An Incremental Learner for Language-Based Anomaly Detection in XML

Harald Lampesberger
Department of Secure Information Systems
University of Applied Sciences Upper Austria
Email: harald.lampesberger@fh-hagenberg.at

***Abstract*—Schema validation for the Extensible Markup Language (XML) corresponds to checking language acceptance and is therefore a first-line defense against attacks. However, extension points in XML Schema can render validation ineffective. Extension points are wildcards and considered best practice for loose composition, but an attacker can also place any content at an extension point without rejection, e.g., for a signature wrapping attack.**

This paper presents an incremental learner that infers types and datatypes in mixed-content XML in terms of a datatyped XML visibly pushdown automaton as language representation. An inferred automaton is free from extension points and capable of stream validation, e.g., in a monitoring component. Additional operations for unlearning and sanitization improve practical usability. The learner is guaranteed to converge to a good-enough approximation of the true language, and convergence can be measured in terms of mind changes between incremental steps. All algorithms have been evaluated in four scenarios, including a web service implemented in Apache Axis2 and Apache Rampart, where XML attacks have been simulated. In all scenarios, the learned representation had zero false positives and outperformed schema validation.

***Index Terms*—XML, grammatical inference, visibly pushdown automata, stream validation, anomaly detection, experimental evaluation.**

1. Introduction

The Extensible Markup Language (XML) [1] is ubiquitous in electronic communication, e.g., web services utilizing the Simple Object Access Protocol (SOAP), the Extensible Messaging and Presence Protocol (XMPP), single sign-on systems using the Security Assertion Markup Language (SAML), and many data serialization formats. The success of XML boils down to its rich data models and tool support: Instead of specifying a language and data model for some protocol from scratch, a software developer can simply define a subset of XML and reuse existing parsing and querying tools.

- *The research was done while the author was affiliated with the Christian Doppler Laboratory for Client-Centric Cloud Computing, Johannes Kepler University Linz, Austria.*

XML attacks, in particular, the signature wrapping attack [2], have motivated this work. Schema wrapping exploits XML identity constraints and composition in the industry standard XML Schema (XSD), and research by Gajek et al. [3], [4], Somorovsky et al. [5], [6], and Jensen et al. [7], [8] indicates that there is a fundamental problem.

This paper discusses a language-theoretic view and extends a previous grammatical inference approach [9]. The contributions are datatyped XML visibly pushdown automata (dXVPAs) and character-data XVPAs (cXVPAs) as language representations for mixed-content XML and capable of stream processing, algorithms for datatype inference from text contents, an incremental learner, and an experimental evaluation of the proposed approach.

The dXVPA and cXVPA models extend XVPAs which have been introduced by Kumar et al. [10]. The learner converges to an approximation of the true language, free from extension points, and convergence can be measured in terms of mind changes between incremental learning steps. A use case is a monitor that validates input for some XML-based interface. Furthermore, unlearning and sanitization operations are added for better usability in a monitor. All algorithms have been implemented and evaluated in four scenarios, where attacks are simulated: two synthetic and two realistic scenarios, utilizing Apache Axis2 and Apache Rampart. In all scenarios, the learned dXVPA representation outperformed baseline schema validation.

1.1. XML

XML specifies a syntax: open- and close-tags for elements, attributes, namespaces, allowed characters for text content and attribute values, processing instructions for the parser, inline Document Type Definition (DTD) declarations, and comments. The syntax allows ambiguities, e.g., an element without text content, and XML Information Set [11] therefore defines a tree-structured data model to remove syntactic ambiguities: A document has an infoset if it is *well formed* and all namespace constraints are satisfied.

Business logic accesses infoset items in a document through an interface. Common APIs for XML can be distinguished into (a) stream based, e.g., Simple API for XML (SAX) [12] and Streaming API for XML (StAX) [13] and (b) tree based, e.g., a Document Object Tree (DOM).

For defining a set of documents, a *schema* is a tree grammar, and the XML community provides several schema languages for specifying production rules, e.g., DTD [1], XSD [14], Relax NG [15], and Extended DTD (EDTD) [16] as a generalization. Productions are of form $a \to B$, where B is a regular expression and called *content model* of a. In DTD, rules are expressed over elements. To raise expressiveness, productions in XSD, Relax NG, and EDTD are defined over *types* instead, and every type maps to an element. This mapping is surjective: two types can map to the same element.

Schema validation corresponds to language acceptance of a document. *Typing* is stricter than validation by assigning a unique type from productions to every element [17]. The power of regular expressions and the surjective relation between types and elements can introduce ambiguity and nondeterminism, but determinism is appreciated, e.g., for assigning semantics to elements based on the type in business logic. DTD and XSD therefore have syntactic restrictions to ensure deterministic typing. Schema validation and typing are first-line defenses against attacks; however, identity constraints in XML and best practices for composable schemas in XSD can render validation ineffective.

1.2. Language-Theoretic Vulnerabilities

The XML syntax is context free and infoset items are tree structured, but logically, a document is not always a tree. Identity constraints like keys (ID) and key references (IDREF, IDREFS) introduce self references that go beyond context freeness. Cyclic and sequential references turn a finite tree data model logically into an infinite tree, and operations such as queries become computationally harder [18]. Furthermore, XSD introduces additional constraints (unique, key, and keyref) over text contents, attribute values, and combinations thereof. Checking identity constraints during validation requires significantly more time and space for constructing indices or traversing the data model multiple times, or constraints are not properly checked at all.

Also, there are two philosophies of modularity in XSD: *schema subtyping* [19] by refining productions and *schema extension points* using wildcards (`xs:any`). Extension points allow loose coupling and are considered best practice [20]. In an XSD, a wildcard is often accompanied by the `processContents="lax"` attribute which has a tremendous effect on validation: if there is no schema in the parser's search space for a qualified element at an extension point, validation is skipped. By choosing a random namespace, arbitrary content can be placed at an extension point. Unfortunately, extension points are in many standard XSDs, e.g., SOAP, XMPP, and SAML.

1.3. XML Attacks

Attacks can be distinguished into parsing and semantic attacks. Parsing attacks target lexical and syntactical analysis, e.g., for Denial-of-Service. Examples are oversized tokens, high node counts from unbounded repetitions [21], and

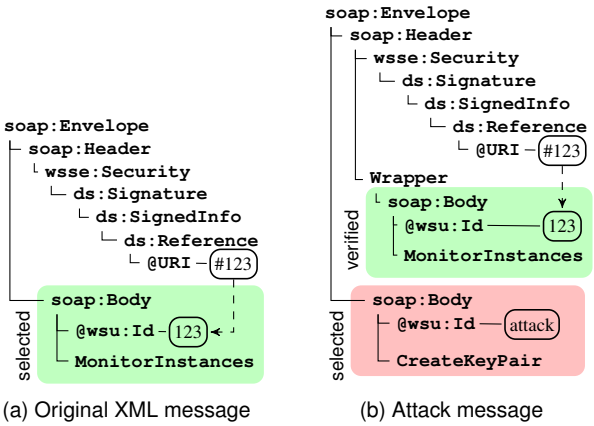


Figure 1. XML signature wrapping attack (reproduced from [5])

coercive parsing [22]. Schema validation is unable to reject parsing attacks when they are placed at an extension point. A special class of parsing attacks originates from inline `DOCTYPE` declarations, i.e., exponential entity expansion, external entities for priviledge escalation, and server-side request forgery (SSRF) [23].

Semantic attacks aim for misinterpretation in the business logic and interacting components. A document is processed by handling SAX/StAX events or traversing a DOM tree, and the simplest attack is tampering with the document structure. CDATA fields [24] can exclude reserved characters (e.g. angled brackets) from lexical analysis, and they are a helper for many semantic attacks, e.g., XML, SQL, LDAP, XPath, and command injection; path traversal; memory corruption in interacting components; and cross-site request forgery (XSRF) and cross-site scripting (XSS) with respect to web applications [21].

1.3.1. Signature Wrapping Attack. Signature wrapping is a sophisticated semantic attack. XML Signature [25] specifies a `ds:signature` element that holds one or more hashes of referenced resources (i.e., elements in the document) and is signed for authenticity. The resources are referenced by ID or an XPath expression. Signature checking verifies the authenticity of the `ds:signature` element and compares the stored hashes with computed ones. Checking is usually treated as a Boolean decision, independent from the business logic, and a vulnerability emerges when verified locations are not communicated.

A signature wrapping attack exploits location intransparency by moving the referenced resource at an extension point and placing a malicious element at the original location. An example is shown in Figure 1. The attacker needs access to some document with a valid signature (Figure 1a), the signature remains in the modified document (Figure 1b), and the document stays valid with respect to its schema. The vulnerable business logic then selects the information provided by the attacker. This example is the Amazon EC2 attack by Somorovsky et al. [5], and similarly, many SAML implementations were vulnerable [6].

1.3.2. Countermeasures. Several countermeasures have been proposed since discovery of signature wrapping. *Security policies* [2] can enforce properties of SOAP messages, but policy checking can be computationally costly. Gruschka and Iacono [26] furthermore show a successful signature wrapping attack on Amazon EC2 that satisfies the mentioned security policies.

Signature wrapping modifies the SOAP message structure, and Rahaman et al. [27] propose an *inline approach* by adding an element that stores document characteristics. Unfortunately, if a single element in the header is unsigned, the approach can be circumvented [3].

Gajek et al. [3] and Somorovsky et al. [6] propose *improved signature verification* by returning a filtered view instead of a Boolean decision, but the business logic needs to be adapted accordingly. Gajek et al. [4] also propose FastXPath, a subset of XPath, for location-aware XPath-based references in signatures. Namespace injection could circumvent this countermeasure if namespaces are not explicitly defined in the XPath expressions [7].

Jensen et al. [8] propose *schema hardening* by removing extension points and restricting repetitions. Hardening is effective because arbitrary elements cannot be hidden; however, all supported schemas need to known beforehand, and generating a single unified hardened schema is computationally hard. Experiments have also shown a significant slowdown in schema validation.

Finally, XSpRES (XML Spoofing Resistant Electronic Signature) [28] is a standard-compliant framework that unifies the mentioned countermeasures.

1.4. Research Questions

Removal of extension points is an effective countermeasure, but schema hardening can be difficult. This paper proposes a learning-based approach in terms of an interface-centric monitor for an XML-based client or service. The monitor has a learner and validator component. The learner component infers a language representation, and the validator component then uses this representation to check the syntax of documents sent to the system under observation. Validation is relative to the language inferred from training examples, and the approach is therefore called *language-based anomaly detection*. If the validator component rejects a document, some filtering or extended policy checking could be performed, but these operations are not in the scope of this work and not further specified.

The assumed attacker is capable of reading and modifying documents in transit and sending a malicious document directly to the system under observation. The attacker is therefore able to instigate the previously mentioned parsing and semantic attacks.

Clients and services are considered black boxes, where message semantics are unknown to the monitor; however, semantics are important under the language-theoretic security threat model because an attack is a special case of misinterpretation. When a system interprets a document, semantics are usually assigned to types (ad hoc or from schema production rules) and datatypes of attribute values and text content. The mentioned attacks affect at least one type or datatype in a document. We assume that the system under observation has *type-consistent behavior*: for all manifestations of an expected type or datatype, the behavior of the system under observation is well-specified. In other words, a language-based anomaly detection approach can only work if attacks are syntactically distinguishable from expected types and datatypes.

To sum up, the research questions are:

RQ1 What is a suitable representation for types and datatypes and capable of checking acceptance?
RQ2 Can this language representation be learned?
RQ3 Can the proposed approach identify attacks?

1.5. Methodology

1.5.1. Language Representation. In mixed-content XML, texts are strings over Unicode, and they are allowed between a start- and an end-tag, an end- and a start-tag, two start-tags, and two end-tags. XSD and Relax NG provide *datatypes* for specifying texts: every datatype has a value space and a lexical space over Unicode.

The language representation needs to be as least as structurally expressive as XSD, and support *stream validation* for open-ended XML protocols (i.e., XMPP) and very large documents. Kumar et al. [10] introduce XVPAs as a language representation that accepts SAX/StAX event streams and enables linear-time stream validation. Every EDTD (including the expressiveness of XSD) can be translated into an XVPA [10]. However, text contents and attribute values are not considered in XVPAs yet.

For answering RQ1, the paper introduces two automaton models. XVPAs are extended by datatypes toward dXVPAs, and cXVPAs are an optimization of dXVPAs for linear-time stream validation of mixed-content XML.

1.5.2. Learning from Positive Examples. The learner component receives examples and computes a dXVPA for the validation component. This learning setting corresponds to Gold's *identification in the limit from positive examples* [29], and the following definition is according to Fernau [30].

Definition 1 (Identification in the limit from positive examples [30])**.** Let $\mathcal{L}$ be a target language class that can be characterized by a class of language-describing devices $\mathcal{D}$. $E\colon \mathbb{N} \to L$ is an enumeration of strings for a language $L \in \mathcal{L}$, and the examples may be in arbitrary order with possible repetitions. Target class $\mathcal{L}$ is *identifiable in the limit* if there exists a inductive inference machine or learner I:

- Learner I receives examples $E(1), E(2), \ldots$
- For every E_i, learner I computes the current hypothesis (e.g., an automaton) $D_i \in \mathcal{D}$.
- For every enumeration of $L \in \mathcal{L}$, there is a convergence point $N(E)$ such that $L = L\left(D_{N(E)}\right)$ and $j \geq N(E) \implies D_j = D_{N(E)}$.

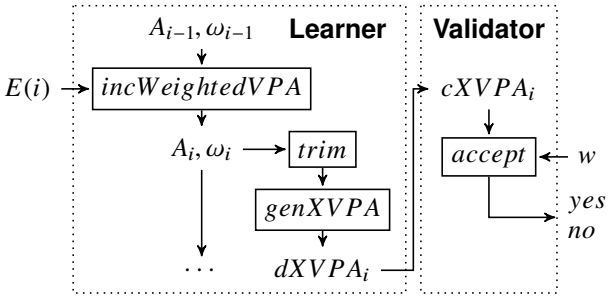


Figure 2. Incremental learning step

To answer RQ2, algorithms for datatype inference from text and a state-merging approach for identification in the limit are specified. State merging exploits the *locality* of types for generalization. Furthermore, algorithms for unlearning and sanitization refine the learner for improving the usability in the proposed monitor.

1.5.3. Experimental Evaluation. A learning-based approach is still a heuristic and requires experimental evaluation. Four datasets have been generated: two synthetic scenarios using a stochastic XML generator and two realistic scenarios from message recordings of a SOAP/WS-* web service implemented in Apache Axis and Apache Rampart. The service has been implemented according to best practices, and attacks have been performed manually and automatically by the WS-Attacker [31] tool for Denial-of-Service and signature wrapping attacks.

For answering RQ3, detection performance, learning progress in terms of mind changes, and effects of unlearning and sanitization have been analyzed for varying degrees of generalization.

2. Grammatical Inference of XML

This section introduces document event streams, dXVPAs, cXVPAs, datatype inference from text contents, a Gold-style incremental learner, unlearning, and sanitization.

Figure 2 illustrates an incremental learning step. The learner component maintains an internal visibly pushdown automaton (VPA) A and counters $\omega_\delta, \omega_Q, \omega_F$ for transitions, states, and final states respectively. A VPA is a special pushdown automaton whose transitions distinguish three disjoints alphabets: a call alphabet that pushes on the stack, an internal alphabet that leaves the stack unchanged, and a return alphabet that pops from the stack; these alphabets are utilized for different kinds of XML events. The set of states is also the stack alphabet in the proposed learner. VPAs have been introduced by Alur et al. [32], and the reader is directed to this work for a full definition.

Algorithm 3 ($incWeightedVPA$) receives an event stream from document $E(i)$ and incrementally updates the VPA and the counters. The counters are frequencies of states or transitions from learning and necessary for unlearning and sanitization operations. Algorithm 4 ($trim$) removes zero-weight states and transitions, and Algorithm 5 ($genXVPA$) generates a minimized dXVPA. The dXVPA has an equivalent cXVPA for the validator component, and acceptance of documents sent to the system under observation can be decided.

```
<a>
  <a>10.0</a>
  <b>some text content</b><b></b>
</a>
```

↧

$$\mathtt{aa[10.0]\overline{a}b[some\ text\ content]\overline{b}b\overline{b}\overline{a}}$$

Figure 3. Symbolic StAX event stream

2.1. Document Event Stream

Definition 2 (Document event stream)**.** A *document event stream* w is a sequence of StAX events e carrying values $lab(e)$. There are three kinds of events: $startElement$ and $endElement$ for qualified element names and $characters$ for texts. Processing instructions, comments, and entity references are ignored. Attributes are alphabetically sorted, treated as special elements with a leading `@` symbol, and mapped to a subsequence of $startElement$, $characters$, and $endElement$. Reserved XML attributes for namespace declarations are removed. The value of a $characters$ event is the text content as string over Unicode U, and nested CDATA sections are automatically unwrapped by the parser.

Figure 3 illustrates an example. A $startElement$ event is denoted by its qualified name `m`, and $\overline{\mathtt{m}}$ for the respective $endElement$ event. A text between squared brackets denotes a $characters$ event and its text content.

Texts can be restricted by datatypes in XSD and Relax NG. For this work, only the lexical spaces of XSD datatypes [33] are considered in a generalized notation of lexical datatypes.

Definition 3 (Lexical datatypes)**.** Let T be a set of *lexical datatypes*. A lexical space is a regular language over Unicode, and $\phi : T \to REG(U)$ assigns lexical spaces.

Lexical datatypes allow to define datatyped event streams, where datatypes replace texts in $characters$ events.

Definition 4 (Datatyped event stream)**.** A *datatyped event stream* w' is a sequence of $startElement$, $endElement$, and $characters$ events. The value of a $characters$ event e is a datatype $lab(e) \in T$. A document event stream w *corresponds* to a datatyped event stream w' if w and w' are congruent with respect to event kinds, the qualified element names in $startElement$ and $endElement$ events are the same, and text content in a $characters$ event in w is in the lexical space of the congruent $characters$ event in w'.

2.2. Language Representation

Next, the language representations are defined, i.e., dXVPAs accept datatyped event streams and cXVPAs accept document event streams.

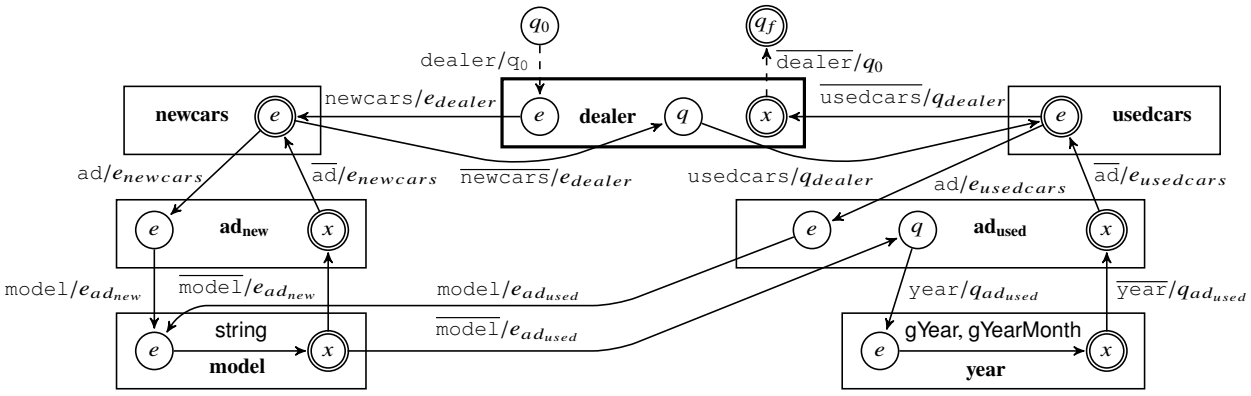


Figure 4. A dXVPA example

2.2.1. Datatyped XVPA. Kumar et al. [10] have introduced XVPAs as a model for stream validation, and dXVPAs extend XVPAs by datatypes.

Definition 5 (dXVPA)**.** A *dXVPA* A over $(\Sigma, M, \mu, T, \phi)$ is a tuple $A = (\{Q_m, e_m, X_m, \delta_m\}_{m \in M}, m_0, F)$. Σ is a set of qualified element names, M is a set of modules (equivalent to types in schemas), $\mu : M \to \Sigma$ is a surjective mapping that assigns elements to modules, T is a set of datatypes, and $\phi : T \to REG(U)$ assigns lexical spaces over Unicode.

For every module $m \in M$:

- Q_m is a finite set of module states
- $e_m \in Q_m$ is the module's entry state
- $X_m \subseteq Q_m$ are the module's exit states
- $\delta_m = \delta_m^{call} \uplus \delta_m^{int} \uplus \delta_m^{ret}$ are module transitions
  - $\delta_m^{call} \subseteq \{q_m \xrightarrow{\subset/q_m} e_n \mid n \in \mu^{-1}(\subset)\}$ and $\subset$ is the value of a *startElement* event
  - $\delta_m^{int} \subseteq \{q_m \xrightarrow{\tau} p_m \mid \tau \in T\}$ and τ is the value of a datatyped *characters* event
  - $\delta_m^{ret} \subseteq \{q_m \xrightarrow{\overline{\subset}/p_n} q_n \mid n \in \mu^{-1}(\subset)\}$, where $\overline{\subset}$ is the value of an *endElement* event, and the relation is deterministic, i.e., $q_n = q'_n$ whenever $q_m \xrightarrow{\overline{\subset}/p_n} q_n$ and $q_m \xrightarrow{\overline{\subset}/p_n} q'_n$

Module $m_0 \in M$ is the start module, $F = X_{m_0}$ are final states, automaton A satisfies the *single-exit property*, and all transitions satisfy *mixed-content restrictions*.

The set of states is also the stack alphabet. Call transitions save the current state on the stack and move to the entry of a module. Internal transitions leave the stack unchanged. Return transitions pop the stack and move from an exit state and to a state in the calling module.

Definition 6 (Single-exit property [10])**.** Let m, n be modules. The non-empty set $X_m \subseteq Q_m$ is a (p_n, q_n)-*exit*, or $X_m(p_n, q_n)$ for short, if $q_m \in X_m \iff q_m \xrightarrow{\overline{\subset}/p_n} q_n \in \delta_m^{ret}$. In XVPAs, X_m is the unique, *single exit* for module m.

In plain words, the return transitions are the same for every exit state in a module. The single-exit property ensures that the language of a module is always the same independent from the caller.

Definition 7 (Mixed-content restrictions)**.** Datatypes are not allowed to affect typing of elements, and the two *mixed-content restrictions* must be satisfied:

- *Datatype choice.* A datatype choice at state q_m must lead to a single next state, i.e., if $q_m \xrightarrow{\tau} q'_m \in \delta_m^{int}$ and $q_m \xrightarrow{\tau'} q''_m \in \delta_m^{int}$ then $q'_m = q''_m$.
- *Datatype sequence.* A return transition can only move to a state that is not a successor of a datatype choice, i.e., if $\exists q.q \xrightarrow{\overline{n}/p_m} q_m \in \delta_n^{ret}$ then $\forall q'.q' \xrightarrow{\tau} q_m \notin \delta_m^{int}$.

The restrictions guarantee that after a *characters* event, a module is either exited or another module is called, and there can never be two subsequent *characters* events.

Based on Kumar et al. [10], semantics of dXVPA A are characterized by VPA $A' = (Q, q_0, \{q_f\}, Q, \delta)$ over the visibly pushdown alphabet $(\Sigma \uplus T \uplus \overline{\Sigma})$:

$$Q = \{q_0, q_f\} \cup \bigcup_{m \in M} Q_m$$

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

A run of A' is denoted as $(q_0, \bot) \xrightarrow{w}_A (q, v)$, where w is a datatyped event stream, q is the reached state, and v is the stack. Event stream w is accepted if $q = q_f$ and $v = \bot$. Automaton A accepts language $L(A) = L(m_0) = L(A')$.

Kumar et al. [10] have also shown that for every EDTD an XVPA that accepts the same language can be constructed and vice-versa. This theorem and constructive proof can be extended to dXVPAs and EDTDs with datatypes, but this exceeds the scope of this paper.

Example 1. Consider the following EDTD using XSD datatypes. Element names are Σ = {`dealer`, `newcars`, `usedcars`, `ad`, `model`, `year`}, types are M = {*dealer*, *newcars*, *usedcars*, ad_{new}, ad_{used}, *model*, *year*}, the distinguished start type is *dealer*, and productions are:

$$
\begin{aligned}
d(dealer) &\mapsto newcars \cdot oldcars \\
d(newcars) &\mapsto ad^*_{new} \\
d(usedcars) &\mapsto ad^*_{used} \\
d(ad_{new}) &\mapsto model \\
d(ad_{used}) &\mapsto model \cdot year \\
d(model) &\mapsto \mathsf{string} \\
d(year) &\mapsto \mathsf{gYear} + \mathsf{gYearMonth}
\end{aligned}
$$

In XSD jargon, type *model* and *year* are simple types, and the others are complex types. Note that element `ad` has a different meaning depending on its context in a document. Figure 4 illustrates the language-equivalent dXVPA, and states q_0 and q_f are added to highlight VPA semantics. The dXVPA modules are exactly the types, module **model** is called by modules $\mathbf{ad_{new}}$ and $\mathbf{ad_{used}}$, and based on the stack, a run returns correctly.

2.2.2. Character-Data XVPA. A dXVPA cannot validate document event streams efficiently. If a dXVPA is in state p_m in module m and a *characters* event e encountered, the automaton can only proceed to state q_m if there exists an internal transition $p_m \xrightarrow{\tau} q_m \in \delta^{int}_m$ such that $lab(e) \in \phi(\tau)$ for the observed *characters* event in the document event stream. In the worst case, text $lab(e)$ needs to be buffered and acceptance needs to be checked for every possible datatype, i.e., $O(|lab(e)| \cdot |T|)$. A cXVPA unifies a datatype choice between two states into a single predicate $\psi \in \Psi$ over Unicode strings.

Definition 8 (cXVPA)**.** A *cXVPA* A over (Σ, M, μ, Ψ) is a tuple $A = (\{Q_m, e_m, X_m, \delta_m\}_{m \in M}, m_0, F)$ and adapts the dXVPA definition by $\delta^{int}_m : Q_m \times \Psi \to Q_m$ for internal transitions. At most one outgoing internal transition per state in a cXVPA is allowed, i.e., if $p_m \xrightarrow{\psi_i} q_m$ and $p_m \xrightarrow{\psi_j} q'_m$ then $q_m = q'_m$ and $\psi_i = \psi_j$.

The semantics of a cXVPA and its accepted language are given by the corresponding VPA over $(\Sigma \uplus \Psi \uplus \overline{\Sigma})$, where Ψ are predicates over Unicode strings, and a run on a event stream moves along an internal transition $p_m \xrightarrow{\psi} q_m \in \delta^{int}_m$ if $\psi(lab(e))$ holds in state p_m.

Theorem 1. *Every dXVPA can be translated into an equivalent cXVPA for checking acceptance of documents.*

To sketch the proof, the mixed-content restrictions in dXVPAs enforce that at most one successor state is reachable through internal transitions. This set of internal transitions can be replaced by a single predicate transition, where the predicate holds for texts accepted by the union of lexical spaces. Lexical spaces in Definition 3 are regular languages, where union is closed, and the predicate can therefore decide acceptance of a text in a single pass and linear time.

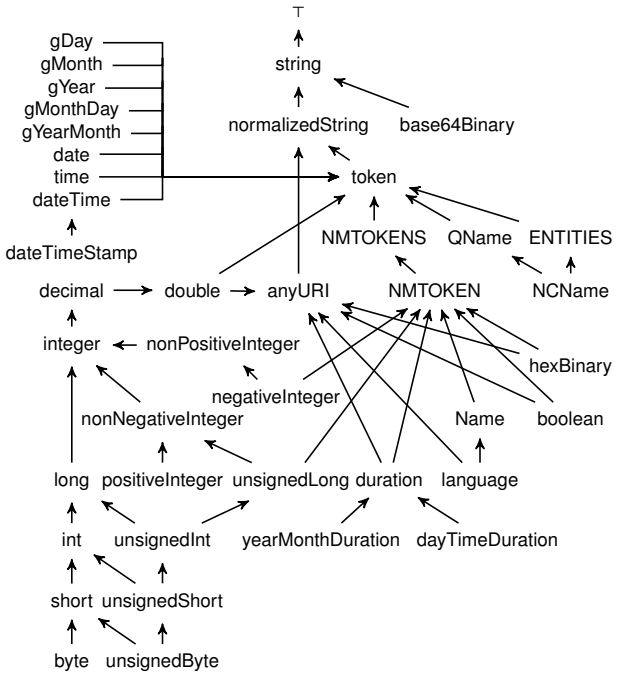


Figure 5. Ordering $\leq_{lex}$ on lexically distinct XSD datatypes

2.3. Datatype Inference from Text Content

A learner observes texts in *characters* events without knowing their language classes; however, this affects learnability, and as a first generalization step, a lexical datatype system is proposed. The lexical datatype system infers a set of minimally matching datatypes for a text content by lexical subsumption and a preference heuristic.

Definition 9 (Lexical datatype system)**.** A *lexical datatype system* is a tuple $dts = (T, \phi, \sim_s, \leq_s)$, where T and ϕ are according to Definition 3. Datatypes must be lexically distinct, i.e., $\phi(\tau) = \phi(\tau') \implies \tau = \tau'$, and ϕ imposes a partial ordering $\tau \leq_{lex} \tau' \iff \phi(\tau) \subseteq \phi(\tau')$. With respect to $\leq_{lex}$, T always contains a unique top datatype $\top$ that accepts any string, i.e., $\phi(\top) \mapsto U^*$. Equivalence relation $\sim_s : T \to K$ partitions datatypes into kinds K with respect to datatype semantics, and $\leq_s$ is an ordering on kinds. Moreover, the kinds impose a semantic ordering $\leq'_s$ on datatypes, i.e., $\tau \leq'_s \tau' \iff [\tau]_{\sim_s} \leq_s [\tau']_{\sim_s}$.

2.3.1. Lexical Subsumption. Figure 5 illustrates the datatype system based on primitive and build-in XSD datatypes [33]. These datatypes are characterized by Unicode regular expressions, and $\leq_{lex}$ is computed in software. Some datatypes are lexically indistinguishable and are therefore not included: double $=_{lex}$ float, NCName $=_{lex}$ ENTITY $=_{lex}$ ID $=_{lex}$ IDREF, and NMTOKENS $=_{lex}$ ENTITIES $=_{lex}$ IDREFS. For text contents, a learner should infer the smallest lexical space approximated by a set of datatypes (i.e., a datatype choice).

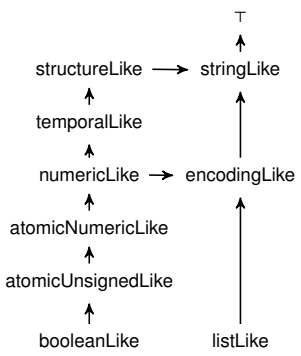


Figure 6. Ordering $\leq_s$ on kinds of lexical datatypes

Definition 10 (Minimally required datatypes)**.** The set of *minimally required datatypes* for a Unicode string w is the nonempty antichain $R \subseteq T$ of minimal datatypes with respect to $\leq_{lex}$ such that $\tau \in R \implies w \in \phi(\tau)$, and $\tau' <_{lex} \tau$ for $\tau \in R \implies w \notin \phi(\tau')$.

The minimally required datatypes are computed by in Algorithm 1 (*minLex*). The algorithm terminates after $|T|$ steps in the worst case. Membership of a string in lexical spaces is checked in topological sort order with respect to $\leq_{lex}$. To minimize the number of checks, a candiates set *cand* is maintained. If w is in some lexical space, w is also in all greater datatypes because $\leq_{lex}$ is transitive, and the up-set can be removed from *cand*. Furthermore, the topological order guarantees that the matched datatypes are minimal and incomparable. Algorithm *minLex* always returns a nonempty set because the $\top$ datatype has lexical space U^* and matches for any string.

Algorithm 1: *minLex*

Input: lexical datatype system $(T, \phi, \sim_s, \leq_s)$
Unicode string w
Output: minimally required datatypes $R \subseteq T$

$R := \emptyset$, $cand := T$
for τ **in** $topologicalSortOrder(T, \leq_{lex})$ **do**
  if $cand = \emptyset$ **then** done
  else if $\tau \in cand$ **and** $w \in \phi(\tau)$ **then**
    $R := R \cup \{\tau\}$
    $cand := cand \setminus \uparrow \tau$ w.r.t. $\leq_{lex}$

2.3.2. Preference Heuristic. Figure 5 already suggests that lexical spaces of XSD datatypes are often incomparable and ambiguous. This leads to ambiguous datatype choices, e.g., $minLex(\texttt{false}) = \{\text{language}, \text{boolean}, \text{NCName}\}$. The antichain is lexically correct, but some datatypes are semantically more specific and preferred over others. A second step in datatype inference is therefore to drop the least specific datatypes from minimally required datatypes.

The proposed heuristic captures type derivations in the XSD standard and datatype semantics in an ordering $\leq_s$ for kinds of datatypes. Figure 6 illustrates the ordering, and kinds are defined as:

stringLike = {string, normalizedString, token, ENTITY, ID, IDREF, NMTOKEN}
listLike = {ENTITIES, IDREFS, NMTOKENS}
structureLike = {anyURI, NOTATION, QName, Name, language, NCName}
encodingLike = {base64Binary, hexBinary}
temporalLike = {gDay, gMonth, gYear, gYearMonth, gMonthDay, date, time, duration, dayTimeDuration, yearMonthDuration, dateTime, dateTimeStamp}
numericLike = {nonPositiveInteger, nonNegativeInteger, positiveInteger, decimal, integer, negativeInteger}
atomicNumericLike = {float, double, long, int, short, byte}
atomicUnsignedLike = {unsignedLong, unsignedInt, unsignedShort, unsignedByte}
booleanLike = {boolean}

Note that there is also a distinguished $\top$ kind for the $\top$ datatype for completeness. Algorithm 2 ($pref$) compares pairs of minimally required datatypes, and if two datatypes are comparable with respect to $\leq_s$, the greater datatype is dropped from the set. The resulting set R' is still an antichain of datatypes with respect to $\leq_{lex}$.

Algorithm 2: *pref*

Input: lexical datatype system $(T, \phi, \sim_s, \leq_s)$
datatypes $R \subseteq T$
Output: preferred datatypes $R' \subseteq T$

$R' := R$
for τ, τ' **in** R **and** $\tau \neq \tau'$ **do**
  if $[\tau]_{\sim_s} <_s [\tau']_{\sim_s}$ **then** $R' := R' \setminus \{\tau'\}$

2.3.3. Datatyped Event Stream for Learning. For learning, every text in a document event stream is mapped to its set of minimally required datatypes:

$$minReq(w) = pref(minLex(w)) \quad \text{for string } w \tag{1}$$

$$dtyped(e) = \begin{cases} minReq(lab(e)) & \text{if } characters \\ e & \text{for other events} \end{cases} \tag{2}$$

The proposed lexical datatype system is not strictly bounded to XSD datatypes and extensible. Only a $\top$ datatype with respect to $\leq_{lex}$ is required.

2.3.4. Combining Datatype Choices. For aggregating datatypes, let v, w be to strings over Unicode. The minimally required datatypes that accept both strings are:

$$minReq(v, w) = \max_{\leq_{lex}} minReq(v) \cup minReq(w) \tag{3}$$

The $\max_{\leq_{lex}}$ operation guarantees a nonempty antichain with respect to $\leq_{lex}$ that cover both strings.

Example 2. Let $S = \{\texttt{1}, \texttt{0}, \texttt{true}, \texttt{33}\}$ be Unicode strings, then $minReq(S) = \{\text{boolean}, \text{unsignedByte}\}$.

2.4. The Incremental Learner

Theorem 2 (Gold [29])**.** *The language class of unrestricted regular expressions is not learnable in the limit from positive examples only.*

This result prohibits exact grammatical inference from example documents only. A learner therefore needs to assume a learnable language class that covers most cases in practice, and for more expressive languages, the learner can only return an approximation. Expressiveness of XML has been studied in terms of schema complexity, and two aspects are relevant:

- *Simplicity of regular expressions.* Bex et al. [34] have examined 202 DTDs and XSDs and conclude that the majority of regular expressions in schema productions are simple because types occur only a small number of times in expressions.
- *Locality of typing contexts.* Martens et al. [17] have studied 819 DTDs and XSDs from the web and XML standards, and typing elements in 98% is local, i.e., the type of an element only depends on its parent.

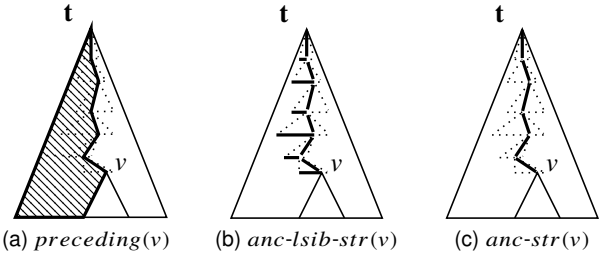
Bex et al. [35] define the class of single-occurrence regular expressions (SOREs) to capture the simplicity. In a SORE, a symbol occurs at most once, and the majority of schema productions in the wild belong to this class. SOREs generate a 2-testable regular language, and k-testable regular languages [36] are known to be efficiently learnable from positive examples only.

A k-testable regular language is fully characterized by a finite set of allowed substrings of length k, and learning is collecting the substrings. This can be done efficiently by constructing a prefix tree acceptor (PTA), i.e., a deterministic finite automaton (DFA) that accepts exactly the examples, and *naming the states* according to the string prefixes that lead to them. Merging states whose names share the same $(k-1)$-length suffix then generalizes the PTA.

State merging can be done implicitly while constructing the PTA, and the proposed learner utilizes this idea by embedding typing information in state names, so state merging generalizes types and content models simultaneously.

2.4.1. Typing Mechanisms. Typing can be thought of as a function that determines the type of an element from the element name and other elements in the document. To understand typing, Murata et al. [37] and Martens et al. [17] have investigated schema expressiveness with respect to determinism. Their motivation originates from composition, e.g., let A, B be schemas that allow deterministic typing, then a composition $A + B$ is not necessarily deterministic.

Figure 7 illustrates typing mechanisms by representing the infoset of a document without text contents as a tree, and $lab(v)$ is the qualified element name of node v. Efficient stream processing requires deterministic typing, and Martens et al. [17] therefore define *1-pass preorder typing* (1PPT): a schema allows 1PPT if the type of every node v can be determined from the $preceding(v)$ subtree as shown in Figure 7a. Furthermore, the authors show that typing based on the ancestor-sibling string $anc\text{-}lsib\text{-}str(v)$ is sufficient for the 1PPT property.



(a) $preceding(v)$ (b) $anc\text{-}lsib\text{-}str(v)$ (c) $anc\text{-}str(v)$

Figure 7. Typing of an element

Let $lsib(v) = lab(u_1) \cdots lab(u_m) \cdot lab(v)$ be a left-sibling string, where $u_1, \ldots, u_m$ are the left siblings of v. The ancestor-sibling string is $anc\text{-}lsib\text{-}str(v) = lsib(i_1)\#lsib(i_2)\#\cdots\#lsib(i_n)$ such that i_1 is the root node, $i_n = v$, and i_{j+1} is a child of i_j. An example is shown in Figure 7b.

Element Declaration Consistency (EDC) and Unique Particle Attribution (UPA) are syntactic restrictions for productions in XSD to ensure deterministic typing. These restrictions are tighter than they need to be for the 1PPT property: node v is typed by the ancestor string $anc\text{-}str(v)$ as shown in Figure 7c. The ancestor string is defined as $anc\text{-}str(v) = lab(i_1) \cdot lab(i_2) \cdots lab(i_n)$, where i_1 is the root node, $i_n = v$, and i_{j+1} is a child of i_j.

2.4.2. Incremental Update. Named states in Algorithm 3 ($incWeightedVPA$) are the foundation for state merging. The algorithm iterates over document event stream w in a single pass and returns an updated VPA and counters. In a run, the algorithm maintains a stack, collects element names, and for every event, a next state is derived from three *state naming functions* with the signatures $call : Q \times \Sigma \to Q$, $int : Q \to Q$, and $ret : Q \times Q \times \Sigma \to Q$. A transition is then stored in the VPA to connect the current to the next state.

Initially, the intermediate VPA has a single nonaccepting start state (ϵ, ϵ), no transitions, and counters are zero. New states and transitions are created inductively from the start state, and counters are increased. Two state naming schemes according to typing mechanisms are proposed.

Definition 11 (State naming)**.** A state is a pair (u, v) of typing context u and left-sibling string v. Symbols # and \$ are a left-sibling separator and a placeholder for text.

- *Ancestor-based.* A state $(u, v) \in (\Sigma^* \times (\Sigma \cup \{\$\})^*)$ is a pair of ancestor string and left-sibling string.
- *Ancestor-sibling-based.* A state $(u, v) \in ((\Sigma \cup \{\$, \#\})^* \times (\Sigma \cup \{\$\})^*)$ is a pair of ancestor-sibling string and left-sibling string.

For learning within the XSD language class, states must be ancestor based, and for learning beyond XSD but within the 1PPT language class, states must be ancestor-sibling based.

Algorithm 3: *incWeightedVPA*

Input: VPA $A = (Q, q_0, F, Q, \delta)$ over $\Sigma \uplus T \uplus \overline{\Sigma}$
lexical datatype system $(T, \phi, \sim_s, \leq_s)$
state naming functions $call$, int, ret
counters ω_Q, ω_F, ω_δ
document event stream w
Output: updated VPA A and counters ω_Q, ω_F, ω_δ

1 $s := \bot$ // empty stack
2 $q := q_0$ // current state
3 **for** e **in** $dtyped(w)$ **do**
4 **switch** $eventType(e)$ **do**
5 **case** $startElement$
6 $\Sigma := \Sigma \cup \{lab(e)\}$
7 $q' := call(q, lab(e))$
8 $\omega_Q(q') := \omega_Q(q') + 1$
9 $\delta^{call} := \delta^{call} \cup \{q \xrightarrow{lab(e)/q} q'\}$
10 $\omega_\delta(q \xrightarrow{lab(e)/q} q') := \omega_\delta(q \xrightarrow{lab(e)/q} q') + 1$
11 $s := s \cdot q$
12 $q := q'$
13 **case** $endElement$
14 **let** $vp = s$ // p is top
15 $q' := ret(q, p, lab(e))$
16 $\omega_Q(q') := \omega_Q(q') + 1$
17 $\delta^{ret} := \delta^{ret} \cup \{q \xrightarrow{\overline{lab(e)}/p} q'\}$
18 $\omega_\delta(q \xrightarrow{\overline{lab(e)}/p} q') := \omega_\delta(q \xrightarrow{\overline{lab(e)}/p} q') + 1$
19 $s := v$
20 $q := q'$
21 **case** $characters$
22 $q' := int(q)$
23 $\omega_Q(q') := \omega_Q(q') + 1$
24 $\delta^{int} := \delta^{int} \cup \{q \xrightarrow{\tau} q' \mid \tau \in lab(e)\}$
25 **for** $\tau \in lab(e)$ **do**
 $\omega_\delta(q \xrightarrow{\tau} q') := \omega_\delta(q \xrightarrow{\tau} q') + 1$
26 $q := q'$
27 $F := F \cup \{q\}$
28 $\omega_F(q) := \omega_F(q) + 1$

2.4.3. Local State Merging. When state naming functions $call$, int, and ret return full ancestor, ancestor-sibling, and left-sibling strings, every event stream prefix would create a unique state. The resulting VPA would accept only the documents learned so far, similar to a PTA in regular languages. For learning, state merging is implicitly embedded in the state naming functions by returning equivalence classes.

The distinguishing criterion is *locality*: two states are equal if they share the same l-local typing context and k-local left siblings. The naming functions are therefore:

$$int_{k,l}((u, v)) = (u, \sigma_k(v \cdot \$)), \quad (4)$$

$$ret_{k,l}(q, p, e) = (\pi_1(p), \sigma_k(\pi_2(p) \cdot lab(e))), \quad (5)$$

$$call^{as}_{k,l}((u, v), e) = (\sigma_l(u \cdot lab(e)), \epsilon), \quad (6)$$

$$call^{als}_{k,l}((r_1\# \cdots \#r_n, v), e) = (r_{n-l+1}\# \cdots \#r_n\#\sigma_k(v \cdot lab(e)), \epsilon), \quad (7)$$

The suffix function $\sigma_i(w)$ returns the i-length suffix of sequence w, and $\pi_i(x)$ denotes the ith field of tuple x. For *characters* events, $int_{k,l}$ is the same for ancestor- (as) and ancestor-sibling-based (als) naming; the typing context remains and \$ is appended to the left siblings as a placeholder. Using a placeholder for the next state is sound because of the mixed-content restrictions in Definition 7. For *endElement* events, $ret_{k,l}$ is also the same for both naming schemes; the next state gets the typing context from stack state p and a new left sibling is added to the ones in p. In case of a *startElement* event, a new typing context is created and left siblings are set to empty.

Parameters k and l specify the hypothesis space of the learner. For the lower bound $k = l = 1$, the learnable language class is a strict subclass of DTDs. For $l \geq 1, k = 1$, both state naming schemes produce congruent automata, and the learnable language class is a strict subclass of XSD. Increasing the parameters also increases the learnable language class, but also the state space grows, and more examples are necessary for convergence.

2.4.4. Generating a dXVPA. The intermediate VPA and its counters still need to be translated into a dXVPA. Algorithm 4 (*trim*) creates a new intermediate VPA without zero-weight states and transitions. Furthermore, *trim* updates internal datatype transitions between two states to a correct antichain of datatypes.

Algorithm 4: *trim*

Input: VPA $A = (Q, q_0, F, Q, \delta)$ over $\Sigma \uplus T \uplus \overline{\Sigma}$
lexical datatype system $(T, \phi, \sim_s, \leq_s)$
counters ω_Q, ω_F, ω_δ
Output: VPA $A' = (Q', q_0, F', Q', \delta')$

1 $\delta'^{call} := \delta^{call} \setminus \{q \xrightarrow{c/q} q' \mid \omega_\delta(q \xrightarrow{c/q} q') = 0\}$
2 $\delta'^{ret} := \delta^{ret} \setminus \{q \xrightarrow{\overline{c}/q} q' \mid \omega_\delta(q \xrightarrow{\overline{c}/q} q') = 0\}$
3 $\delta'^{int} := \delta^{int} \setminus \{q \xrightarrow{\tau} q' \mid \omega_\delta(q \xrightarrow{\tau} q') = 0\}$
4 $\delta''^{int} := \emptyset$
5 **foreach** $\{(q, q') \mid \exists \tau . q \xrightarrow{\tau} q' \in \delta'^{int}\}$ **do**
6 **let** $R = \{\tau \mid q \xrightarrow{\tau} q' \in \delta'^{int}\}$
7 $\delta''^{int} := \delta''^{int} \cup \{q \xrightarrow{\tau} q' \mid \tau \in \max_{\leq_{lex}} R\}$
8 $\delta' = \delta'^{call} \uplus \delta'^{ret} \uplus \delta''^{int}$
9 $Q' := Q \setminus \{q \mid \omega_Q(q) = 0\}$
10 $F' := F \setminus \{q \mid \omega_F(q) = 0\}$

Algorithm 5 ($genXVPA$) generates a valid dXVPA from a trimmed intermediate VPA. States are partitioned into modules based on their typing context. The initial module m_0 is the one called from state (ϵ, ϵ). States are partitioned into their respective modules, and the $call$ function guarantees that the entry of module m is always state (m, ϵ). Return transitions are added to all module exit states to satisfy the single-exit property (Line 10). Mapping μ is then derived from the call transitions that point to module entry states.

Algorithm 6 (*minimize*) merges congruent modules. Kumar et al. [10] have shown that XVPA modules can be translated to DFAs, and this construction can be easily extended

Algorithm 5: $genXVPA$

Input: VPA $A = (Q, q_0, F, Q, \delta)$ over $\Sigma \uplus T \uplus \overline{\Sigma}$
lexical datatype system $(T, \phi, \sim_s, \leq_s)$
Output: dXVPA A' over $(\Sigma, M, \mu, T, \phi)$, where
$A' = (\{Q_m, e_m, X_m, \delta_m\}_{m \in M}, m_0, X_{m_0})$

$M := \{u \mid (u, v) \in Q \text{ and } u \neq \epsilon\}$
$m_0 := u$ such that $q_0 \xrightarrow{c/q_0} (u, \epsilon) \in \delta^{call}$
for $m \in M$ **do**
  $Q_m := \{(u, v) \in Q \mid u = m\}$
  $e_m := (m, \epsilon)$
  $X_m := \{q \in Q_m \mid q \xrightarrow{\overline{c}/p} q' \in \delta^{ret}\}$
  $\delta_m^{call} := \{q \xrightarrow{c/q} q' \in \delta^{call} \mid q \in Q_m\}$
  $\delta_m^{int} := \{q \xrightarrow{\tau} q' \in \delta^{int} \mid q, q' \in Q_m\}$
  $\delta_m^{ret} := \{q \xrightarrow{\overline{c}/p} q' \in \delta^{ret} \mid q \in Q_m\}$
  $\delta_m^{ret} := \delta_m^{ret} \cup \{q \xrightarrow{\overline{c}/p} q' \mid q \in X_m \text{ and } \exists q_m.q_m \xrightarrow{\overline{c}/p} q' \in \delta_m^{ret}\}$
  $\delta_m = \delta_m^{call} \uplus \delta_m^{call} \uplus \delta_m^{call}$
  if $\exists q.q \xrightarrow{c/q} e_m \in \delta^{call}$ **then** $\mu(m) := c$
$A' := minimize(A')$

to dXVPA modules. The algorithm compares modules m and n, and if they are reachable by the same element name and have congruent DFAs, n folds into m by redirecting calls and returns to corresponding states in Q_m. The state bijection φ follows from bisimulation of the DFAs, and after a fold, *minimize* keeps restarting until no fold occurs.

2.4.5. Learner Properties. Algorithm 7 assembles the incremental learner. Maintaining the intermediate VPA prevents information loss from premature minimization of dXVPA modules. For a given lexical datatype system, three naming functions, and parameters k and l, the incremental learner computes a dXVPA from document event stream w. The equivalent cXVPA can then check acceptance of documents.

Theorem 3. *The learner is (1) incremental, (2) set-driven, (3) consistent, (4) conservative, (5) strong-monotonic, and identifies a subclass of 1PPT mixed-content XML.*

Incremental learning follows from Algorithm 7. Extending to a set-driven learner is possible by calling $incWeightedVPA$ repeatedly for all examples and generating the dXVPA after the last example. A set-driven learner is insensitive to the order of presented examples, and this property follows from state naming and treating states and transitions as sets. A learner is consistent if all learned examples are also accepted, conservative if a current hypothesis is kept as long as no contradicting evidence is presented, and strong-monotonic if the language increases with every example [38], [39]. These properties follow from updating sets of states and transitions in the intermediate VPA using the state naming functions for implicit state merging. States and call and return transitions are never deleted, and new ones are only added if new evidence is presented. Also, an

Algorithm 6: $minimize$

Input: dXVPA A over $(\Sigma, M, \mu, T, \phi)$, where
$A = (\{Q_m, e_m, X_m, \delta_m\}_{m \in M}, m_0, X_{m_0})$
Output: minimized dXVPA A

while $\exists m \exists n.m, n \in M$ *and* $m \neq n$ *and* $\mu(m) = \mu(n)$ *and* $DFA_m \simeq DFA_n$ **do**
  let $\varphi : Q_n \to Q_m$ `// bisimulation`
  for $q_n \xrightarrow{\overline{c}/p_i} q_i \in \delta_n^{ret}$ **do**
    $\delta_i^{call} := \delta_i^{call} \setminus \{p_i \xrightarrow{c/p_i} e_n\} \cup \{p_i \xrightarrow{c/p_i} e_m\}$
    $\delta_m^{ret} := \delta_m^{ret} \cup \{x_m \xrightarrow{\overline{c}/p_i} q_i \mid x_m \in X_m\}$
  for $q_n \xrightarrow{\overline{c}/q_n} e_i \in \delta_n^{call}$ **do**
    $\delta_i'^{ret} = \emptyset$
    for $q_i \xrightarrow{\overline{c}/p_j} q_j \in \delta_i^{ret}$ **do**
      if $j = n$ **then** $\delta_i'^{ret} := \delta_i'^{ret} \cup \{q_i \xrightarrow{\overline{c}/\varphi(p_j)} \varphi(q_j)\}$
      else $\delta_i'^{ret} := \delta_i'^{ret} \cup \{q_i \xrightarrow{\overline{c}/p_j} q_j\}$
    $\delta_i^{ret} := \delta_i'^{ret}$
  if $n = m_0$ **then** $m_0 := m$
  $M := M \setminus \{n\}$ `// remove module n`
  $\mu(n) := \emptyset$

Algorithm 7: Incremental learner

Input: persistent VPA A
lexical datatype system $dts = (T, \phi, \sim_s, \leq_s)$
persistent counters $\omega = (\omega_Q, \omega_F, \omega_\delta)$
state naming $f = (int_{k,l}, call_{k,l}, ret_{k,l})$ with k, l
document event stream w
Output: dXVPA A'

initially, $A = (\{(\epsilon, \epsilon)\}, (\epsilon, \epsilon), \emptyset, \{(\epsilon, \epsilon)\}, \emptyset)$
$A, \omega := incWeightedVPA(A, dts, f, \omega, w)$
$A' := genXVPA(trim(A, dts, \omega))$

internal transition on datatype τ is only removed if a new transition on τ' is added and τ' covers τ.

A learned dXVPA is always deterministic because of the restriction to k-l-local 1PPT, and checking acceptance using the corresponding cXVPA is therefore linear in the length of the document.

2.5. Anomaly Detection Refinements

Training data could be the target of a poisoning attack [40]. Poisoning attacks could be uncovered in the future or stay hidden. Operations for *unlearning* a once learned example and *sanitization* by removing low-frequent transitions and states from the intermediate VPA are therefore proposed.

Algorithm 8 (*unlearn*) simulates a run on a document event stream, traverses the intermediate VPA, and counters are decremented. The document must have been learned before at an earlier time for the operation to be sound. In case of a *characters* event, one particular datatype of the minimally matching datatypes is picked to locate the next

Algorithm 8: *unlearn*

Input: VPA $A = (Q, q_0, F, Q, \delta)$ over $\Sigma \uplus T \uplus \overline{\Sigma}$
lexical datatype system $dts = (T, \phi, \sim_s, \leq_s)$
counters ω_Q, ω_F, ω_δ
document event stream w
Output: updated VPA A and counters ω_Q, ω_F, ω_δ

$s := \bot$ // empty stack
$q := q_0$ // current state
for e **in** $dtyped(w)$ **do**
  switch $eventType(e)$ **do**
    case *startElement*
      $q' := \delta^{call}(q, lab(e))$
      $\omega_Q(q') := \omega_Q(q') - 1$
      $\omega_\delta(q \xrightarrow{lab(e)/q} q') := \omega_\delta(q \xrightarrow{lab(e)/q} q') - 1$
      $s := s \cdot q$
      $q := q'$
    case *endElement*
      let $vp = s$ // p is top
      $q' := \delta^{ret}(q, lab(e), p)$
      $\omega_Q(q') := \omega_Q(q') - 1$
      $\omega_\delta(q \xrightarrow{\overline{lab(e)}/p} q') := \omega_\delta(q \xrightarrow{\overline{lab(e)}/p} q') - 1$
      $s := v$
      $q := q'$
    case *characters*
      $q' := \delta^{int}(q, \tau)$ for some $\tau \in lab(e)$
      $\omega_Q(q') := \omega_Q(q') - 1$
      for $\tau \in lab(e)$ **do**
      $\omega_\delta(q \xrightarrow{\tau} q') := \omega_\delta(q \xrightarrow{\tau} q') - 1$
      $q := q'$
$\omega_F(q) := \omega_F(q) - 1$
$A := trim(A, dts, \omega_Q, \omega_F, \omega_\delta)$

Algorithm 9: *sanitize*

Input: VPA $A = (Q, q_0, F, Q, \delta)$ over $\Sigma \uplus T \uplus \overline{\Sigma}$
lexical datatype system $dts = (T, \phi, \sim_s, \leq_s)$
counters ω_Q, ω_F, ω_δ
Output: updated VPA A' and counters ω'_Q, ω'_F, ω'_δ

for *any defined transition* x **do** $\omega'_\delta(x) := \omega_\delta(x) - 1$
for $q \in Q$ **do**
  $\omega'_Q(q) := \sum_{\text{transition } x \text{ to } q} \omega'_\delta(x)$
  if $q \in F$ **then** $\omega'_F(q) := \omega'_Q(q)$
$A' := trim(A, dts, \omega'_Q, \omega'_F, \omega'_\delta)$
let Q_u be the unreachable states in A'
if $Q_u \neq \emptyset$ **then**
  if $(F' \setminus Q_u) = \emptyset$ **then** // revert changes
    $\omega'_Q := \omega_Q$
    $\omega'_F := \omega_F$
    $\omega'_\delta := \omega_\delta$
    $A' := A$
  else // remove unreachable states
    for $q \in Q_u$ **do**
      for *any transition* x *to* q **do** $\omega'_\delta(x) := 0$
      $\omega'_Q(q) := \omega'_F(q) := 0$
    $A' := trim(A, dts, \omega'_Q, \omega'_F, \omega'_\delta)$

state and decrement all internal transitions between current and next state.

Under the assumption that hidden poisoning attacks in training data are rare [40], Algorithm 9 (*sanitize*) removes low frequent states and transitions. The algorithm has two stages: first, counters of transitions are decremented and counters of states are recomputed; and second, a trimmed VPA is generated for identifying unreachable states. If no final state is reachable, all weight counters are restored because sanizitation is not applicable.

It should be stressed that sanitization should only be applied after a sufficiently many examples have been learned. The operation violates the consistent, conservative, and strong-monotonicity properties of the learner. Also, after a *sanitize* operation, *unlearn* becomes unsound forever.

3. Experimental Evaluation

The proposed approach has been implemented in Scala 2.11.7 using the `dk.brics.automaton` [41] library for datatype inference and predicates in cXVPAs. Two aspects of performance are considered: detection performance and learning progress.

3.1. Measures

By assuming a binary classification setting for normal documents and attacks, the following performance measures are computed: recall/detection rate (Re), false-positive rate (FPR), precision (Pr), and F_1 for overall performance in a single value [42]. Identification in the limit has a convergence point; however, in practice, convergence can only be estimated from counting the mind changes between incremental steps [43].

Definition 12 (Mind changes)**.** Mind changes MC_i are the sum of states and transitions whose counters switched from zero to one after learning document w_i.

Parameters k and l embody a strong combinatorial upper bound on the number of states and transitions for a finite number of elements. In the worst case of randomness, convergence is reached when the state space is fully saturated.

3.2. Datasets

Table 1 summarizes the properties of the four datasets. The incremental learner learns a dXVPA from training data, and performance is measured by evaluating the testing data. Datasets Carsale and Catalog have been synthetically generated using the stochastic XML generator ToXGene by Barbosa et al. [44]. Furthermore, for providing a realistic setting, a *VulnShopService* and a randomized *VulnShopClient* have been implemented for collecting SOAP messages. This Apache Axis2 1.6.0 SOAP/WS-* web service uses Apache Rampart 1.6.0 for WS-Security and provides two operations: regular shop orders (dataset VulnShopOrder) and

TABLE 1. EVALUATION DATASETS

Dataset	Training Normal	Testing Normal	Testing Attack	XML tampering	High node count	Coercive parsing	Script injection	Command injection	SQL injection	SSRF	XML injection	Signature wrapping
Carsale	50	1000	17	1	3	2	3	2	2	1	3	0
Catalog	100	2000	17	1	3	2	3	2	2	2	2	0
VulnShopOrder	200	2000	28	2	4	2	5	3	5	3	4	0
VulnShopAuthOrder	200	2000	78	0	0	0	0	0	0	0	0	78

digitally signed shop orders (dataset VulnShopAuthOrder). For realism, the implementation strictly followed the Axis2 and Rampart examples. The business logic is implemented in Java beans, and Java2WSDL automatically generates an Axis2 service. Names for operations and Java classes have been deliberately chosen to exceed DTD expressiveness.

Attacks in synthetic datasets were added manually. Attacks in the simulated datasets are recordings of actual attacks that were executed manually or automatically by WS-Attacker-1.7 [31] for Denial-of-Service (high node count, coercive parsing) and signature wrapping.

3.3. Performance

3.3.1. Baseline. Explicit schema validation using Apache Xerces 2.9.1 establishes a baseline, and results are listed in Table 2. The schemas for the Carsale and Catalog datasets are extracted from ToXGene configurations, and simple types have been set to datatype string or more specific datatypes when applicable. The VulnShopOrder and VulnShopAuthOrder datasets need a schema collection from the web service because of several WS-* standards.

TABLE 2. BASELINE PERFORMANCE USING SCHEMA VALIDATION

Dataset	Pr	Re	FPR	F_1
Carsale	100%	82.35%	0%	90.32%
Catalog	100%	76.47%	0%	86.67%
VulnShopOrder	100%	50%	0%	66.67%
VulnShopAuthOder	undef.	0%	0%	undef.

The schemas in synthetic datasets are free from extension points, and schema validation achieves good performance as expected. The baseline for the simulated *VulnShopService* however illustrates the effect of extension points. Half of the attacks in VulnShopOrder can be identified because of structural violations or datatype mismatches, but all Denial-of-Service attacks at extension points pass. Furthermore, no signature wrapping attack can be identified.

3.3.2. Detection Performance. Table 3 summarizes the best detection performance results by the proposed learner and validator for smallest parameters k and l. The best parameters have been found in a grid search over values $k, l \in \{1, \ldots, 5\}$ and the two state naming schemes.

TABLE 3. BEST PERFORMANCE USING ANCESTOR-BASED STATES

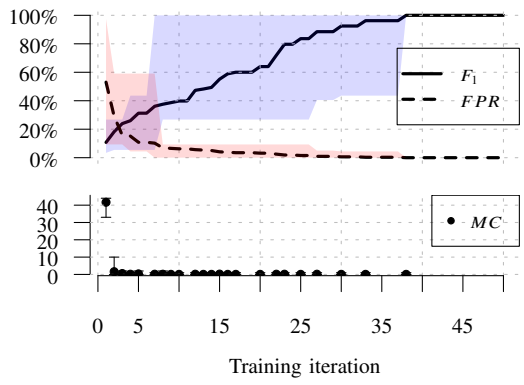
Dataset	k l	Pr	Re	FPR	F_1
Carsale	1 1	100%	100%	0%	100%
Catalog	1 1	100%	82.35%	0%	90.32%
VulnShopOrder	1 1	100%	92.86%	0%	96.30%
VulnShopAuthOrder	1 1	100%	100%	0%	100%

The proposed language-based anomaly detection approach exceeds the baseline. No false positives are detected, and the best results are already achieved with the simplest parameters, i.e., ancestor-based state naming and $k = l = 1$. All structural anomalies caused by attacks are detected. It should be stressed that $k = l = 1$ is a good-enough approximation of the language to identify attacks but more sound types are inferred for $l > 1$.

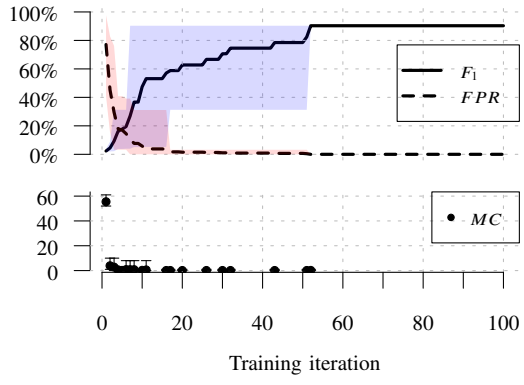
Some script and command injection attacks cannot be identified. These attacks have in common that exploitation code appears in texts and uses CDATA fields to hide special characters, e.g., angled brackets and ampersands, from the XML parser's lexical analysis. The lexical datatype system is too coarse in this case because the inferred datatype normalizedString allows the attack-revealing characters.

3.3.3. Learning Progress. Learning progress is measured in mind changes, and Figure 8 summarizes the fastest converging settings for the four datasets. The converged detection performance coincides with Table 3. For every training iteration, the learner randomly samples a document without replacement from training data, and the validator checks acceptance of testing data for measuring performance. Because of randomness, runs are repeated 15 times, average values for F_1 and FPR are computed, and the error regions in the plots illustrate minimal and maximal values.

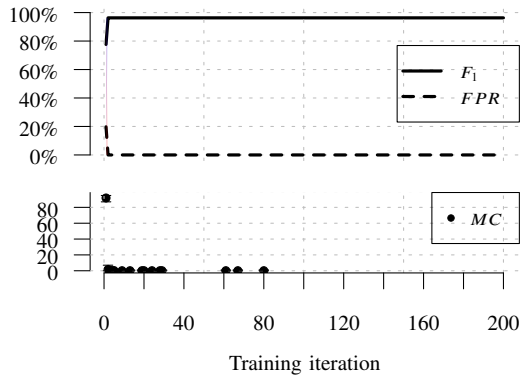
The first training example causes many mind changes because there are no states and transitions yet. The strong-monotonicity property guarantees that detection performance either increases or stays the same after learning an additional example. Furthermore, detection performance has an upper bound because the learner is constrained to a learnable language class (parameterized by the naming scheme, k, and l) which either covers or approximates the true language. In practice, detection performance cannot be measured, but mind changes can be observed. As shown in



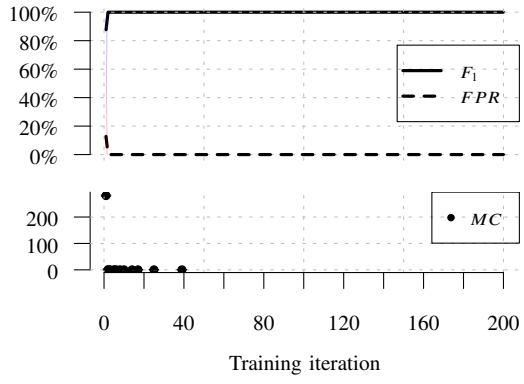
(a) Carsale, anc.-sib. naming, $k = 1, l = 2$



(b) Catalog, anc.-sib. naming, $k = 1, l = 2$



(c) VulnShopOrder, anc.-sib. naming, $k = 1, l = 2$



(d) VulnShopAuthOrder, anc.-sib. naming, $k = 1, l = 2$

Figure 8. Learning progress highlights

the figures, mind changes become less frequent over time, and a long period of zero mind changes can be seen as a heuristic for convergence.

The quick convergence in Figure 8c and 8d stems from the simplicity of the language automatically generated by Java2WSDL. The generator only supports sequential (member variables) and iterating (arrays) productions but no choice. A few examples are sufficient for finding a good-enough approximation with small parameters k and l.

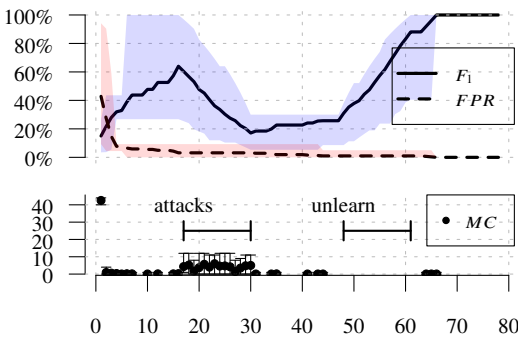
3.3.4. Unlearning and Sanitization. Unlearning reverses learning, and Figure 9a illustrates the effects of the operation. In this scenario, a successful attacker was able to feed poisoning attacks to the learner, and detection performance collapses. At a later moment of time, the poisoning attacks were identified by an expert and consequently unlearned. The detection performance recovers accordingly.

Sanitization prunes the intermediate VPA of low-frequent states and transitions. A single hidden poisoning attack is placed after learning 10% of the training iterations. After 75% of the training iterations, sanitization is performed and learning continues. Figure 9b shows the learning progress and effects of sanitization. In at least one of the 15 trials, the learner did not have a stable language representation at the moment of sanitization, good knowledge was removed, and detection performance collapses. More learning after sanitization was necessary to recover, and in the end, the poisoning attack was removed. Knowledge gained in between poisoning and unlearning is not lost.

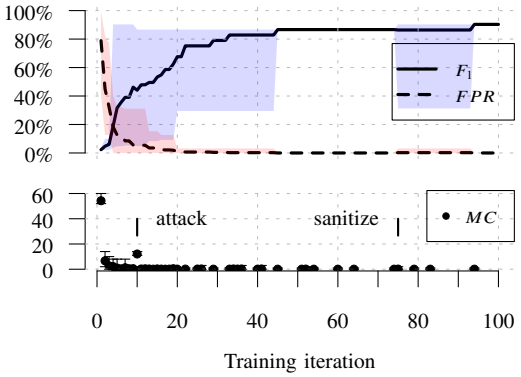
4. Related Work

This work focuses on a language representation for stream validation because of large documents and open-ended streams (e.g., XMPP). Stream validation has been introduced by Segoufin and Vianu [45] using finite-state machines and pushdown automata. Kumar et al. [10] consider document event streams as visibly pushdown languages (VPLs), a class of deterministic context-free languages, and the authors propose XVPAs as a better representation. XVPAs have therefore been extended by datatypes for mixed-content XML.

Schema inference from a set of documents beyond DTD expressiveness focuses on finding simple regular expressions for schema productions. Chidlovskii [46] and Mlýnková and Nečaský [47] propose grammar-based approaches, where infoset tree nodes turned into productions. These productions are then generalized by determinism constraints [46] and heuristics [47]. Bex et al. [48] propose schema inference in terms of tree automata, where up to k ancestor elements in a document characterize a type. This work has motivated the use of locality as a generalization strategy. Lexical subsumption for datatype inference was fist mentioned by Chidlovskii [46] and Hegewald et al. [49]; however, not all XSD datatypes have been considered. The proposed approach considers a datatype choice instead of a single datatype, all distinguishable XSD datatypes are used, and a preference heuristic tightens a choice.



(a) Unlearning in Carsale anc.-sib., $k = 1, l = 2$



(b) Sanitization in Catalog anc.-sib., $k = 1, l = 2$

Figure 9. Effects of unlearning and sanitization

With respect to anomaly detection, Menahem et al. [50] propose a feature extraction process for documents, so existing machine-learning algorithms can be applied. However, structural information is lost, the focus is on text content, and a schema is assumed to be available. This direction has therefore not been further pursued. Another anomaly detection approach specifically for tree structures is based on geometry. Rieck [51] introduces tree kernels as measures of shared information between two parse trees derived from network protocols. The kernels enable global and local anomaly detection, and this method could eventually be extended to XML infoset trees. Global anomaly detection finds a volume-minimal sphere that encloses the vector-embedded trees, and local anomaly detection computes kernel-based distances to the nearest neighbors. Approximate tree kernels [52] are then a trade-off for reducing computational costs. However, this method also assumes a tree data model which conflicts with streaming requirements.

5. Conclusions

Schema validation is a first-line defense against XML attacks, but extension points can render schema validation ineffective. Jensen et al. [8] propose schema hardening by removing extension points; however, all involved schemas need to be known beforehand. As an alternative, this paper proposes a grammatical inference approach for learning the accepted language. The contributions are dXVPAs and cXVPAs as streaming-capable language representations of mixed-content XML, algorithms for datatype inference from text, an incremental learner, and an experimental evaluation in two synthetic and two realistic scenarios.

The proposed approach has shown promising results with simple parameters ($k = 1$, $l = 2$) and exceeded baseline schema validation in experiments; nonetheless, there are limitations. Some attacks in experiments could not be identified because lexical spaces of XSD datatypes are too corse. Improvements by introducing more fine-grained string datatypes would raise the detection rate. Also, repetitions are not bounded, and Definition 2 assumes an order on unordered attributes. Repetition bounds and unordered attributes are two open research questions.

Furthermore, the k-l-local state merging approach restricts the learnable language class and returns only an approximation of more powerful languages. Improving the learning setting, e.g., query learning [53], is a major open research question. Finally, a dXVPA representation captures only syntax and no integrity constraints. Extending the learner to infer integrity constraints is another open research question; however, Arenas et al. [54] have already shown that this problem is computationally much harder.

Acknowledgments

The research has been supported by the Christian Doppler Society.

References

[1] W3C, "Extensible Markup Language (XML) 1.0 (Fifth Edition)," Nov. 2008, accessed 2014-02-17. [Online]. Available: http://www.w3.org/TR/2008/REC-xml-20081126/

[2] M. McIntosh and P. Austel, "XML signature element wrapping attacks and countermeasures," in *Proc. SWS'05*. ACM, 2005, pp. 20–27.

[3] S. Gajek, L. Liao, and J. Schwenk, "Breaking and fixing the inline approach," in *Proc. SWS'07*. ACM, 2007, pp. 37–43.

[4] S. Gajek, M. Jensen, L. Liao, and J. Schwenk, "Analysis of signature wrapping attacks and countermeasures," in *Proc. ICWS'09*. IEEE, 2009, pp. 575–582.

[5] J. Somorovsky, M. Heiderich, M. Jensen, J. Schwenk, N. Gruschka, and L. Lo Iacono, "All your clouds are belong to us: Security analysis of cloud management interfaces," in *Proc. CCSW'11*. ACM, 2011, pp. 3–14.

[6] J. Somorovsky, A. Mayer, J. Schwenk, M. Kampmann, and M. Jensen, "On breaking SAML: Be whoever you want to be," in *Proc. Security'12*. USENIX, 2012, pp. 21–21.

[7] M. Jensen, L. Liao, and J. Schwenk, "The curse of namespaces in the domain of XML signature," in *Proc. SWS'09*. ACM, 2009, pp. 29–36.

[8] M. Jensen, C. Meyer, J. Somorovsky, and J. Schwenk, "On the effectiveness of XML schema validation for countering XML signature wrapping attacks," in *Proc. IWSSC'11*. IEEE, Sep. 2011, pp. 7–13.

[9] H. Lampesberger, "A grammatical inference approach to language-based anomaly detection in XML," in *Proc. ECTCM'13*. IEEE, 2013, pp. 685–693.

[10] V. Kumar, P. Madhusudan, and M. Viswanathan, "Visibly pushdown automata for streaming XML," in *Proc. WWW'07*. ACM, 2007, pp. 1053–1062.

[11] W3C, "XML Information Set (Second Edition)," Feb. 2004, accessed 2014-08-02. [Online]. Available: http://www.w3.org/TR/xml-infoset/

[12] SAX Project, "Simple API for XML (SAX)," Apr. 2004, accessed 2014-06-05. [Online]. Available: http://www.saxproject.org/

[13] Java, "JSR 173: Streaming API for XML," Dec. 2013, accessed 2015-02-16. [Online]. Available: https://www.jcp.org/en/jsr/detail?id=173

[14] W3C, "XML Schema Part 0: Primer Second Edition," Oct. 2004, accessed 2014-07-14. [Online]. Available: http://www.w3.org/TR/xmlschema-0/

[15] OASIS, "RELAX NG," Dec. 2001, accessed 2014-07-14. [Online]. Available: https://www.oasis-open.org/committees/relax-ng/spec.html

[16] Y. Papakonstantinou and V. Vianu, "DTD inference for views of XML data," in *Proc. PODS'00*. ACM, 2000, pp. 35–46.

[17] W. Martens, F. Neven, T. Schwentick, and G. J. Bex, "Expressiveness and complexity of XML Schema," *ACM Trans. Database Syst.*, vol. 31, no. 3, pp. 770–813, 2006.

[18] H. Wu, T. Ling, G. Dobbie, Z. Bao, and L. Xu, "Reducing graph matching to tree matching for xml queries with id references," in *Proc. DEXA'10*, ser. LNCS. Springer Berlin Heidelberg, 2010, vol. 6262, pp. 391–406.

[19] J. Lindsey, "Subtyping in W3C XML Schema, part 1," *The Data Administration Newsletter*, Apr. 2008, accessed 2015-03-23. [Online]. Available: http://www.tdan.com/view-articles/7185

[20] D. Stephenson, "XML Schema best practices," HP, Tech. Rep., 2004, accessed 2015-03-25. [Online]. Available: http://xml.coverpages.org/HP-StephensonSchemaBestPractices.pdf

[21] 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.

[22] A. Falkenberg, M. Jensen, and J. Schwenk, "Welcome to ws-attacks.org," 2011, accessed 2015-02-05. [Online]. Available: http://www.ws-attacks.org

[23] T. D. Morgan and O. A. Ibrahim, "Xml schema, dtd, and entity attacks," Virtual Security Research, LLC, Tech. Rep., May 2014, accessed 2015-03-16. [Online]. Available: http://www.vsecurity.com/download/papers/XMLDTDEntityAttacks.pdf

[24] A. Vorobiev and J. Han, "Security attack ontology for web services," in *Proc. SKG'06*. IEEE, Nov. 2006, pp. 42–47.

[25] W3C, "XML Signature Syntax and Processing (Second Edition)," Jun. 2008, accessed 2015-01-23. [Online]. Available: http://www.w3.org/TR/xmlschema-0/

[26] N. Gruschka and L. Iacono, "Vulnerable cloud: SOAP message security validation revisited," in *Proc. ICWS'09*. IEEE, Jul. 2009, pp. 625–631.

[27] M. A. Rahaman, A. Schaad, and M. Rits, "Towards secure SOAP message exchange in a SOA," in *Proc. SWS'06*. ACM, 2006, pp. 77–84.

[28] C. Mainka, M. Jensen, L. L. Iacono, and J. Schwenk, "XSpRES - robust and effective XML signatures for web services," in *Proc. CLOSER'12*. SciTePress, 2012, pp. 187–197.

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

[30] H. Fernau, "Identification of function distinguishable languages," *Theor. Comput. Sci.*, vol. 290, no. 3, pp. 1679–1711, 2003.

[31] C. Mainka, J. Somorovsky, and J. Schwenk, "Penetration testing tool for web services security," in *Proc. SERVICES'12*. IEEE, Jun. 2012, pp. 163–170.

[32] R. Alur and P. Madhusudan, "Visibly pushdown languages," in *Proc. STOC'04*. ACM, 2004, pp. 202–211.

[33] W3C, "W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes," Apr. 2012, accessed 2015-04-02. [Online]. Available: http://www.w3.org/TR/xmlschema11-2/

[34] G. J. Bex, F. Neven, and J. Van den Bussche, "DTDs versus XML Schema: A practical study," in *Proc. WebDB'04*. ACM, 2004, pp. 79–84.

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

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

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

[38] D. Angluin, "Inductive inference of formal languages from positive data," *Inform. Control*, vol. 45, no. 2, pp. 117–135, 1980.

[39] K. P. Jantke, "Monotonic and non-monotonic inductive inference," *New Generat. Comput.*, vol. 8, no. 4, pp. 349–360, 1991.

[40] G. F. Cretu, A. Stavrou, M. E. Locasto, S. J. Stolfo, and A. D. Keromytis, "Casting out demons: Sanitizing training data for anomaly sensors," in *Proc. S&P'08*. IEEE, 2008, pp. 81–95.

[41] A. Møller, "dk.brics.automaton – finite-state automata and regular expressions for Java," 2010, accessed 2015-08-07. [Online]. Available: http://www.brics.dk/automaton/

[42] J. Davis and M. Goadrich, "The relationship between Precision-Recall and ROC curves," in *Proc. ICML'06*. ACM, 2006, pp. 233–240.

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

[44] D. Barbosa, A. Mendelzon, J. Keenleyside, and K. Lyons, "ToXgene: A template-based data generator for XML," in *Proc. SIGMOD'02*. ACM, 2002, pp. 616–616.

[45] L. Segoufin and V. Vianu, "Validating streaming XML documents," in *Proc. PODS'02*. ACM, 2002, pp. 53–64.

[46] B. Chidlovskii, "Schema extraction from XML data," Xerox, Tech. Rep. 2001/200, 2001, accessed 2015-06-20. [Online]. Available: http://www.xrce.xerox.com/Research-Development/Publications/2001-200

[47] I. Mlýnková and M. Nečaský, "Heuristic methods for inference of XML schemas: Lessons learned and open issues," *Informatica*, vol. 24, no. 4, pp. 577–602, 2013.

[48] G. J. Bex, F. Neven, and S. Vansummeren, "Inferring XML schema definitions from XML data," in *Proc. VLDB'07*. VLDB Endowment, 2007, pp. 998–1009.

[49] J. Hegewald, F. Naumann, and M. Weis, "XStruct: Efficient schema extraction from multiple and large XML documents," in *Proc. ICDEW'06*. IEEE, 2006, pp. 81–81.

[50] E. Menahem, A. Schclar, L. Rokach, and Y. Elovici, "XML-AD: Detecting anomalous patterns in XML documents," *Inform. Sciences*, 2015.

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

[52] K. Rieck, T. Krüger, U. Brefeld, and K.-R. Müller, "Approximate tree kernels," *Journal Mach. Learn. Res.*, vol. 11, pp. 555–580, 2010.

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

[54] M. Arenas, J. Daenen, F. Neven, M. Ugarte, J. V. D. Bussche, and S. Vansummeren, "Discovering XSD keys from XML data," *ACM Trans. Database Syst.*, vol. 39, no. 4, pp. 28:1–28:49, Dec. 2014.