Flexible authentication of XML documents

Prem Devanbu*, Michael Gertz, April Kwong, Chip Martel, Glen Nuckolls
Department of Computer Science
University of California
Davis, California CA 95616 USA
{nuckolls|martel|gertz|devanbu|kwonga}@cs.ucdavis.edu

Stuart G. Stubblebine
CertCo
55 Broad Street – Suite 22
New York, NY 10004
stubblebine@cs.columbia.edu

Abstract

XML is increasingly becoming the format of choice for information exchange, in such critical areas such as government, finance, healthcare and law, where integrity is of the essence. As this trend grows, one can expect that documents (or collections thereof) may get quite large, and clients may wish to query for specific elements of these documents. In critical applications, clients must be assured that they are getting complete and correct answers to their queries. Existing approaches to signing XML documents don't support the authentication of answers to queries. Certainly, a server could process queries and certify answers by digitally signing them with an on-line private key; however, the server, and its on-line private key, would be vulnerable to external hacking and insider attacks. We propose a new approach to signing XML documents which allows *untrusted* servers to answer certain types of path queries and selection queries over XML documents without the need for trusted on-line signing keys. This approach enhances both the security and scalability of publishing information in XML format over the internet. In addition, it provides greater flexibility in authenticating parts of XML documents, in response to commercial or security policy considerations.

1 Introduction

XML is increasingly becoming the format of choice for publication of information over the internet, in such critical areas such as government, finance, healthcare and law, where integrity is of the essence. As the volume of information available in XML format grows, one can expect that clients may wish to query for specific elements of interest. In critical applications, clients must be assured that they are getting complete and correct answers to their queries. Thus, for example, an employer seeking to hire a driver might wish to query for all available information on traffic violations in all precincts with the same social security number. A complete and correct listing of all violations would be critical. Another example is servicing requests submitted under the Electronic Freedom of Information Act Amendments (E-FOIA), which requires the US Government agencies to provide an online index, and search for records by electronic means[1]. Other democracies have similar procedures that allow ad-hoc oversight of governmental activities by concerned citizens. Traditional, paper-based processes that have been used in the past to satisfy FOIA requests are haunted by the specter of "plausible deniability". Some may fear that governments might contrive to hide or destroy records, other than through lawfully and procedurally sound means.

The focus of this paper is the following question: When an untrusted party returns a part of an XML document claiming it be the complete and correct answer to a query, how can this claim be verified?

Existing approaches to signing XML documents [XMLDSIG] don't support the authentication of answers to queries. Certainly, a server could process queries and certify answers by digitally signing each answer with an on-line private key; however, the server, and its on-line private key, would be

*We gratefully acknowledge support from the NSF ITR Program, Grant No. 0085961

[1]See 44 U.S.C 3506(b)(4) and 5 U.S.C 552(a)(2) and (3)(C)

vulnerable to external hacking and insider attacks. At any rate, if the server itself is untrusted, then having it digitally sign answers serves no purpose. We propose a new approach to signing XML documents which allows *untrusted* servers to answer certain types of path queries and selection queries over XML documents without the need for trusted on-line signing keys. This approach enhances both the security and scalability of publishing information in XML format over the internet. In addition, it provides greater flexibility in authenticating parts of XML documents, in response to commercial or security policy considerations. Our approach currently applies only to non-recursive DTD's; however in a random survey of 100 publicly available DTDs, we found that 64 were non-recursive. So we believe our approach will be useful in many applications.

In Section 2, we present background information on related technologies assumed in this paper. In Section 2.2, we present the related work. In Section 3, we present our basic approach for authenticating answers to path queries. We conclude in Section 4.

2 Background

Suppose that a client C desires to process queries over a large XML document D held by a trusted server S. The traditional model can be described thus:

1. $C(Pk) \xrightarrow{Q} S(D, Sk)$ (Client C, with a Public key Pk, asks server S, who has document D and private key Sk for the answer to executing query Q over document D)
2. $C(Pk) \overset{eval(Q.D)}{\underset{\sigma_{Sk}(eval(Q,D))}{\longleftarrow}} S(D, Sk)$ (Server returns answer and verifable digital signature to client).

In the above scenario, there is no way to pre-compute all possible signatures to all possible answers; so the server needs to be trusted, and needs to securely maintain an on-line signing key. For the reasons discussed above, we would like to have the queries processed by untrusted servers, without the need for on-line signing keys. We use a scenario where an owner O computes a digest and signs it once, and thenceforth all the query processing is done by an untrusted server U, who always provides a certificate of correctness with every answer. This simplifies the key management burden; e.g., the owner can relegate the signing key to a smart card that he keeps locked up in a safe between updates.

1. $C_1 \ldots C_n(Pk) \overset{sd(D),\sigma_{Sk}(sd(D))}{\longleftarrow} O(D, Sk)$ (All clients recieve from the data owner O a signed, specially computed digest of the data using a non-standard digesting algorithm sd, which relies on a keyless one-way hash function h).

2. $C_i(Pk, sd(D)) \xrightarrow{Q} U(D)$ (Client C, submits request to untrusted server U, who has the data, but *no private signing keys!* Client, however, has the verified digest she got from O).

3. $C_i(Pk, sd(D)) \overset{eval(Q,D)}{\underset{cert(eval(Q,D))}{\longleftarrow}} U(D)$ (Client C, receives back from U an answer, and a specially computed *certificate* that lets her check that the answer is correct (next step). The certificate uses only the keyless one-way hash function h).

4. $C_i(Pk, sd(D)) : verify(cert(eval(Q, D)), sd(D))$ (Client C runs a special *verification* procedure that compares the digest received from O against the results of a particular hash calculation over the certificate received from U. If the comparison succeeds, she accepts; otherwise she rejects).

Various researchers have developed this approach [NN99, DGM00a, GTS01, BLL00]; we review these in more detail in a subsequent section. We seek to adapt this approach to XML documents. Various advantages have been reported in earlier literature: scalability, flexibility, security etc. We would like to bring these advantages to the increasingly popular XML data model.

In this section, we present some background material. First, we introduce a simple data-model that captures the essence of XML documents relevant to our purposes, and introduce DTDs, path queries and selection queries. Then, we review the relevant background and related work in the area of certified data publishing, and finally present the DOMHASH standard for securely hashing XML documents, as well as the XML digital signature standard.

2.1 XML Document Model, DTDs, and Path Queries

We employ an XML document model in which a document is represented as an ordered, node-labeled tree. This conventional terminology for XML documents is also widely used in W3C proposals such as, e.g., XML Information Set [Cow01] or XPath [CD99].

Assume a set Σ of element names and a set D of string values disjoint from Σ. In the XML document model, an XML document X is a 4-tuple $(V, r, label, elem)$ where

- V is a set of vertices with r being a distinguished element in V, called the root node,
- $label$ is a mapping from vertices to element labels, i.e., a function $V \rightarrow \Sigma$, and
- $elem$ is a mapping from vertices to their children, i.e., a function $V \rightarrow List(V \cup D)$.

For the sake of simplicity and to motivate the basic concepts of our approach, we do not consider entities, comments, processing instructions etc. that can occur in an XML document. In particular, we do not consider element attributes since they can easily be included in our approach (as another type of node).

A node $v \in V$ is called a *text-node* if $elem(v) \in D$. Only leaf-nodes in a document tree can be text nodes. Each node v different from r (the root node) has a parent node, denoted $parent(v)$. For each node $v \in V$, there is a unique *node path* in X, which consists of a sequence of nodes, starting with the root node r and ending with the node v. Associating a label with each node in a node path results in a so-called *ancestor path*, denoted $path(v)$, which is a sequence of element names. Thus different node paths can have the same ancestor path.

In order to allow for meaningful exchange of XML documents and to formally describe admissible structures of XML documents, a *document type definition* (DTD) can be associated with a collection of XML documents. A DTD includes declarations for elements, attributes, notations, and entities. Most importantly, element declarations in a DTD specify the names of XML elements and their content (aka *content model*). A DTD consists of `ELEMENT` rules that present an element, and (using an extended BNF grammar), a description of the elements that can occur below it, with their cardinality. An XML document is said to be *valid* if it conforms to a given DTD. Figure 1 shows an example of an XML document in its linear form as well as an ordered, node-labeled tree. Figure 2 shows a DTD the document conforms to. In this DTD, the element `beneficiary` has its subelements, `name`, `ssno`, and `address`. The element `will` is not the subelement of any other element, and is referred to below as the *root element*.

It should be noted that though DTDs are just one schema formalism for XML documents, it is the most popular one and is also widely used in practice. Other schema formalisms, e.g., XML schema,

```
<will>
 <principal><name> Pete Princ </name></principal>
 <preparer>
    <name> Nolo Willmaker </name>
 </preparer>
 <witness> <name> Bob Witness </name></witness>
 <witness> <name> Barb Witness </name></witness>
 <filing> <town> Davis </town> <county> Yolo </county><state> CA </state></filing>}
 <bequeath>
    <item> W. Earth </item>
    <beneficiary><name> T. Meek </name><ssno> 111-222-3333</ssno>
       <address> 1 Main Street, anytown, CA 11111 </address>
    </beneficiary>
 </bequeath>
</will>
```

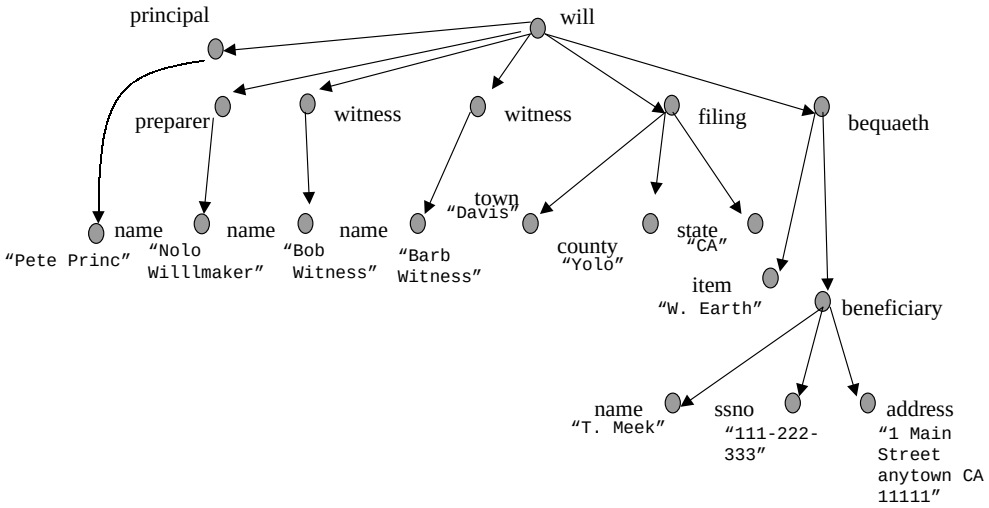

Figure 1: XML Document X in Linear and Tree Form

have been proposed and studied in the literature but have not yet reached the same level of usage as DTDs (see [LC00] for a comparison of different schema proposals).

Besides providing a formal description of valid XML documents, a DTD also serves as a schema for querying XML documents. In the past few years, several XML query languages have been proposed, including XML-QL, Quilt, and XQL (see [BC00] for an overview). Although the languages differ in terms of expressiveness, underlying formalism and data model, there is an important feature common to all languages, namely *path queries* [AV97].

The primary purpose of a path query is to address subtree structures of an XML document using regular expressions over XML element names. Path expressions also build the foundation of XPath [CD99], which, in turn, builds the basis of the widely used XSL Transformation [Cla99]. Instead of focusing on a specific XML query language, we base our XML document authentication framework on path queries a client issues against a document maintained by a publisher. This framework is sufficiently general to tailor it to more specific types of applications, e.g., XSLT.

Definition 2.1 *(Path Query) Let Σ be a set of element names. Path queries over Σ are regular expressions. The general syntax of a regular expression is*

$$q := \varepsilon \mid e \mid q.q \mid q* \mid q+ \mid q? \mid q|q \mid \_$$

where e ranges over Σ, q over expressions, and ε is the empty expression. The expressions $q.q$ and $q|q$ stand for the concatenation and alternative expressions, respectively. $q*$ (Kleene Star) stands for 0 or more repeats of q and $q+$ stands for at least one repeat of q. $q?$ denotes zero or one occurrence of q. The wildcard "_" stands for any element of Σ.

```
<!ELEMENT will (principal preparer witness* filing bequeath*)>
<!ELEMENT principal name>
<!ELEMENT preparer name>
<!ELEMENT witness name>
<!ELEMENT filing (town county state)>
<!ELEMENT bequeath (item beneficiary)>
<!ELEMENT town (\#PCDATA)
<!ELEMENT ssno (\#PCDATA)
<!ELEMENT item (\#PCDATA)
<!ELEMENT county (\#PCDATA)
<!ELEMENT state (\#PCDATA)
<!ELEMENT address (\#PCDATA)
<!ELEMENT beneficiary (name ssno address)>
<!ELEMENT name (\#PCDATA)>
<!ELEMENT address (\#PCDATA)>
```

Figure 2: DTD associated with XML document X in Figure 1

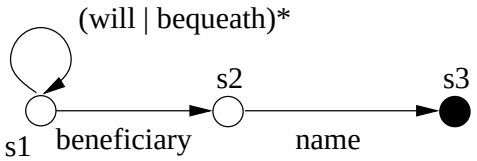

Figure 3: Path Automaton for a given Path Query

In the following, we do not explicitly consider expressions of the type $q+$ since they can be expressed as $q.q*$. We also assume that the wildcard "_" is only used in combination with Kleene Star as "_*", which is equivalent to $*$, meaning any number of elements from Σ. It should be noted that concatenation "." and Kleene Star "*" correspond to the widely used operators / and //, respectively, in XPath.

Intuitively, given a document X, a path query p determines a (possibly empty) set of subtree structures in X such the ancestor path of each root node of such a subtree matches the expression p. For example, based on the XML document shown in Figure 1, the path query $*.(\texttt{witness} \mid \texttt{bequeath}).*.\texttt{name}$ selects all subtree structures from X where the root of the subtree has the label `name` and can be reached from the root node of the document through a label path that matches the expression $*.(\texttt{witness} \mid \texttt{bequeath}).*$. For the above document, there are three such subtree structures.

An interesting aspect and also important to our approach is that for a given path query, one can construct an equivalent *finite path automaton* using the well-known transformation from regular expressions to finite automata. For example, the path query $(\texttt{will} \mid \texttt{bequeath}).*.\texttt{beneficiary.name}$ can be represented by the finite automaton shown in Figure 3.

We can define a path automaton $\mathcal{PA}$ as a tuple $\langle \Sigma, Q_p, \alpha, F_p \rangle$, where Σ is set of element names (tags), Q_p is the set of states, $F_p \subseteq Q_p$ is the set of accepting states, and α is the transition function $\alpha : \Sigma \times Q_p \rightarrow Q_p$.

Now consider the DTD shown in Figure 2. We can see that the tag `name` occurs in several different places: under `principal`, `preparer`, `witness` and under `beneficiary`. Based on the DTD, it is evident that the answer to the query shown in Figure 3 is the subtree that occurs under the tag `name` which can be reached from the root labeled `will` through the nodes labeled `bequeath` and `beneficiary`, (in that order). On the other hand, consider the path query (`will.witness.name`). Since there can be many `witness` tags, the above path query can potentially retrieve a list of subtrees. These two examples illustrate a central fact: Given a path query and a document that conforms to a DTD, we can make use of the DTD to constrain, *a priori* in which part of the document tree the

answers to the path query will arise. Our goal is to efficiently retrieve and authenticate answers to path queries. To do this, we construct a special data structure, called the *xtrie*, that condenses the information in the DTD in a manner that helps process path queries and which is discussed in detail in Section 3.

Path queries specify document structures based on ancestor paths. They do not, however, specify conditions on string values associated with text-nodes. For example, several subtree structures of a document may match a path query but an application might be interested in only those structures that furthermore satisfy certain condition on the values of text-nodes. These two aspects resemble the same functionality as projection and selection provided in, e.g., relational algebra. In order to provide more expressive types of queries against documents in our XML data authentication framework, we introduce the concept of *selection queries*, which naturally extend path queries. Obviously, a selection can only be applied to document leaf nodes where $elem(v) = s, s \in \Sigma$. A selection query against a document X is composed of two parts: (1) a path query that determines subtrees in X, and (2) a selection part that specifies a condition on the leaf nodes in the selected subtrees.

Definition 2.2 *(Selection Query) Given a path query q and a comparison $p \equiv e\Theta c$ where $\Theta \in \{=, <>, <, >, \leq, \geq\}$, $c \in D$ is a string constant, and $e \in \Sigma$. A* selection query*, denoted $select(D, q, s, p)$, determines all subtree structures $T_1, \ldots, T_n$ in a document X such that for each T_i*

1. *T_i is a correct answer to the path query q, and*
2. *there exists a leaf node, reached by path s starting at the root of T_i such that the string value at that leaf satisfies the predicate Θ regarding c.*

Note that we only require that a text-node satisfies the predicate p at the path selected by s under the answer subtrees. Also, since we do not assume some kind of typed schema underlying XML documents (e.g., XML Schema [Fal01]), all values associated with leaf nodes are assumed to be strings. Consider, for example, the selection query $select(\texttt{will.witness}, \texttt{name}, \texttt{name} = \text{"}\texttt{BarbWitness}\text{"})$ on the XML document shown in Figure 1. There are two subtree structures rooted with a node labeled `witness`. Among these two, only the second one is selected as the result to the query since only this subtree has a leaf-node with the text `Barb Witness`.

2.2 Certified Query Processing

Our work on certifying answers to queries over XML documents follows several other efforts aimed at producing certified answers to queries in other contexts. Most of these efforts are based on Merkle hash tree constructions, as illustrated in figure 4. Such hash trees enable certified query processing over some types of recursive data-structures. A trusted party computes a systematic hash digest of a data-structure, progressively digesting it from the leaves to the root, using a secure hash function. The trusted party then signs the root hash, which is distributed to clients. In response to a query from a client, an *untrusted* party can traverse the data-structure, and provide an answer. The certificate accompanying this answer would consist of the part of the tree traversed during the search, and enough other hash values so that the root hash value can be recomputed and checked by the client.

To our knowledge, this approach was first used in [NN99] for proving the presence or absence of certificates on revocation lists. For example, in figure 4 suppose the root hash value has been already obtained by a client. Now, an untrusted party can show that the value 23 occurs at a leaf of the tree, by providing the values *h(34)*, *h(h(312) || h(1123))* and the value *hl*. With these values, the client recompute the root hash, and thus be sure (subject to the security of the hash function) that the value 23 did occur in the list. Likewise, a pair of consecutive values, 312, and 1123, along with the necessary intervening hash values to compute the root hash, can establish that they were indeed consecutive

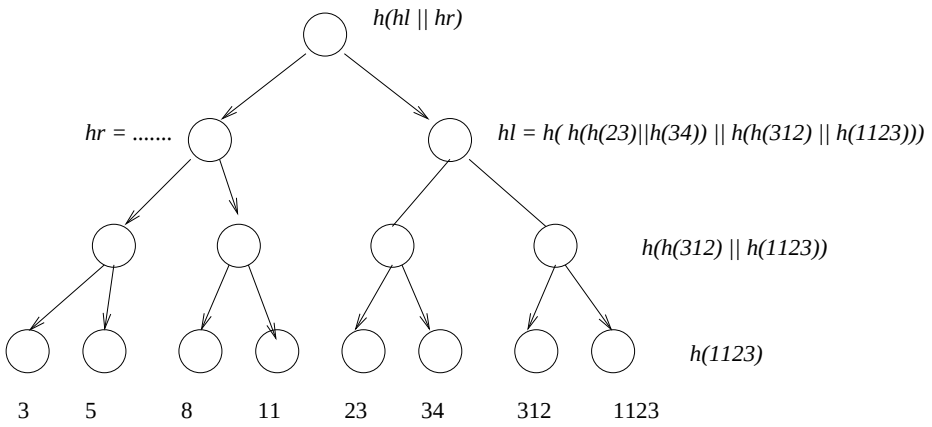

Figure 4: A Merkle hash tree associates hash values with nodes in a tree, in this case a binary tree. The leafs are sorted values in the binary tree. The leaves get the hash of the values, and each interior node gets the hash of the appended values of all its children. The root digest is signed by a trusted party. Now any untrusted party can certify answers to queries, using only the hash values to provide evidence of a correctly conducted search

values, and thus that (e.g.) the value 650 does *not* occur in the tree. Range queries, which ask for all values in a range (e.g., between 30 and 400) can also be handled. This line of work has been extended to some limited forms of querying over relational databases [DGM00a], used over skip lists [GTS01] and in other, related settings [BLL00]. When used with hierarchical "divide-and-conquer" type search datastructures, this approach provides answer certificates whose size is on the order of $\mid A \mid log \mid D \mid$ where $\mid A \mid$ is the size of the answer and $\mid D \mid$ is the size of the data set used in constructing the search datastructure. In the XML context (details in Section 3.2), to answer selection queries over leaf values, we sort the leaf values and build an index datastructure over these values. This index structure gets Merkle-hashed. Once a client has a secure way of obtaining the root hash of this structure, it is possible for untrusted publishers to provide credible evidence of complete answers to selection queries.

Although these techniques are useful in the XML context (particularly for doing selections, we make use of divide-and-conquer index structures), the much less structured nature of the XML data model requires some additional machinery to certify answers to queries.

2.3 Hashing and Signing XML Documents—Standards

As described earlier, XML documents have a simple data model based on trees. DOM [DH00] is a standard interface (API) that defines how XML documents are to be accessed by programs. DOM is naturally a tree-like representation; as such, it admits a hashing procedure very similar to the Merkle-hashing procedure describe above, and illustrated in figure 4. One-way hash functions are used for security. The procedure basically hashes the leaves of the document, and recursively proceeds up the document tree, hashing both the element types as well as the order of occurrence of the elements within the document. The full details of DOMHASH are available elsewhere [DH00]. For our purposes, we note some important properties of the DOMHASH process.

If the root hash of an entire document D is known to a party a, it is possible to provide evidence to a that any subtree τ of the document occurs under D without revealing all of D. First, note that a can DOMHASH the subtree τ to get the root hash of τ. Now, a can be given just the hash values of the siblings of τ and the siblings of all it's parents, and a can recompute the root hash of D. Since

the hash function is assumed to be one-way, a can be reasonably sure the hash values could not have been forged, and that τ really did occur under D. The same process can be used to prove that one subtree τ_1 occurred under another subtree τ_2 within the same document, by just using the hash values along the path from τ_1 to τ_2, without revealing any of the other subtrees of τ_2. Furthermore, since DOMHASH includes information about the order of occurrences of elements within a document D, it can be used to provide evidence about the relative order of occurrence of subtrees.

The XML Digital signature standard [XMLDSIG] essentially computes a signature over digesting procedure such as DOMHASH. The standard allows a great deal of flexibility in the digesting and signature process, in terms of algorithms used, transformations applied, parts selected etc. However, it has a central limitation: only fixed parts of a document can be signed. It is not possible to certify answers to selection and path queries based on a single signature over the entire document. For our purposes, we assume an XML document digital signature based on DOMHASH.

3 Certifying answers to path queries

The central goal in this paper is to allow certification of answers to a wide range of queries over XML documents, without requiring a trusted party to sign the answer to each query. We would like to certify an XML document in one shot, with a digital signature, and literally lock up the secret key in a drawer. Now, we want to certify answers to a wide range of queries over that document, without the need for any additional digital signatures. Unfortunately, there are no *a priori* limits on size and variety of XML documents, even on those that conform to a given DTD. Given this, how can this be done? We exploit a key property of non-recursive DTD's: although the documents that conform to such DTDs can be infinitely varied, and arbitrarily long, *there are only a finite number of semantically different path queries that apply to such documents.*

We describe our approach in several steps. First, we argue that there are only finitely many different path queries over any document conforming to a non-recursive DTD. Second, we show a naive approach that uses this property, to sign a document only once and use this signature to certify answers to all the possible different queries over this document, and show that this naive approach is secure subject to cryptographic assumptions. Third, we describe an improvement to the naive approach using the structure of the DTD itself to build an optimized data structure, an *xtrie*, to store the different possible answers to path queries. We illustrate the improvements provided by this datastructure using some empirical data. Finally, we describe how to use the xtrie, in conjunction with auxiliary data structures, to perform selection queries that use paths.

Our work currently focuses on non-recursive DTDs. In a survey of 100 different DTDs found on a public XML DTD site[2] we identified 64 non-recursive DTD's. We present the non-recursive DTDs in Appendix 1: it can be seen that this class captures a wide range of applications from different domains.

3.1 Step 1: Path Queries in non-recursive DTD's

Given a non-recursive DTD, we argue that there are only finitely many different meanings for path queries (irrespective of infinitely many syntactically different path queries). If this seems surprising, we remind the reader that path queries are analogous to projections in relational databases; there, once the schema is fixed, there are only finitely many different projections. Likewise, given a DTD, there are only finitely many different ways to carve it up with path queries. We argue this below:

[2]Please see http://www.oasis-open.org.

Definition 3.1 *Two path queries q1 and q2 are semantically distinct with respect to a DTD if there can be constructed a document D that conforms to that DTD, and for which q1 and q2 give different answers.*

Lemma 3.2 *Given a non-recursive DTD, there are only finite number of semantically distinct path queries.*

Proof Sketch: Without loss of generality, consider an arbitrary XML document D conformant to a given non-recursive DTD *dtd*. Clearly the maximum depth of D conforming to *dtd* is bounded. Now given any node n in D, the number of *different* possible tags that can occur as children of that node is bounded. Certainly a node may have any number of children (i.e., if the corresponding `ELEMENT`) statement includes a Kleene closure; however, the number of different tags that can occur below any node is given by the (finite) number of different tags that appear in the corresponding `ELEMENT` statement. Since the depth is bounded, and the "fan-out" of each different type of node is bounded, there are only a finite number of different *paths* that can occur in documents conforming to *dtd*. Call this number N_{dtd}. Thus an arbitrary path query would have to select from the finite number of subsets of these finite number of paths. Thus the maximum number of semantically different path queries over documents conforming to a non-recursive DTD is $2^{N_{dtd}}$. □

Thus all possible path queries fall into one of a finite number of groups of equivalence classes. This number ($2^{N_{dtd}}$) will be quite large for non-trivial DTD's. As can be seen from table in figure 6, there often can be thousands of different paths in documents conforming to practical DTDs. Considering a table with on the order of 2^{1000} entries, each representing an equivalence class, we can quickly see that it would not be feasible to store answers to each equivalence class in a table and simply retrieve it. We present a better approach first, in the following subsection, and then improve it even further.

3.2 Step 2: A naive approach to Flexible certification

We now consider a naive approach to storing potential answers to path queries in a table, and certifying this table. We also show that this approach is secure, subject to the use of a one-way hash function. There are three algorithms that constitute our approach: the first that signs the data, the second that processes queries, and builds *certifiers* that establish that the answers are correct, and third, an algorithm that checks an answer and its certifier to ensure that the answer is correct. The certifier is a cryptographic object that uses no digital signatures, only hash computations; the second algorithm could thus be executed by an untrusted adversary.

The details of the algorithms are shown below. These algorithms rely upon several facts; first, that there are only a finite number of different paths in a non-recursive DTD, second, that an XML document, and subtrees there-of, can be securely hashed using DOMHASHes, which cannot be forged; and third, that groups of hashes can be securely and verifiably linked into a single digital signature using Merkle hash trees. The central security result here (Theorem 3.3) is that the client will always reject an incorrect answer and certifier unless the adversarial party that built the certifier managed to break the one-way hash function.

We begin with Algorithm 1, which is executed by the owner. It is given a DTD, a document conforming to the DTD, and a table τ with a finite number of (empty) entries, one for each possible path in the DTD. It first processes the document, associating each subtree the path from the subtree to the root. It also builds a DOMHASH, associating a secure hash value with each subtree. It then associates the subtree and the hash thereof with the entry in the path table, based on reaching path. There could be many subtrees associated with each entry in the path table. These are digested together, using a Merkle hash-tree, to give a digest per table entry. Finally the set of all the table entries are digested

together using another Markle hash tree. This final digest is the specially computed digest $sd(D)$ mentioned in Section 2. We note that the Merkle hash construction allows us to show that an entry occurred in a path table, and that a subtree occurred in a path table entry.

Algorithm 1 (**Data certification**):
Inputs: a non-recursive DTD, a table τ of all paths associated with the DTD, an XML document, D and a signing key k^{-1}

1. Process the document D, associating each sub-tree of D with an element label from the DTD.

2. Build DOMHASH of the document.

3. For each subtree of the document, find the associated entry in the path table, and enter the subtree identifier, and the DOMHASH for that subtree. The entries should be in the order of occurrence in the document.

4. The entries in a table entry are digested together using a Merkle Hash tree; the root hash of this tree is associated with the table entry.

5. All the entries in the table are digested together using another Merkle Hash tree.

6. The root hash of this tree is signed with the signing key, producing a root signature $sd(D)$.

Once the owner has produced the digest as specified above, and clients have received and checked the digest, the untrusted server can process queries from clients, using Algorithm 2. First, the path query is matched against every entry in the path table (this is slow, but we improve this later). The matching entries are the paths that match the path query, and they contain all the subtrees reached by those paths. The server can return just the subtrees in those path table entries. Using these the client can compute the DOMHASH of each subtree and then can recompute the digests for those path table entries. If also given the hash paths required to verify that the digests really belong under the Merkle hash tree leading to the overall document digest computed in Algorithm 1, the client can trust the returned values.

Algorithm 2 (**Answer certification**):
*Inputs:*The table τ with entries produced by algorithm 1, a path query Q from a client.

1. Match Q against each entry in τ.

2. If Q matches the entry, retrieve (1) the path, (2) the hash path from that entry to the root signature $\sigma(\delta)$, and (3) all the subtrees that are associated with that table entry. Build this certifier pair for each matching entry in τ

3. Return this list of triples, containing both the answer to the query and the certifier, to the client

Algorithm 3 simply verifies the certificate (the list of triples) produced above by recomputing the document digest.

Algorithm 3 (**Answer verification**):
Inputs: The list of certifier triples above, the table τ, and the query Q.

1. Match Q against each entry in τ.

2. For each matching entry, there should be a corresponding certifier triple. If there is no corresponding certifier triple, reject.

3. In each triple, first DOMHASH each returned subtree

4. Build a Merkle tree out of the DOMHASHes and compute the root hash.

5. Use the hash path provided, beginning with the entry digest, to the root table digest, and check that the root digest matches. If not reject. Otherwise, accept.

Theorem 3.3 *Assume that (given a DTD and a conforming XML document D) Algorithm 1 is executed correctly by data signer, and the sd(D) received intact by the client. Assume that given a query Q from a client, that a set of certifier triples are claimed to have been constructed (possibly by an untrusted party) as described in Algorithm 2. Now a client, who has received sd(D), and runs Algorithm 3 will always reject an incorrect answer, and accept a correct one, unless the party running Algorithm 2 has succeeded in engineering a collision in the hash function used in Algorithm 1.*

Proof Sketch: We assume here that the client correctly computes a set of table entries, using the algorithm, and asks the publisher for the set of subtrees in each of those table entries. The argument that the client will accept a correct answer is straightforward, based on the fact that he simply repeats the computation done by the Algorithm 1 & 2 and comes up with the same resulting $sd(D)$ value. We now argue that the client will reject incorrect answers and certificates. Its sufficient to establish that the client will not accept a wrong set of subtrees from the publisher for any table entry. If an adversarial publisher returns an incorrect subtree, then the corresponding DOMHASH will be different from that used in the process of computing the digest for that table entry. So the adversary has to have found a second pre-image that hashes to the same value in some step in the process of computing the digest for entry; alternately the adversary has to have found a hash collision in some step of the process of computing the digest for the entire table. In either case, the publisher has to engineer collisions in the hash functions that produce a specific output. □

3.3 Step 3: Efficient path storage using an xtrie

In a DTD, we find that several different element tags can occur under one single element. For example, under `beneficiary`, in our running example, we have `name`, `ssno` and `address`. This leads to a lot of shared prefixes among paths; thus, all of the above elements occur under the shared path prefix `will.bequeath`. The naive table that stores all paths wastes space; in addition, matching a path query automaton against each entry in the table wastes repeated effort matching identical, repeated prefixes. Below, we present a more compact and efficient datastructure, an xtrie, for storing the set of possible paths in a DTD.

3.3.1 Constructing the xtrie

We first present Algorithm 3.4, which builds an xtrie from a given DTD. We later show that it correctly captures all the paths in a non-recursive DTD, and provide empirical evidence of its compactness. The xtrie has at least one node for each element in the DTD, and has an edge from an element to all the subelements that can occur below it. The edge is labeled with the "tag" of the subelement. Thus, in our running example, there would be a node corresponding to the `witness` element, and an edge named `name` from it to a node corresponding to the `name` element. Different occurrences of an element are represented by different nodes. Algorithm 3.4 begins at the root element (which cannot occur under any other elements) in a DTD, and systematically explores the `ELEMENT` rules, finding all the possible paths that can exist. Intuitively, it "grows" a tree representation of all the different paths, producing a branch in the tree everytime a path prefix is shared among several different paths. Since DTDs allow the same element to occur under several distinct elements, there may be many paths that can reach a given element. This is handled by using an array of counters c_i, one for each element, which track the number of times an element has been encountered; each distinct encounter bumps the counter, and creates a new copy of the "path tree" associated with that element[3].

[3]It is possible to further optimize the xtrie to some extent, to eliminate this copying, but we omit this step here for simplicity.

Algorithm 3.4 *An* xtrie *corresponding to a DTD is a directed tree* $\langle N_{ta}, E_{ta} \rangle$*, with labeled nodes and edges. The construction of this graph is described below:*

Initialize *1. Let* Σ *be the complete set of distinct element tags that occur in the given DTD. Initialize a table of counters,* $c_t \leftarrow 0$*, for each* $t \in \Sigma$*.*

2. For each root element, with element label s *in the DTD, add a node* n_s*, labeled* $(s, 0)$*, to* N_{ta}*, and bump the associated counter,* $c_s \leftarrow c_s + 1$*.*

3. For each root element tag $f \in \Sigma$*, we add a specially labeled* top-level *node* $\hat{f} \in N_{ta}$*, labeled* $(\hat{f}, 0)$*; we also add an edge* $(\hat{f}, n_f)$*, from* $\hat{f}$ *to the node* $n_f \in N_{ta}$*. This edge is labeled with the tag* f*.*

4. Initialize an agenda list A *to the list of labels of root nodes in* N_{ta} *(not the top-levell nodes).*

Iterate *While labels remain in* A*, remove a label, say* (a, i)*, associated with a node* n*. For each* `ELEMENT` *rule associated with the tag* a*, if the BNF expression in this rule mentions a tag* t*, then we add a new node* n' *labeled* (t, c_t) *to* N_{ta}*; we add this new label to the agenda* A *to be processed later; we increment* $c_t \leftarrow c_t + 1$*; and add an edge* (n, n')*, labeled with the tag* t *to* E_{ta}*.*

Terminate *If* A *is empty, terminate.*

The initialization steps initialize the set of counters, add one node for each element in the DTD to the xtrie, and extra "top-level" nodes for each root element; we also add an edge from the top-level nodes to the root element's nodes. The counters associated with each element get initialized at this point, and the agenda is initialized with the list of root element nodes. At this point, the maximum path length (l) encountered is 1. As each step of the ensuing iteration the xtrie represents all paths of length $1 \ldots l$, from the "top-level" node to elements whose labels are in A. The iteration step now looks at elements in the agenda, and grows the paths by 1, and adds the nodes newly reached to the node set N_{ta} of the xtrie, and adds the appropriately labeled edges. It's easy to see that the algorithm terminates for non-recursive DTD's: there can only be a finite number of elements ever added to the Agenda. Elements (rather, element labels) get added to the agenda when they occur under another element. Each element can occur only a finite number of times in under different other elements in a DTD. Since there are only a finite number of copies of a finite number of elements that can ever be added to the Agenda, and each time through the agenda we remove an element, it will eventually empty out.

A sample xtrie is shown in figure 5. We note that redundancy is avoided in storing the three paths beginning with the common prefix `will filling`. We also note that the element `name` can occur in different places, under `witness`, `preparer`, etc, and so there are several copies of `name`, reachable through the different possible "super" elements. Xtries, as we have described them, resemble in some ways *schema graphs* [ABS200] that have been used in semi-structured databases, primarily for type computations; however, we use them here for certified query processing. We return to the central property of an xtrie: it captures exactly all the possible paths through a DTD.

Definition 3.5 *The sequence of labels* $f \ldots s$ *on the edges leading to a node* n_s *from a top-level node is called the reaching path for the node* n_s

We can now state the main "completeness and correctness" property of the xtrie:

Lemma 3.6 *Corresponding to the reaching paths for each node in an xtrie, one can construct an XML document conforming to the corresponding DTD that contains that path. In addtion, the set of all reaching paths to all nodes in an xtrie corresponds to all possible paths that can occur in any document conformant to that DTD.*

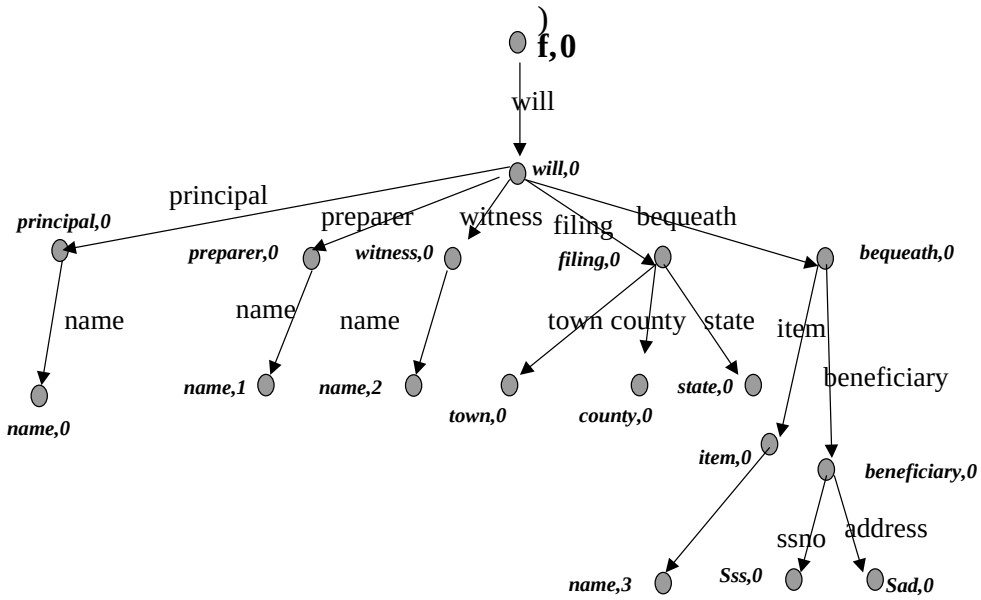

Figure 5: An xtrie constructed from the DTD shown in Figure 2

Proof Sketch: We show this by induction on the length of the path. The initialization step explores all paths of length 1, and inserts them into the xtrie; at this point, the agenda contains all the labels associated with elements reachable by paths of length 1. Now for the induction step, assume that the agenda contains labels of all the elements reachable by paths of length l, and the trie includes all those paths. For each such label, the iteration step explores paths of length $l+1$, inserts them into the xtrie, and adds the newly encountered elements into the agenda. The algorithm eventually terminates when no additional path lengthening is possible.

3.3.2 Using the xtrie

Given a DTD, we can pre-construct the xtrie. Now given a document D conforming to the DTD, we can find the subtrees of D associated with each path stored in the xtrie, and the rest of the process proceeds exactly as described in Algorithm 1 in Section 3.2. Since this algorithm is agnostic about the actual implementation of the table, it works exactly as before.

Algorithms 2 and 3 in Section 3.2 to match a query Q against the possible paths. We construct a path automaton corresponding to Q, and match it against the xtrie. This is a simple process, which we describe informally. The matching process marks nodes in the xtrie with states from the path automaton. Initially, the top-level nodes in the xtrie are marked with the initial states of the path automaton. We then match transition labels in the path automaton against edge labels in the xtrie. Whenever the labels match, the automaton "advances", and the other edge of the xtrie is marked with "destination state" of the corresponding transition. This process eventually terminates, since there are only a finite number of ways to mark xtrie nodes with path automaton labels. At any point, this process guarantees that the path automaton state will label a node if and only if the path automaton would be in that state when processing the path reaching that node. Upon termination, the xtrie will

have accepting labels on some nodes. These nodes correspond to the desired reaching paths.

Of all the parties involved in our approach, the client is most likely to be resource-limited; so she would benefit most from using the xtrie to match query automata against path queries. We note that she does not have to compute the xtrie herself; any trusted party could compute the xtrie from the DTD, and send it the client in an integrity-preserving manner.

With the naive approach, if the total length of all possible paths is N, and there are m states in the path automata, the matching process requires atmost $N * m$ comparisons. If there are n edges in the trie, the above process requires atmost $n * m$ comparisons. In the following section, we provide some empirical data concerning the relative sizes of n and N for published DTDs.

3.3.3 Xtries in practice

We empirically analyzed several published DTDs, to determine the number of elements the DTDs, the total length of all the possible paths through the DTD, and the size of the xtrie representation. The naive approach would require one to store all the paths explicitly in a table, and match a query against these paths; the xtrie is a more compressed representation. Our goal in doing this analysis was to determine the degree of compression actually achieved in practical, standards-based DTDs. We collected 100 separate DTDs from the OASIS DTD repository[4]. Out of these 37 were recursive. For the other 63, we enumerated all the paths, and counted the total length thereof; we also implemented our approach, and measured the total size of the xtries in each case. The results are shown in figure 6. We found that xtries always produce a smaller representation. This leads to a proportionate reduction in the effort required to evaluate queries, since fewer string comparisons are required.

Generally, however, one can expects that documents will be very large as compared to the DTDs; so the work done by the client with Algorithm 3 will be quite a bit less compared to with the work done by the owner and the query processor.

3.4 Certifying answers to selection queries

We now consider the selection queries as presented in Definition 2.2. To recapitulate: given a document D, and a query *select(d,q,r, p)*, where p is a Θ predicate, query *select(d,q,r, p)*, where p is a Θ predicate, we seek to return a set of subtrees $T_1, \ldots, T_n$ such that

1. T_i is a correct answer to the path query q, and
2. there exists a leaf node, reached by path s starting at the root of T_i such that the string value c at that leaf satisfies the predicate p regarding c.

We seek to provide a certified answer to this query. Such queries would be extremely useful with documents that contain many repeated elements, as in, say an XML document C which is a collection of traffic violation reports:

$$<!\texttt{ELEMENT records } (\texttt{trafvio}*) >$$

There may be millions of traffic-violations; somewhere under the violation, there would appear a `drlic` (driver's license) subtree, containing a social-security number in leaf element such as `ssno`. Clearly, an

[4]Please see http://www.oasis-open.org.

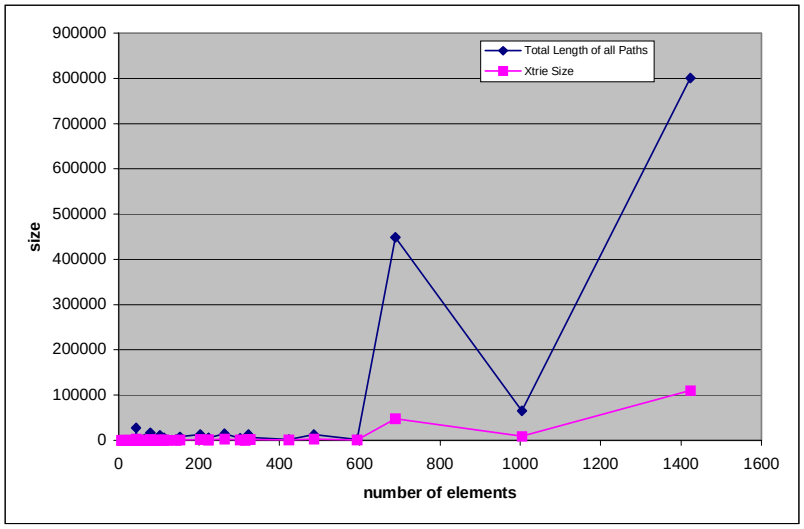

Figure 6: A plot showing the relative sizes of the naive approach (a table of all paths) versus the xtrie approach. The xtrie approach always has significantly lower storage requirements (on the average, by a factor of 5), *and* proportionately reduces query processing effort, particularly for larger, more complex DTDs; however, we note that there are few DTDs with over 500 elements. This graph includes 62 data-points; one outlier with a very large number of paths, and very high compression, was removed for clarity.

employer considering an applicant would like a complete list of all the violations in which the candidate had been involved, and could well desire a certified, complete and answer to a query such as:

$$select(\mathtt{C}, \mathtt{records.trafvio}, *.\mathtt{drlic}.*.\mathtt{ssn}, ssn = 11111111)$$

To answer this query we need to first locate the collection leaf nodes corresponding to the following path:

$$(\mathtt{records.trafvio}.*.\mathtt{drlic}*.\mathtt{ssn})$$

Once we've located this collection of leaf nodes, we need to search through this list to find the leaves with the string value identical to `111111111`. If we had an efficient index over these leaves (e.g., a binary or B-tree) we could search this list quickly. Once the leaves are found we need to find the subtrees of these leaves that are reached by the path `records.travfio` from the root element.

To answer such queries quickly, and provide compact certificates, we need only slightly modify the algorithms shown in Section 3.2. First we note that an efficient index structure over the leaves can support efficient searching. In addition, an index structure such as binary tree can be Merkle-hashed as shown in figure 4 to provide compact certificates for correct search procedures.

First, in Algorithm 1, step 4, if the current table entry corresponds to a leaf node, we build a search tree over the set of leaves, and Merkle-hash this search tree to give a root hash. This root hash gets associated with the table entry as before.

Second, when evaluating the query *select(D,q,r,p)*, we compute the table entries T_1 for the path query $q.r$ and the entries T_2 for the path query q. First, we search the index tree for the T_1 (which must be leaves) and retrieve the set of leaf values satisfying p, and build a certificate for this search procedure. This part of the process uses known methods for answer certification, from existing literature, as described in Section 2.2. Next, we search up from the answer leaves in the document tree to find the subtrees S reached by the path query q. DOMHASH values can be used to certify for each leaf value L that it occurs under an element of S; the use of DOMHASH for this purpose was reviewed in Section 2.2. Next, we can use the entry digests for the entries in T_2 to certify that all these entires in S are reached path satisfying the query q.

Finally, Algorithm 3 is modified slightly to check the certificates described above. The selection process using the Merkle-hashed search tree for the leaves reached by paths satisfying $q.r$ guarantees that we retrieved *all* the leaves that satisfy the answer; the DOMHASH guarantees that these leaves occur under sub-trees in S, and finally, the hash digest for entries satisfying the q path query guarantee that the subtrees in S are indeed reached by paths satisfying q.

How large are these certificates? Clearly, they are proportional to the size of the answer $\mid A \mid$. In addition, the search Merkle-hashed index tree induces a certificate of size $O(logS)$ where S is the number of leaves indexed in the tree. Clearly, S is bounded by the size of the document $\mid D \mid$. In addition, each leaf carries a DOMHASH chains certifying that the leaves occur under the desired subtree are of height $O(H)$, where H is the total height of the document tree. Thus, we end up with a certificate of size $O(\mid A \mid Hlog(\mid D \mid))$. We note that the entire document (e.g., the list of all traffic violations) are likely to be very large relative to the size of the answer, or the height (level of nesting) of the document.

4 Conclusion

XML is rapidly gaining in strength as the data model of choice for information on the Internet. Certifying the correctness of XML documents is clearly an important problem. The current draft XML signature standard only allows the certification of pre-determined pieces of XML documents. To certify parts of XML documents selected by content, it would be necessary to use an on-line signing key. To our knowledge, there has so far been no way to use one digital signature over an XML document to certify answers to arbitrary selection queries over such documents. We support just this functionality. Our approach currently works on non-recursive DTDs only. However empirical analysis indicates that a majority of published DTDs are non-recursive, and we believe our approach will be quite useful in a variety of contexts. We are also pursuing the elimination of this limitation.

References

[ABS200] S. Abiteboul, P. Buneman, D. Suciu, Data on the Web: From Relations to Semistructured Data and XML, Morgan-Kaufman Publishers, 2000.

[AV97] S. Abiteboul, V. Vianu: Regular Path Queries with Constraints. In *Proc. 16th ACM Symposium on Principles of Database Systems (PODS)*, 122–133, ACM Press, 1997.

[BC00] A. Bonifati, S. Ceri: Comparative Analysis of Five XML Query Languages. *SIGMOD Record 29:1 (March 2000)*, 68–79.

[BLL00] A. Buldas, P. Laud, H. Lipmaa, Accountable certificate management using undeniable attestations, *Proceedings of the 7th ACM conference on Computer and communications Security* November 1 - 4, 2000, Athens Greece

[Cla99] J. Clark: XSL Transformation (XSLT), Version 1.0. W3C Recommendation, Nov 1999.

[Cow01] J. Cowan: XML Information Set. W3C Working Draft, March 2001.

[CD99] J. Clark, S. DeRose: XML Path Language (XPath). W3C Recommendation, Nov 1999.

[DDP00] E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, P. Samarati: XML Access Control Systems: A Component-Based Approach In *14th IFIP 11.3 Working Conference in Database Security*, 2000.

[DDP00b] E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, P. Samarati: Securing XML Dcoument. In *6th International Conference on Extending Database Technology (EDBT)*, LNCS 1777, 121–135, Springer, 2000.

[DGM00a] P. Devanbu, M. Gertz, C. Martel, S. Stubblebine: Authentic Third-party Data Publication. In *14th IFIP 11.3 Working Conference in Database Security*, 2000.

[DH00] Digest Values for DOM (DOMHASH). RFC2803, http://www.landfield.com/rfcs/rfc2803.html, April 2000.

[XMLDSIG] D. Eastlake, J. Reagle, D. Solo, XML–Signature Syntax and Processing, Internet Draft, http://www.ietf.org/internet-drafts/draft-ietf-xmldsig-core-2-00.txt

[Fal01] D. C. Fallside: XML Schema Part 0: Primer. W3C Proposed Recommendation, Mar 2001.

[LC00] L. Dongwon, W.W. Chu: Comparative Analysis of Six XML Schema Languages Sigmod Record 29:3 (September 2000), 76–87

[GTS01] M.T. Goodrich, R. Tamassia, and A. Schwerin, Implementation of an Authenticated Dictionary with Skip Lists and Commutative Hashing, *DISCEX II*, 2001 (also *U. S. Patent Filing*)

[NN99] M. Naor and K. Nissim. Certificate revocation and certificate update. In *Proceedings, 7th USENIX Security Symposium*, 1999.

Appendix

Table of XML DTDs we analyzed. Note that a wide range of application domains are included. The table incldues the number of elements in the DTD, and whether the DTD was recursive. The table continues on into the next page.

name	#elements	recursion?
Interactive Financial Exchange (IFX)	689	no
Extensible Financial Reporting Markup Language (XFRML)	143	no
Open Financial Exchange (OFX/OFE)	1004	no
National Library of Medicine (NLM) XML Data Formats	100	no
Financial Products Markup Language (FpML)	73	no
Digital Signatures for Internet Open Trading Protocol (IOTP)	58	no
Investment Research Markup Language (IRML)	323	no
Market Data Markup Language (MDML)	48	no
Mortgage Bankers Association of America MISMO Standard	424	no
Data Link for Intermediaries Markup Language (daliML)	95	no
ACORD - XML for the Insurance Industry	1423	no
Real Estate Transaction Markup Language (RETML)	153	no
Active Digital Profile	303	no
Intrusion Detection Message Exchange Format	52	no
Customer Identity / Name and Address Markup Language (CIML, NAML)	315	no
History Event Markup and Linking	13	no
AND Global Address XML Definition	44	no
Marketplace XML (mpXML)	73	no
SMBXML: An Open Standard for Small to Medium Sized Businesses	78	no
EDGARspace Portal	51	no
Open eBook Initiative	29	no
HRML (Human Resources Markup Language)	62	no
Vocabulary Markup Language (VocML)	47	no
XML Encoding for SMS (Short Message Service) Messages	48	no
Molecular Dynamics [Markup] Language (MoDL)	15	no
Open Philanthropy Exchange (OPX)	65	no
Weather Markup Language (WeatherML)	45	no
XMLPay Specification	103	no
XML-MP: XML Mortgage Partners Framework	486	no
XML-MP: XML Mortgage Partners Framework	224	no
Trading Partner Agreement Markup Language (tpaML)	120	no
International Development Markup Language (DML)	72	no
adXML.org: XML for Advertising	35	no
Rosetta Group XML Résumé Library	58	no
ISIS European XML/EDI Healthcare Pilot Project (XMLEPR)	264	no
Clinical Data Interchange Standards Consortium	86	no
The CISTERN Project - Standard XML Templates for Healthcare	328	no
Marine Trading Markup Language (MTML)	25	no
VISA XML Invoice Specification	99	no
Guideline XML (gXML)	66	no
Printing Industry Markup Language (PrintML)	113	no
Schools Interoperability Framework (SIF)	224	no
bibteXML: XML for BibTeX	47	no
European Visual Archive Project (EVA)	16	no
Digital Property Rights Language (DPRL)	79	no
Microarray Markup Language (MAML)	88	no
Electronic Commerce Modeling Language (ECML)	28	no
Taxonomic Markup Language	31	no
Image Metadata Aggregation for Enhanced Searching (IMAGES)	43	no
XML Messaging Specification (XMSG)	9	no

name	#elements	recursion?
XML-Based 'Chem eStandard' for the Chemical Industry	203	no
Jabber XML Protocol	13	no
Common Profile for Instant Messaging (CPIM)	8	no
STEPml XML Specifications	67	no
Materials Property Data Markup Language (MatML)	39	no
XML for Multiple Sequence Alignments (MSAML)	11	no
Product Data Markup Language (PDML)	594	no
ECIX QuickData Specifications	16	no
Platform for Privacy Preferences (P3P) Project	82	no
IPDR.org Network Data Management Usage Specification	22	no
Point of Interest Exchange Language Specification (POIX)	33	no
Navigation Markup Language (NVML)	30	no
Petrotechnical Open Software Corporation (POSC) XML Related Projects	113	no
Navy CALS Initiatives XML	107	yes
COSCA/NACM JTC XML Court Filing Project	136	yes
W3C XML Specification	157	yes
FinXML - 'The Digital Language for Capital Markets'	175	yes
Open Catalog Protocol (OCP)	11	yes
eCatalog XML (eCX)	32	yes
Portal Markup Language (PML)	46	yes
XML for the Automotive Industry - SAE J2008	149	yes
NISO Digital Talking Books (DTB)	89	yes
Human Resource Management Markup Language (HRMML)	133	yes
US Patent and Trademark Office Electronic Filing System	323	yes
OMG Common Warehouse Metadata Interchange (CWMI) Specification	856	yes
BiblioML - XML for UNIMARC Bibliographic Records	225	yes
authoritiesML	79	yes
Alexandria Digital Library Project	83	yes
Bioinformatic Sequence Markup Language (BSML)	34	yes
Gene Expression Markup Language (GEML)	89	yes