigma_{call}, \Sigma_{int}, \Sigma_{ret})$ is the pushdown alphabet made of three distinct alphabets, Q is the set of states, $q_0 \in Q$ is the start state, $Q_F \subseteq Q$ are the final states, Γ is the stack alphabet and the transition relation is $\delta = \delta^{call} \cup \delta^{int} \cup \delta^{ret}$, where $\delta^{call} \subseteq (Q \times \Sigma_{call} \times Q \times \Gamma)$, $\delta^{int} \subseteq (Q \times \Sigma_{int} \times Q)$ and $\delta^{ret} \subseteq (Q \times \Gamma \times \Sigma_{ret} \times Q)$.

A transition $(q, c, q', \gamma) \in \delta_{call}$, denoted as $q \xrightarrow{c/\gamma} q'$, is a call-transition from state q to q' that pushes γ on the stack when symbol $c \in \Sigma_{call}$ is read. A transition $(q, \gamma, \overline{c}, q') \in \delta_{ret}$, written as $q \xrightarrow{\overline{c}/\gamma} q'$, is a return-transition from state q to q' that pops γ from the stack when symbol $\overline{c} \in \Sigma_{ret}$ is read. An internal transition $(q, a, q') \in \delta_{int}$, denoted as $q \xrightarrow{a} q'$, moves from state q to q' at input $a \in \Sigma_{int}$ without changing the stack. We direct the reader to Alur and Madhusudan [20] for the semantics of VPA.

Contrary to traditional pushdown automata, VPA can be determinized and are closed under complement, intersection, union, concatenation and Kleene-star. Also language equivalence, emptiness, universality and inclusion are decidable.

Next we will show the equivalence of Δ-EDTDs and XML VPA (XVPA) [21]. XVPA are a special form of *modular VPA* that go back to program modeling. In a modular VPA, states are partitioned into *modules* and the stack alphabet is exactly the set of states. When a module calls another one, the current state is saved on the stack and popped for returning. With respect to XML, modules are exactly the types. Call, return and internal transitions of the VPA are the open-tag, close-tag and character events of the SAX interface to documents.

We assume the following about SAX: There exists a global datatype system (Δ, U, ϕ) and every datum r between two tags or attribute value is reduced to one character event. For stream validation, the SAX interface reports only the first matching datatype $firstType(r)$ to the XVPA instead of r for efficiency. So, the XVPA processes *datatyped documents* and the internal alphabet over datatypes is guaranteed to be finite.

Definition 6 (XVPA [21]): An XVPA A is a tuple $A = (\Sigma, \Delta, M, \mu, \{(Q_m, e_m, X_m, \delta_m)\}_{m \in M}, m_0, F)$, where Σ, Δ, M and μ have the same meaning as in Δ-EDTDs, m_0 is the distinguished start type and $F = X_{m_0}$ are final exit states. Every type $m \in M$ characterizes a module, where

- Q_m is the finite set of module states,
- $e_m \in Q_m$ is a single entry state of the module,
- $X_m \subseteq Q_m$ is the exit of module m (exit states),
- Transitions $\delta_m = \delta_m^{call} \cup \delta_m^{ret} \cup \delta_m^{int}$, where
  - $\delta_m^{call} \subseteq \{q_m \xrightarrow{c/q_m} e_n \mid n \in \mu^{-1}(c)\}$,
  - $\delta_m^{ret} \subseteq \{q_m \xrightarrow{\overline{c}/p_n} q_n \mid q_m \in X_m \wedge n \in \mu^{-1}(c)\}$ and is deterministic, i.e. $q_n = q'_n$ whenever $q_m \xrightarrow{\overline{c}/p_n} q_n$ and $q_m \xrightarrow{\overline{c}/p_n} q'_n$, and
  - $\delta_m^{int} \subseteq \{q \xrightarrow{a} q' \mid q, q' \in Q_m \wedge a \in \Delta\}$.

Return transitions are always deterministic by definition. The XVPA is *deterministic* if also the call transitions are deterministic. The semantics of an XVPA are given by its corresponding VPA $A' = (\tilde{\Sigma}, Q, q_0, \{q_f\}, Q, \delta)$, where $\tilde{\Sigma} = (\Sigma, \Delta, \overline{\Sigma})$, q_0 and q_f are start and accepting state, $Q = \{q_0, q_f\} \cup \bigcup_{m \in M} Q_m$ and transition function δ is defined as

$$\delta = \bigcup_{m \in M} \delta_m \cup \{q_0 \xrightarrow{\mu(m_0)/q_0} e_{m_0}\} \cup \{q \xrightarrow{\overline{\mu(m_0)}/q_0} q_f \mid q \in F\}.$$

The language $L_A(m)$ of module m is a *datatyped word language* and accepted words are of form $\mu(m) w \overline{\mu(m)}$. The accepted language $L(A) = L_A(m_0)$ of XVPA A is the datatyped word language $L(A')$ of its corresponding VPA.

The set X_m are exit states, where at least one return transition originates from. In a valid XVPA, the single-exit property [21] must hold: If there is some return transition $q_m \xrightarrow{\overline{c}/p_n} q_n$ from module m to n, then there must be return transitions $q'_m \xrightarrow{\overline{c}/p_n} q_n$ for all exit states $q'_m \in X_m$. The single-exit property guarantees that $L_A(m)$ is always the same, independent from the calling state or module.

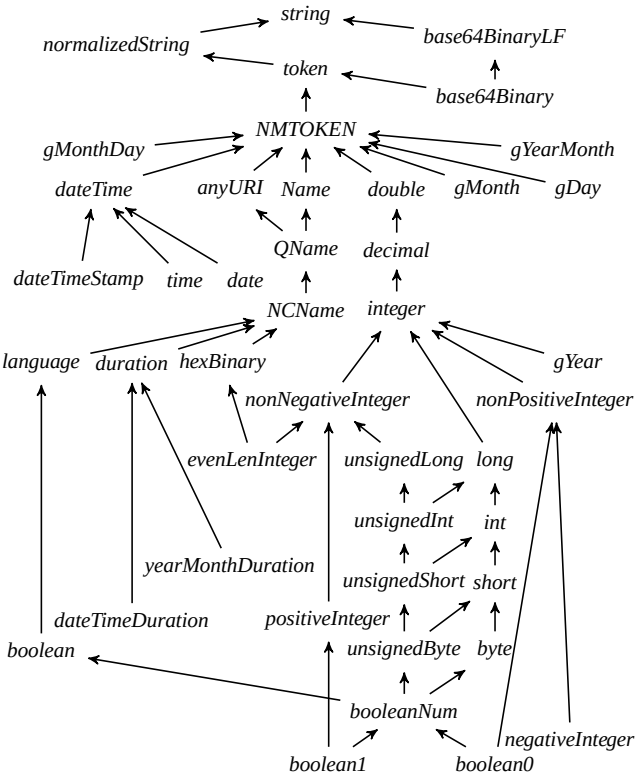


Figure 3. Poset of lexical datatypes Δ in lexical inclusion order.

Theorem 1: Given a datatype system (Δ, U, ϕ), every Δ-EDTD D has a corresponding XVPA A such that their datatyped word languages are equal $L(A) = L^w_\Delta(D)$. Also, for every XVPA A there is an equivalent Δ-EDTD D such that $L^w_\Delta(D) = L(A)$.

The proof is skipped, it refines Kumar et al. [21] with datatypes. Intuitively, every type m has an intermediate DFA D_m in the translation between XVPA modules and regular expressions $d(m)$ in Δ-EDTD production rules. The time complexity of the automaton for processing a document is linear in the length of the document and the required space is bounded by the nesting depth.

IV. Inference from Streaming XML

The goal is to learn an XVPA from a set of example documents S_+. Inference boils down to (1) defining a datatype system, (2) characterizing types and states and (3) learning the language over types and datatypes from examples in S_+.

A. XML Schema Compatible Lexical Datatype System

XVPA in our definition use datatypes as finite alphabet for internal transitions. In Section II we introduce the notion of datatype system but for inference we need a concrete instance. XSD defines a rich set of 47 atomic datatypes together with a hierarchy [22], where every datatype has a semantical value space and a lexical space

that is characterized by a regular expression. Unfortunately, the lexical spaces of XSD datatypes heavily overlap, for example the word '0' is in the lexical space of datatypes *boolean*, *Integer*, or *string* to name a few. A learner only experiences the lexical space and this leads to the problem of choosing the correct datatype for a set of words.

Let (Δ, U, ϕ) be our datatype system, where U is the Unicode alphabet. Based on XSD datatypes we define Δ as a *poset* of 44 datatypes and it is shown in Figure 3. The partial order is the subset relation $\subseteq$ over individual lexical spaces, i.e.

$$a, b \in \Delta\colon a \leq b \iff \phi(a) \subseteq \phi(b) \text{ , where}$$

surjection ϕ maps datatypes to the lexical space definitions of XSD [22] respectively. The following adoptions to datatypes and lexical spaces are made:

- Datatype *anyURI* has an unrestricted lexical space in the XSD standard. In our definition, a datum has datatype *anyURI* iff it is a RFC 2396 *Unified Resource Identifier* with a defined scheme and path.
- The exponents of datatype *double* are unrestricted.
- Datatypes *float, IDREF, IDREFS, ENTITY, ENTITIES, ID, NOTATION* and *NMTOKENS* are dropped because their lexical spaces are indistinguishable from others.
- We add *boolean0, boolean1, booleanNum, evenLenInteger* and *base64BinaryLF* to resolve some severe ambiguities.

If some content r matches datatype a then it also matches datatypes $b_1, b_2, \ldots, b_n$ iff $a \leq b_i$ for $1 \leq i \leq n$. So, the best characterization of r is its *minimal datatype*. We refine functions $types$ and $firstType$ to reflect the partial order: Function $types(r)$ returns all matching minimal datatypes and $firstType(r)$ chooses one matching minimal datatype. For inference, we define the inverse closure $cl^{-1}\colon \mathcal{P}(\Delta) \to \mathcal{P}(\Delta)$ that returns all datatypes that are smaller or equivalent than a given set of datatypes.

B. Characterizing Types and States

Martens et al. [16] characterize types in XSD based on ancestors. Given document w and position i, then the ancestor string $anc\text{-}str(w,i) = c_1 c_2 \cdots c_j$ is the string of unmatched open-tags in the document prefix $w_{1,i} = c_1 w_1 c_2 w_2 \ldots c_j w_j$. A schema has *ancestor-based types* if there exists a hypothetical function $f\colon \Sigma^* \to M$ that assigns every open-tag at position i in document w a single type $f(anc\text{-}str(w_{1,i}, i))$. This restriction is exactly the Element Declarations Consistent (EDC) rule of XSD [17]. Identifying types is then defining relation $\sim_M$ that partitions Σ^* into equivalence classes of ancestor strings. Note that we restrict our learning algorithm automatically to a subset of language class $\Delta\text{-}\mathcal{EDTD}^{st}$ by assuming that types are ancestor-based.

Regarding content models we know that the full class of regular languages is not learnable from positive examples [13]. Bex et al. [23] show that the majority of regular expressions in real world schemas are in fact simple such that every type occurs at most k times in an expression (k-ORE). The language of a k-ORE is a $(k+1)$-*testable regular language*, where grammatical inference from positive examples is feasible [24].

```
1: function DTVPPA^st(S_+)
2:     global (Δ, U, φ)                                  ▷ datatype system
3:     Σ ← Q_F ← δ^call ← δ^ret ← δ^int ← ∅   ▷ initialization
4:     t: Q × Q → P(Δ)                                   ▷ empty dictionary
5:     q_0 ← (ϵ, ϵ)
6:     Q ← {q_0}
7:     for all w ∈ S_+ do                                ▷ iterate over documents
8:         stack ← [⊥]
9:         q ← q_0
10:        for all (event, data) ∈ SAXEvents(w) do
11:            if startElement(event) then               ▷ open-tag
12:                Σ ← Σ ∪ {data}
13:                push(stack, q)
14:                q' ← (π_1(q) · data, ϵ)
15:                δ^call ← δ^call ∪ {q --data/q--> q'}
16:            else if endElement(event) then            ▷ close-tag
17:                assert(data = π_{-1}(π_1(q)))         ▷ matching?
18:                p ← pop(stack)
19:                q' ← (π_1(p), π_2(p) · data)
20:                δ^ret ← δ^ret ∪ {q --data/p--> q'}
21:            else if characters(event) then            ▷ content
22:                q' ← (π_1(q), π_2(q) · $)
23:                t(q, q') ← t(q, q') ∪ types(data)
24:            end if
25:            Q ← Q ∪ {q'}
26:            q ← q'
27:        end for
28:        Q_F ← Q_F ∪ {q}
29:    end for
30:    δ^int = {q --a--> q' | a ∈ cl^{-1}(t(q, q'))}     ▷ int. transitions
31:    return ((Σ, Δ, Σ̄), Q, q_0, Q_F, Q, δ^call ∪ δ^int ∪ δ^ret)
32: end function
```

Figure 4. Visibly Pushdown Prefix Acceptor for class $\Delta\text{-}\mathcal{EDTD}^{st}$.

Our learning strategy is *state-merging*: We first construct a specific VPA that represents exactly S_+ and then generalize by merging similar states. We denote pairs $(x, y) \subseteq (\Sigma^* \times (\Sigma \cup \{\$\})^*)$ as VPA states, where x is an ancestor string and y is a left sibling string $lsib\text{-}str(w,i)$. Symbol $\$ \notin \Sigma$ denotes a placeholder for XML content and $lsib\text{-}str(w,i) = d_1 c_1 d_2 c_2 \cdots d_{n-1} c_{n-1} d_n$, where c is the rightmost unmatched open-tag in the document prefix $w_{1,i} = u c d_1 c_1 v_1 \overline{c_1} d_2 c_2 v_2 \overline{c_2} \cdots d_{n-1} v_{n-1} c_{n-1} \overline{c_{n-1}} d_n$, $d_1, d_2, \ldots, d_n \in \{\$, \epsilon\}$ are optional placeholders and the well-matched substrings $c_j v_j \overline{c_j}$ for $1 \leq j < n$ represent sibling nodes in the tree w.r.t. to position i. As an example, suppose w is the document in Figure 2a and position i is just before tag $\langle$*/review*$\rangle$ then $lsib\text{-}str(w,i) = \$ \cdot em \cdot \$$.

C. The Learning Algorithm

Intuitively, the inference algorithm (1) constructs a so-called *visibly pushdown prefix acceptor* (VPPA) from the sample set, (2) merges similar states, (3) partitions states into modules, (4) adds missing return transitions to satisfy the single-exit property of XVPA and (5) minimizes the XVPA by merging equivalent modules. Figure 5 gives the full algorithm.

function INFERDTXVPA$^{st}_{k,l}(k, l, S_+)$
  global (Δ, U, ϕ) ▷ datatype system
  $((\Sigma, \_, \_), Q, q_0, Q_F, Q^F, \delta) \leftarrow$ DTVPPA$^{st}(S_+)$
  while $\exists q_1, q_2 \in Q\colon q_1 \sim_{k,l} q_2$ **do** ▷ state merging
    $mergeStates(f_{k,l}, q_1, q_2)$
  end while
  $M \leftarrow \{\pi_1(q) \mid \text{for all } q \in Q \wedge \pi_1(q) \neq \epsilon\}$
  $m_0 \leftarrow \pi_1(\delta(q_0, c, q_0))$ ▷ module called by q_0
  for all $m \in M$ **do** ▷ XVPA conversion
    $e_m \leftarrow (m, \epsilon)$
    $Q_m \leftarrow \{q \in Q \mid \pi_1(q) = m\}$
    $\delta_m \leftarrow \{rel \in \delta \mid \pi_1(rel) \in Q_m\}$
    $X_m \leftarrow \{q_m \mid \exists c, p_n, q_n\colon (q_m \xrightarrow{\overline{c}/p_n} q_n) \in \delta_m\}$
    $\delta_m \leftarrow \delta_m \cup \{q_m \xrightarrow{\overline{c}/p_n} q_n \mid \text{for all } q_m \in X_m$ if $\exists q'_m\colon (q'_m \xrightarrow{\overline{c}/p_n} q_n) \in \delta_m\}$
  end for
  while $\exists m, n \in M\colon m \sim_M n$ **do** ▷ minimization
    $mergeModules(m, n)$
  end while
  $\mu \leftarrow \{m \mapsto c \mid m \in M \wedge \exists q\colon (q \xrightarrow{c/q} e_m) \in \delta^{call}\}$
  return $(\Sigma, \Delta, M, \mu, \{(Q_m, e_m, X_m, \delta_m)\}_{m \in M}, m_0, X_{m_0})$
end function

Figure 5. The learning algorithm returns an XVPA with datatypes.

With $\pi_1, \pi_2, \ldots, \pi_n$ we denote projections of the first, second and n-th element and π_{-1} is the last element of a tuple or word A VPPA is a deterministic VPA that represents exactly the examples from S_+ and construction requires only a single pass. The idea of a VPPA is that every prefix of every document in S_+ leads to a unique state in the automaton, similar to a prefix tree acceptor [5, p. 238]. Algorithm DTVPPAst is listed in Figure 4. While iterating over documents in S_+, the algorithm remembers all datatypes that occur between two states in a dictionary-like data structure. After iteration, internal transitions are added for all datatypes in the inverse closure of remembered datatypes. This guarantees that during stream validation the automaton allows a transition if the first matching minimal datatype returned by SAX is valid.

Merging states in the second step generalizes the VPPA. Function $f_{k,l}\colon (\Sigma^* \times (\Sigma \cup \{\$\})^*) \to (\Sigma^{\leq l} \times (\Sigma \cup \{\$\})^{\leq k})$ is a so-called *distinguishing function* [25] that restricts a state q to its local neighborhood by stripping down $\pi_1(q)$ to its l-length suffix and $\pi_2(q)$ to its k-length suffix. With respect to $f_{k,l}$, two states are similar $q_1 \sim_{k,l} q_2$ if they map to the same state $f_{k,l}(q_1) = f_{k,l}(q_2)$. The single state $f_{k,l}(q_i)$ represents equivalence class $[q_i]_{\sim_{k,l}}$, all states in the equivalence class and their transitions are merged into the representative and the VPA stays deterministic.

In the third step, the VPA is turned into an XVPA by partitioning all states $q \in Q$ based on their ancestor-string component $\pi_1(q)$. Types then are $M \subseteq \Sigma^{\leq l}$ and algorithm DTVPPAst guarantees that (m, ϵ) is the single entry state of every module m. Start type m_0 is the one called from state (ϵ, ϵ) and the module of type ϵ is ignored. The XVPA does not satisfy the single-exit property yet. Let X_m be all module states, where some return transition originates from. We add missing returns such that every module n calling m experiences the same language $L_A(m)$.

In the last step, the XVPA is minimized by merging equivalent modules. We define equivalence relation $\sim_M$ such that types m and n are the same if their modules are called by the same open-tag and their corresponding DFA D_m and D_n as constructed in the proof of Theorem 1 are equivalent. If $m \sim_M n$ we redirect all calls and returns from n to m and remove n. Finally, μ maps all types to the elements they are called by. Note that learning the VPPA and state merging can be combined into one efficient step.

D. Example and Discussion

Figure 6 gives a toy example, where the sample set holds a single document. The SAX interface abstracts the contents **10.0** and **TEXT** into simplified datatypes $decimal_\Delta$ and $string_\Delta$ respectively. Note that the state (ab, ϵ) is visited twice during the VPPA construction because both open-tags b in context of element a have the same ancestor-string ab. During state merging, the states (a, ab) and (a, abb) are collapsed into the single state (a, b). In the example, the parameter $l = 2$ leads to two different types **a** and **aa** in the final XVPA. While both corresponding modules are called by the same tag a, they have completely different content models.

The parameters k and l constrain locality of a state. The language class Δ-$\mathcal{EDTD}^{st}_{k,l} \subsetneq \Delta$-$\mathcal{EDTD}^{st}$ is learnable if k and l are bound and S_+ is *characteristic* such that every valid transition in the XVPA appears at least once in the set. Unfortunately, we do not know whether a sample set is characteristic. But we can guarantee that the quality of the learned automaton stays the same or improves with every example in the sample set if the hidden target is in language class Δ-$\mathcal{EDTD}^{st}_{k,l}$.

If $l = 1$ then types are exactly element names and the algorithm learns a proper subset of Δ-$\mathcal{DTD}$. A parameter $k = 1$ limits the left-sibling string of a state to element names or the \$ symbol, so inferred XVPA modules become equivalent to Single Occurrence Automata [26] in terms of expressiveness. In the case that k and l are chosen too small, the resulting automaton over-generalizes the language. Contrary, increasing the parameters requires much larger characteristic sets for convergence.

V. RELATED WORK

XML stream validation is first discussed by Segoufin and Vianu [27]. Kumar et al. [21] introduce VPA as executable model for XML that captures the entire class of regular tree languages. Schewe et al. [28] extend VPA for approximate XML validation and Picalausa et al. [29] present an XML Schema framework using VPA.

For a survey of grammatical inference we direct the reader to the book of de la Higuera [5]. Fernau [25] introduces function distinguishable languages and we apply this concept in Section IV for state merging. Kumar et al. [30] mention that query learning VPA with counterexamples is possible but our setting is different.

Several results on DTD inference from XML have been published [26], [31]–[33], but we aim for the strictly larger

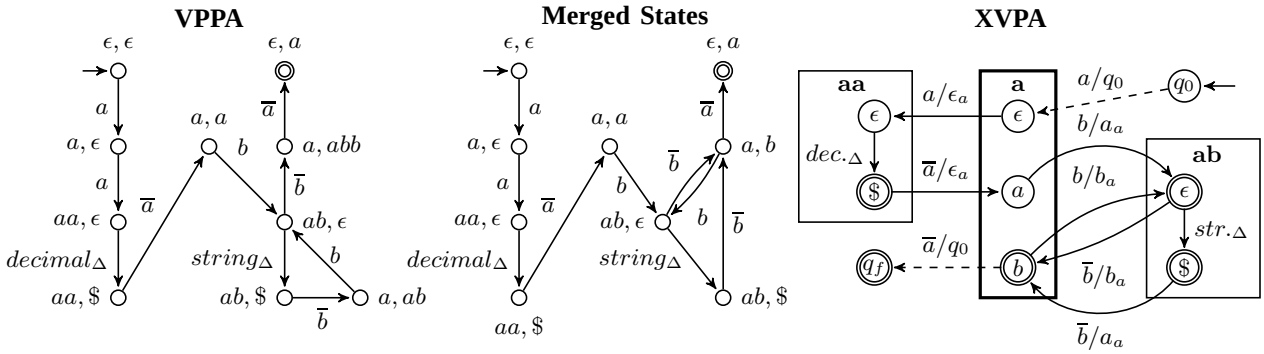


Figure 6. INFERDTXVPA$^{st}_{k,l}$ example for $S_+ = \{aa\mathbf{10.0}\overline{a}b\mathbf{TEXT}\overline{b}b\overline{b}\overline{a}\}$ and parameters $k = 1, l = 2$.

class of XSDs. Mlýnková [34] presents a survey of XSD inference. The general idea is to start with an extended context-free grammar as schema abstraction, inferred from examples, and merge non-terminals [35]. Hegewald et al. [36] and Chidlovskii [37] also handle datatypes in their presented methods. Our approach is similar to Bex et al. [38]. Their algorithms use tree automata for learning l-local Single Occurrence XSDs in a probabilistic setting but without datatypes.

In the field of information retrieval, Kosala et al. [39] and Raeymaekers et al. [40] give algorithms to infer HTML wrappers as tree automata. Regarding intrusion detection, Rieck et al. [41] introduce approximate tree kernels as a similarity measure for trees and use them for anomaly detection in HTML.

To our knowledge the presented approach is the first that directly learns an automaton model with both streaming and datatypes in mind. A hard problem in learning schemas is to find nice regular expressions for content models. We focus on learning an automaton representation and intentionally leave conversion to regular expressions open, as many of the noted references propose heuristics or solutions.

VI. Conclusion and Future Work

We approached the problem of anomaly detection in XML more formally and introduced Δ-EDTDs as abstraction of practical schema languages with datatypes. We showed that XVPA are an equivalent model capable of stream validation and contributed a lexical datatype system and an algorithm for learning an XVPA from a set of documents. The algorithm converges for target class Δ-$\mathcal{EDTD}^{st}_{k,l}$ given the sample set is characteristic. A learned automaton could theoretically be converted into an XSD schema.

The presented work is still in an early stage. We already have a working prototype which is our baseline for further research and the next step is a thorough evaluation with XML-based attacks. First experiments with the prototype indicate that abstraction by the lexical datatype system using XSD datatypes is too coarse in some cases. We will therefore look into approximations of specific datatypes during learning. Other improvements are to extend the learnable language class and redefine the algorithms for incremental learning. Also, we do not know if some sample set is characteristic and leads to convergence. A refinement to a probabilistic learning setting could enhance applicability when sample sets are incomplete or noisy.

Finally, it is of great interest how our approach to XML inference and stream validation translates to other prominent semi-structured languages like JSON or HTML. An application in mind is a client-side component that learns how Web applications and services communicate with a Web client and detects syntactical deviations, for example caused by Cross-Site Scripting attacks.

Acknowledgment

We thank Philipp Winter for the helpful feedback and suggestions. This research has been supported by the Christian Doppler Society.

References

[1] L. Bilge and T. Dumitras, "Before we knew it: an empirical study of zero-day attacks in the real world," in *Proc. of the 2012 ACM conference on Computer and communications security - CCS '12*. ACM Press, 2012, pp. 833–844.

[2] Symantec, "W32.Stuxnet," http://www.symantec.com/security_response/writeup.jsp?docid=2010-071 [Online. Last accessed: 2013-3-19].

[3] S. Axelsson, "The base-rate fallacy and its implications for the difficulty of intrusion detection," in *Proc. of the 6th ACM conference on Computer and communications security - CCS '99*. ACM Press, 1999, pp. 1–7.

[4] R. Sommer and V. Paxson, "Outside the closed world: On using machine learning for network intrusion detection," in *Proc. IEEE Symposium on Security and Privacy*. IEEE, 2010, pp. 305–316.

[5] C. de la Higuera, *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, 2010.

[6] W3C, "Document object model (dom)," http://www.w3.org/DOM/ [Online. Last accessed: 2013-1-24].

[7] The SAX Project, "Simple api for xml (sax)," http://www.saxproject.org/ [Online. Last accessed: 2013-1-24].

[8] L. Sassaman, M. L. Patterson, S. Bratus, M. E. Locasto, and A. Shubina, "Security applications of formal language theory," Dartmouth College Computer Science Department, Tech. Rep. TR2011-709, 2011.

[9] A. Falkenberg, M. Jensen, and J. Schwenk, "Ws-attacks.org," http://www.ws-attacks.org [Online. Last accessed: 2013-2-5].

[10] M. Jensen, N. Gruschka, and R. Herkenhöner, "A survey of attacks on web services," *Computer Science - Research and Development*, vol. 24, no. 4, pp. 185–197, 2009.

[11] S. Grijzenhout and M. Marx, "The quality of the xml web," in *Proc. of the 20th ACM int. conference on Information and knowledge management - CIKM '11*. ACM Press, 2011, pp. 1719–1724.

[12] J. J. Garrett, "Ajax: A new approach to web applications," http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications [Online. Last accessed: 2013-3-27].

[13] E. M. Gold, "Language identification in the limit," *Information and Control*, vol. 10, no. 5, pp. 447–474, 1967.

[14] H. Fernau, "Algorithms for learning regular expressions from positive data," *Information and Computation*, vol. 207, no. 4, pp. 521–541, Apr. 2009.

[15] F. Neven, "Automata, logic, and xml," in *Computer Science Logic*, ser. LNCS. Springer, 2002, vol. 2471, pp. 671–711.

[16] W. Martens, F. Neven, T. Schwentick, and G. J. Bex, "Expressiveness and complexity of xml schema," *ACM Trans. on Database Systems*, vol. 31, no. 3, pp. 770–813, 2006.

[17] W3C, "Xml schema," http://www.w3.org/XML/Schema.html [Online. Last accessed: 2013-2-1].

[18] M. Murata, "Relax ng," http://relaxng.org/ [Online. Last accessed: 2013-2-1].

[19] M. Murata, D. Lee, M. Mani, and K. Kawaguchi, "Taxonomy of xml schema languages using formal language theory," *ACM Trans. on Internet Technology*, vol. 5, no. 4, pp. 660–704, 2005.

[20] R. Alur and P. Madhusudan, "Visibly pushdown languages," in *Proc. of the thirty-sixth annual ACM Symposium on Theory of Computing - STOC '04*. ACM Press, 2004, pp. 202–211.

[21] V. Kumar, P. Madhusudan, and M. Viswanathan, "Visibly pushdown automata for streaming xml," in *Proc. of the 16th Int. Conf. on World Wide Web - WWW '07*. ACM Press, 2007, p. 1053.

[22] W3C, "Xml schema part 2: Datatypes second edition," http://www.w3.org/TR/xmlschema11-2/ [Online. Last accessed: 2013-3-22].

[23] G. J. Bex, F. Neven, and J. Van den Bussche, "Dtds versus xml schema: A practical study," in *Proc. of the 7th Int. Workshop on the Web and Databases - WebDB '04*. ACM Press, 2004, p. 79.

[24] P. García and E. Vidal, "Inference of k-testable languages in the strict sense and application to syntactic pattern recognition," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. 12, no. 9, pp. 920–925, 1990.

[25] H. Fernau, "Identification of function distinguishable languages," *Theoretical Computer Science*, vol. 290, no. 3, pp. 1679–1711, 2003.

[26] G. J. Bex, F. Neven, T. Schwentick, and S. Vansummeren, "Inference of concise regular expressions and dtds," *ACM Trans. on Database Systems*, vol. 35, no. 2, pp. 1–47, 2010.

[27] L. Segoufin and V. Vianu, "Validating streaming xml documents," in *Proc. of the twenty-first ACM Symposium on Principles of Database Systems - PODS '02*. ACM Press, 2002, p. 53.

[28] K.-D. Schewe, B. Thalheim, and Q. Wang, "Updates, schema updates and validation of xml documents - using abstract state machines with automata-defined states," *J.UCS*, vol. 15, no. 10, pp. 2028–2057, 2009.

[29] F. Picalausa, F. Servais, and E. Zimányi, "Xevolve: an xml schema evolution framework," in *Proc. of the 2011 ACM Symposium on Applied Computing - SAC '11*. ACM Press, 2011, p. 1645.

[30] V. Kumar, P. Madhusudan, and M. Viswanathan, "Minimization, learning, and conformance testing of boolean programs," in *CONCUR 2006 – Concurrency Theory*, ser. LNCS. Springer, 2006, vol. 4137, pp. 203–217.

[31] G. J. Bex, W. Gelade, F. Neven, and S. Vansummeren, "Learning deterministic regular expressions for the inference of schemas from xml data," *ACM Trans. on the Web*, vol. 4, no. 4, pp. 1–32, 2010.

[32] H. Fernau, "Learning xml grammars," in *Machine Learning and Data Mining in Pattern Recognition*, ser. LNCS. Springer, 2001, vol. 2123, pp. 73–87.

[33] M. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, and K. Shim, "Xtract: Learning document type descriptors from xml document collections," *Data Mining and Knowledge Discovery*, vol. 7, no. 1, pp. 23–56, 2003.

[34] I. Mlýnková, "An analysis of approaches to xml schema inference," in *2008 IEEE Int. Conf. on Signal Image Technology and Internet Based Systems*. IEEE, 2008, pp. 16–23.

[35] I. Mlýnková and M. Nečaský, "Towards inference of more realistic xsds," in *Proc. of the 2009 ACM Symposium on Applied Computing - SAC '09*. ACM Press, 2009, p. 639.

[36] J. Hegewald, F. Naumann, and M. Weis, "Xstruct: Efficient schema extraction from multiple and large xml documents," in *22nd Int. Conf. on Data Engineering Workshops (ICDEW'06)*. IEEE, 2006, pp. 81–81.

[37] B. Chidlovskii, "Schema extraction from xml: A grammatical inference approach," in *Proc. of the 8th Int. Workshop on Knowledge Representation meets Databases (KRDB 2001)*, 2001.

[38] G. J. Bex, F. Neven, and S. Vansummeren, "Inferring xml schema definitions from xml data," in *VLDB '07 Proc. of the 33rd Int. Conf. on Very Large Data Bases*. VLDB Endowment, 2007, pp. 998–1009.

[39] R. Kosala, H. Blockeel, M. Bruynooghe, and J. Van den Bussche, "Information extraction from structured documents using k-testable tree automaton inference," *Data & Knowledge Engineering*, vol. 58, no. 2, pp. 129–158, 2006.

[40] S. Raeymaekers, M. Bruynooghe, and J. den Bussche, "Learning (k, l)-contextual tree languages for information extraction from web pages," *Machine Learning*, vol. 71, no. 2, pp. 155–183, 2008.

[41] K. Rieck, "Machine learning for application-layer intrusion detection," Ph.D. dissertation, Berlin Institute of Technology, TU Berlin, Germany, 2009.