

Flexible Authentication of XML Documents

P. Devanbu*, M. Gertz, A. Kwong, C. Martel, G. Nuckolls
Department of Computer Science
University of California
Davis, CA 95616, USA
{devanbu | gertz | kwonga | martel | nuckolls}@cs.ucdavis.edu

S. G. Stubblebine
Stubblebine Consulting, LLC
8 Wayne Blvd,
Madison, NJ 07940
stuart@stubblebine.com

Abstract

XML is increasingly becoming the format of choice for information exchange on the Internet. As this trend grows, one can expect that documents (or collections thereof) may get quite large, and clients may wish to query for specific segments of these documents. In critical areas such as healthcare, law and finance, integrity is essential. In such applications, clients must be assured that they are getting complete and correct answers to their queries. Existing methods for signing XML documents cannot be used to establish that an answer to a query is complete. A simple approach has a server processing queries and certifying 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 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 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

*We gratefully acknowledge support from the NSF ITR Program, Grant No. 0085961. Devanbu and Gertz were also supported by the Defense Advanced Research Projects Agency and the Space and Naval Warfare Systems Command under Contract Number N66001-00-8945. The content of the information does not necessarily reflect the position or the policy of the U.S. Government and no official endorsement should be inferred.

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

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.

Motivated by the above examples, we consider the following question: When an untrusted party returns parts of an XML document claiming them to be the complete and correct answer to a query, how can this claim be verified?

Existing approaches to signing XML documents [14] allow a server to use a single signature over an entire document to authenticate a given portion of the document; however, it is not possible to use the document signature to show that a set of document segments is a complete and correct answer to a query.

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 vulnerable to external hacking and insider attacks. At any rate, if the server itself is untrusted, then having it digitally sign answers is pointless. 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. Although our approach can be applied to any type of DTD, in this paper, we focus mainly on non-recursive DTDs; however in a survey of 100 publicly available DTDs, we found that 63 were non-recursive. So we believe our approach will be useful in many applications. Our approach extends easily to recursive DTDs, by simply bounding the recursive depth using the target document itself, where the recursion depth is naturally bounded.

In Section 2, we present background information on related technologies assumed in this paper and we also present 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 result of query Q over document D)
2. $C(Pk) \xleftarrow{\sigma_{Sk}(eval(Q,D))} S(D, Sk)$ (Server returns answer and verifiable digital signature to client)

In the above scenario, it is not practical to pre-compute all possible signatures for all possible answers; so the server needs to be trusted, and needs to securely maintain an on-line signing key. For greater security, as well as scalability, 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 publisher, who always provides a certificate of correctness with every answer. This simplifies the key management burden; for example, the owner can relegate the signing key to a smart card that he keeps locked up in a safe between updates. The detailed operation is presented in several steps, corresponding to the steps shown in Figure 1.

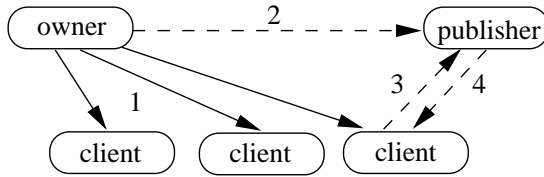


Figure 1: Authentic publication of XML documents. Document is created by owner, who uses a special approach to digesting it. Clients receive the digest through an authentic channel (1) and the document itself is sent to a publisher (2). In response to a query from a client (3) the publisher returns both an answer and a *verification object* certifying the correctness of the answer (4).

1. All *clients* receive from the *data owner* a signed, specially computed digest of the data using a digesting algorithm which relies on a keyless one-way hash function. This digest is transmitted securely, perhaps using a public-key signature. *With the exception of checking for a database update, this is the only step where a cryptographic signature is required.*
2. The data itself is transmitted to the *data publishers*.
3. A *client* submits a request to an untrusted publisher.
4. A *Client* receives back from the publisher an answer, and a specially computed *certificate* that lets her check that the answer is correct. The certificate uses only the keyless one-way hash function (*no private signing keys!*).
5. (Not shown in figure) The client runs a special *verification* procedure that compares the digest received from *owner* against the results of a particular hash calculation over the *publisher's* certificate. If the comparison succeeds, she accepts; otherwise, she rejects.

Various researchers have developed this approach in other settings [21, 11, 16, 3]; 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 exchange format.

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 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 [7] or XPath [8].

Assume a set Σ of element names (also known as *tags*) and a set S of string values disjoint from Σ . In the XML document model, an XML document D is a node-labeled tree described by 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 names, i.e., a function $V \rightarrow \Sigma$, and
- $elem$ is a mapping from vertices to other vertices or strings (also known as *sub-elements*), i.e., a function $V \rightarrow List(V \cup S)$.

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 $elem(v) \in S$. 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 D , 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 *label path*, denoted $path(v)$, which is a sequence of element names. Different node paths can have the same label 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) [4] 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 context-free 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 2 shows an example of an XML document in its linear form as well as an ordered, node-labeled tree. Figure 3 shows a DTD the document conforms to. In this DTD, the element **beneficiary** has as subelements **name**, **ssno**, and **address**. The element **will** is not a 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, have been proposed and studied in the literature but have not yet reached the same level of usage as DTDs (see [13] for a comparison of some 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 XQuery [5], XML-QL, Quilt, and XQL (see [2] 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* [1].

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 on the foundation of XPath [8], which, in turn, forms the basis of the widely used XSL Transformation [6]. 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.

```

<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>

```

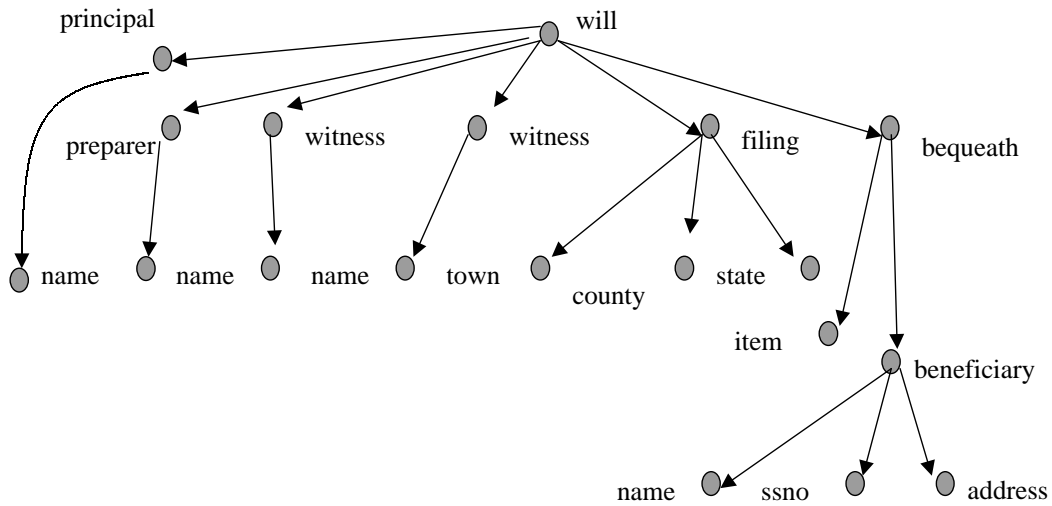


Figure 2: XML Document *D* in Linear and Tree Form. Text values are omitted.

```

<!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 item (#PCDATA)
<!ELEMENT county (#PCDATA)
<!ELEMENT state (#PCDATA)
<!ELEMENT address (#PCDATA)
<!ELEMENT beneficiary (name ssno address)>
<!ELEMENT ssno (#PCDATA)
<!ELEMENT name (#PCDATA)>
<!ELEMENT address (#PCDATA)>

```

Figure 3: DTD associated with XML document D in Figure 2

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 . \square

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 D , a path query p determines a (possibly empty) set of subtree structures in D such that the label path of each root node of such a subtree matches the expression p . For example, based on the XML document shown in Figure 2, the path query $*(\text{witness} \mid \text{bequeath}).*.name$ selects all subtree structures from D 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 $*(\text{witness} \mid \text{bequeath}).*$. For the document in Figure 2, there are several such subtree structures, with may contain text values (not shown in the figure) It is immediately evident that for a given path query, one can construct an equivalent finite state machine, known as a *path automaton* using the well-known transformation from regular expressions to finite automata. For example, the path query $(\text{will} \mid \text{bequeath}).*.beneficiary.name$ can be represented by the finite automaton shown in Figure 4.

We can formally define a path automaton \mathcal{PA} as a tuple $\langle \Sigma, Q_p, \alpha, F_p \rangle$, where Σ is a set of element names, 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 3. We can see that the element `name` occurs in several different contexts: under `principal`, `preparer`, `witness` and under `beneficiary`. Based on the DTD, it is evident that the answer to the query shown in Figure 4 is the subtree that occurs

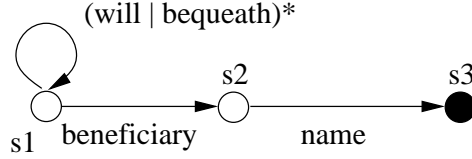


Figure 4: Path Automaton for path query $(\text{will} \mid \text{bequeath})^*.\text{beneficiary}.\text{name}$

under the element `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 $(\text{will}.\text{witness}.\text{name})$. 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 can 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 label 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 also satisfy certain conditions on the values of text-nodes. These two aspects resemble the same functionality as projection and selection provided in 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 $\text{elem}(v) = s, s \in S$. A selection query against a document D is composed of two parts: (1) a path query that determines subtrees in D , and (2) a selection part that specifies a condition on the leaf nodes in the selected subtrees.

Definition 2.2 (Selection Query) *Given two path queries q and s , and a comparison $p \equiv e\Theta c$ where $\Theta \in \{=, <>, <, >, \leq, \geq\}$, $c \in S$ is a string constant, and $e \in \Sigma$. A selection query, denoted $\text{select}(D, q, s, p)$, returns all subtree structures T_1, \dots, T_n in a document D 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 text value at that leaf e satisfies the comparison predicate p .

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 a typed schema underlying XML documents (e.g., XML Schema [15]), all values associated with leaf nodes are assumed to be strings. Consider, for example, the selection query $\text{select}(\text{will}.\text{witness}.\text{name}, \text{text} = \text{"Barb Witness"})$ on the XML document shown in Figure 2. There are two subtree structures rooted with a node labeled `witness`. Among these two, only the those are selected as the result to the query that have 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 [20], as illustrated in Figure 5. 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 securely distributes the root hash 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.

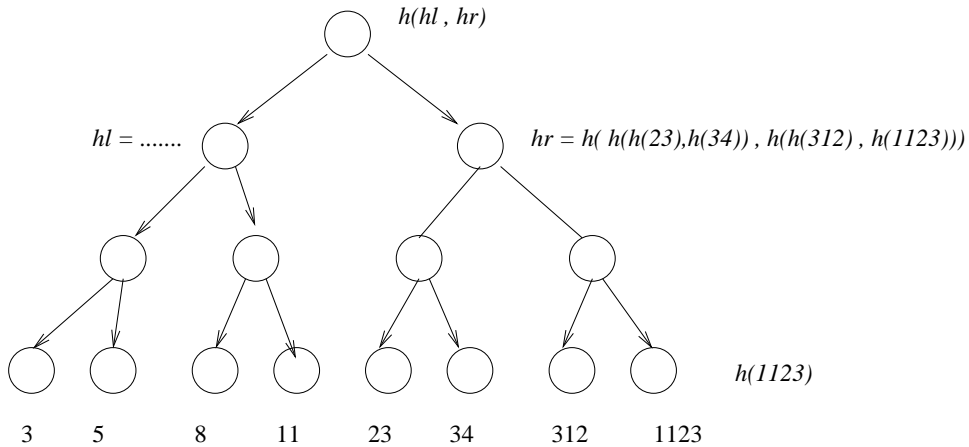


Figure 5: A Merkle hash tree associates hash values with nodes in a tree, in this case a binary tree. The leafs are sorted values in the binary tree. The leafs get the hash of the values, and each interior node gets the hash of all the values of all its children, combined in a suitable way. Once the root digest is authenticated by a trusted party, anyone can certify answers to queries, using only the hash values to provide evidence of a correctly conducted search

Merkle trees were used in this way in [21] for proving the presence or absence of certificates on revocation lists. For example, in figure 5 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 can recompute the root hash, and thus be sure (subject to the security of the hash function) that the value 23 did occur in the tree. 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 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. We extended this line of work, and introduced the notion of *authentic publication*, beginning with some forms of querying over relational databases [11]. The same idea has been used over skip lists [16] and in other, related settings [3]. When used with hierarchical “divide-and-conquer” type search data-structures, this approach provides answer certificates whose size is in the order of $A + \log(D)$ where A is the size of the answer and D is the size of the data set used in constructing the search

data structure. Recently [19] we present a general result showing that data structures admitting efficient search procedures can usually be adapted to authenticated versions with similar overheads. To answer selection queries in the XML context (details in Section 3.2) we sort the leaf values and build an index data structure 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.

These techniques are useful in the context of XML documents (particularly for doing selections, we make use of divide-and-conquer index structures). However, XML data is not as structured, and requires some additional machinery to certify answers to queries. In this sense, our work can be viewed as building an index to answer path queries, and then certifying this index.

2.3 Hashing and Signing XML Documents

As described earlier, XML documents have a simple data model based on trees. DOM [12] 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 5. 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 elements within the document. The full details of DOMHASH are available elsewhere [12]. 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 P , it is possible to provide evidence to P that any subtree τ of the document occurs under D without revealing all of D . First, note that P can DOMHASH the subtree τ to get the root hash of τ . Now, P can be given just the hash values of the siblings of τ and the siblings of all its parents, and P can recompute the root hash of D . Since the hash function is assumed to be one-way, P can be reasonably sure that the hash values could not have been forged, and that τ really did occur in D . The same process can be used to prove that one subtree τ_1 occurred under another subtree τ_2 within the same document, using the hash values along the path from τ_1 to τ_2 , without revealing any of the other subtrees under τ_2 ; we can then hash to the root value, as before.

The XML Digital signature recommendation [14] essentially computes a signature over a digesting procedure such as DOMHASH. The proposal 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. While the XML signature approach allows the use of a single signature over an entire document to authenticate any given part of that document, it cannot be used to certify that the answer to a path query is complete (e.g., is this really the *entire* set of traffic violations in all precincts by drivers who have the social security number 555-55-5555?). The goal of this paper is eliminate this limitation: we introduce an auxiliary structure, the *xtrie*, that is based on the DTD, and enables a publisher to certify completeness of an answer to a query. For our purposes, we assume an XML document digital signature based on DOMHASH.

3 Certifying Answers

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 vital property of non-recursive DTDs: 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, beginning with pure path queries. 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. We 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 data structure using some empirical data. After describing the handling of pure path queries, we finally describe how to use the xtrie, in conjunction with auxiliary data structures, to perform selection queries.

Some of the above steps are dependent only on the DTD, and can be carried out once per DTD; the results can then be reused for every document conforming to that DTD. In this paper, we mainly focus on non-recursive DTDs. In a survey of 100 different publicly available DTDs at www.oasis-open.org we identified 63 non-recursive DTDs. However, our approach easily extends to recursive DTDs, as we will show in Section 3.3.4. In this case, we simply use the existing XML documents to enumerate all paths that occur in the documents rather than by enumerating possible paths from a given DTD.

3.1 Path Queries in the Context of non-recursive DTDs

Given a non-recursive DTD, there are only finitely many different meanings for the infinitely many *syntactically different* path queries. As an analogy, consider 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.

Definition 3.1 *Two path queries q_1 and q_2 are semantically distinct with respect to a DTD if there exists a document D that conforms to that DTD, and for which q_1 and q_2 give different answers.*
□

Lemma 3.2 *Given a non-recursive DTD, there are only a 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 is bounded. Now given any node n in D , the number of *different* elements 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 content

model includes a Kleene Star). However, the number of different elements that can occur below any node is bounded by the (finite) number of different elements that appear in the corresponding ELEMENT content model. 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 label 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 distinct 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 DTDs. As it can be seen from the table in Figure 7, 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 A Naive Approach to Flexible Certification

We now consider a naive approach to storing potential answers to path queries in a table, called a *path table*, and certifying this table. We also show that this approach is secure, if the operative one-way hash function is secure. 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 distinct paths in a non-recursive DTD, second, that an XML document and subtrees thereof can be securely hashed using DOMHASHes which are hard to forge; 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 (and accept correct ones) unless the adversarial party that built the certifier managed to find a collision in 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 DTD². It first processes the document, associating with each subtree the label 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 the label 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. The set of all the table entries are digested together using another Merkle 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}

²We note here that this table can be produced independently by each party from the DTD.

1. Process the document D , associating each root node of a subtree in D with an element name. Give each subtree a number, using an ordered (e.g., DFS) traversal that guarantees that earlier parts of the document get a lower number.
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 correctly transmitted it to clients, the untrusted server can process path 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 label path, (2) the hash path from that entry to the table digest for τ , and (3) all the subtrees that are associated with that table entry. Build this certifier 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 now 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 (empty, document-independent) 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. For each triple, (a) DOMHASH each returned subtree, (b) build a Merkle tree out of the DOMHASHes and compute the root hash for the subtrees in the triple, (c) use the hash path provided, beginning with the entry digest computed in step 3(b), 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, a set σ of returned certifier triples is claimed to have been constructed (possibly by an untrusted party) as described in Algorithm 2. Now the client, who has received $sd(D)$ and σ , 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 uses the DTD to compute the precise set of table entries which match his query, and expects from the publisher 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 Algorithms 1 and 2 and comes up with the same root signature $sd(D)$. We now argue that the client will reject incorrect answers and certifiers. It is 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 an 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. ■

Note that τ contains entries for all possible paths based on a given DTD. For a specific document D conforming to the DTD, there may be many fewer paths which actually occur in D . We could construct a path table specialized to D but then we would need to securely transmit this list of paths to clients for each document D . Also note that this approach also works with recursive DTDs since we base the table construction on the document and not on a DTD.

3.3 Efficient Path Storage using an xtrie

In a DTD, we usually find that several different element tags can occur under a single element. For example, under `beneficiary`, in our running example, we have `name`, `ssno`, and `address`. This leads to many shared prefixes among paths; thus, all of the above elements occur under the shared path prefix `will.bequeath`. The naive path table that stores all paths wastes space; in addition, matching a path query against each entry in the table wastes repeated effort matching identical, repeated prefixes. Below, we present a more compact and efficient data structure, an xtrie, for storing the set of possible paths based on 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 that arise from a non-recursive DTD, and provide empirical evidence of its compactness. We also show how to extend xtries to recursive DTDs. 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 name of the subelement. Thus, in our running example, there would be a node corresponding to the `witness` element, and an edge labeled `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 every time a path prefix is shared among several different paths. Since DTDs allow the same element to occur under different elements, there may be many different labeled 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 occurrence bumps the counter, and creates a new copy of that element in the xtrie.

Algorithm 3.4 *An xtrie corresponding to a DTD is a directed tree $\langle N, E \rangle$ with node and edge labels and is constructed as follows:*

Initialize

1. Let Σ be the set of element names that occur in the given DTD. Initialize a table of counters, $c_e \leftarrow 0$, for each $e \in \Sigma$.
2. For the document root element with label s in the DTD, add a node labeled $(s, 0)$ to N .
3. Add an extra top-level node labeled $(f, 0)$ to N ; for the element $s \in \Sigma$ that is the document root element, we add an edge from $(f, 0)$ to the node $(s, 0)$. This edge is labeled s and added to E .
4. Initialize an agenda list A containing $(s, 0)$.

Iterate

While nodes remain in A , remove each node, (a, i) currently in the agenda list. For the `ELEMENT` rule associated with the element named a , if the DTD rule for a mentions an element t , then we add a new node labeled (t, c_t) to N ; we add this new node to the agenda A to be processed in the next iteration; we add an edge from (a, i) to (t, c_t) , labeled with the name t to E , increment $c_t \leftarrow c_t + 1$.

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 an extra “top-level” node. This top-level node has an edge (labeled with the root element name) to the root element node. The counters associated with each element get initialized at this point, and the agenda is initialized with the root element node. At this point, the maximum path length l encountered is 1.

Our construction algorithm is doing a breadth-first-search of possible label paths resulting from the DTD. Thus at each step of the ensuing iteration, the xtrie represents all paths of length $1 \dots l$, from the “top-level” node to nodes in A . The iteration step now looks at nodes in the agenda, grows the paths by 1, and adds the nodes newly reached to the node set N of the xtrie and also adds the appropriately labeled edges. It is easy to see that the algorithm terminates for non-recursive DTDs: there can only be a finite number of nodes (element names) ever added to the agenda. Elements (rather, (element name, count) pairs) are added to the agenda when they occur under another element. Each element can occur only a finite number of times under different 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 6. We note that redundancy is avoided in storing the three paths beginning with the common prefix `will filing`. We also note that the element name can occur in

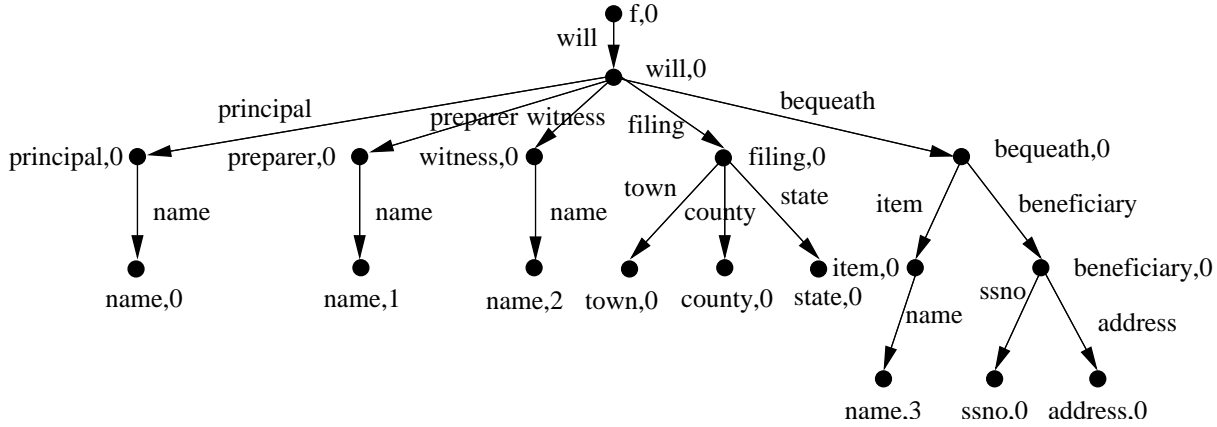


Figure 6: An xtrie constructed from the DTD shown in Figure 3

different places, under *witness*, *preparer*, etc, and so there are several copies of *name*, reachable through the different possible parent elements³. Xtries, as we have described them, resemble in some ways *schema graphs* and *Dataguides* [1] 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 specified by a DTD.

Definition 3.5 *The sequence of labels $s_1 \dots s_k$ on the edges leading to a node (s_k, c_k) from the top-level node $(f, 0)$ is called the reaching path for the node (s_k, c_k) in the xtrie. \square*

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

Lemma 3.6 *Corresponding to the reaching path for each node in an xtrie, one can construct an XML document conforming to the corresponding DTD that contains that path. In addition, 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.*

Proof: We show this by induction on the length of the reaching 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 xtrie includes all those reaching 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. Thus we now have all reaching paths of length $l + 1$. The algorithm eventually terminates when no additional path lengthening is possible. \blacksquare

3.3.2 Using the xtrie

Given a DTD, we can construct the xtrie. Each node in the xtrie corresponds to a possible reaching path in an XML document conforming to the DTD. Thus, given a document D conforming to the

³We note here that an element W can occur several times under an element X , as per the DTD, but the xtrie will show only a single node, labeled (W, n) under the node (X, m) (for some m and n)

DTD, we can find the subtrees of D associated with each in the xtrie, and the rest of the process proceeds in a manner analogous to Section 3.2.

First, we must securely digest the xtrie. The digesting algorithm is a special case of the algorithm described in [19] for digesting generalized search dags. Each node of the xtrie has associated with it 1) a set of pairs of document subtree identifiers reached by that path and either their DOMHASHes, if the subtree is not a leaf, or their values, if the subtree is a leaf (Later in Section 3.4, when we consider path-based selection queries, we describe how we speed up the selection by using an index over the values, rather than just the raw set of values), and 2) the digests of all the xtrie nodes under this node. These two should be in the order they occur in the document. These values are hashed together to give the digest associated with that xtrie node. Note that for a given document there may be no matching subtrees for this path, which in turn means there will be no matching subtrees for any children of this xtrie node. In this case we can associate a special NULL value with the xtrie node. In the next section we will see that this makes it easy to construct a pruned version of the xtrie that optimizes the query processing for specific documents that do not contain certain paths.

Algorithms 2 and 3 in Section 3.2 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 (see Section 2.1). We then match transition labels in the path automaton against edge labels in the xtrie. Whenever the labels match, the automaton “advances” and the next node 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. A similar approach is used for string-matching regular expressions over normal tries in computational biology applications [17].

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 to 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 at most $N * m$ comparisons. If there are n edges in the xtrie, the above process requires at most $n * m$ 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 in the DTDs, the total length of all the possible paths through specified the DTD, and the size of the xtrie representation. The naive approach would require 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 repository at www.oasis-open.org. 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 7. 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.

In addition, it is reasonable to expect that documents will be very large as compared to the DTDs (and thus, by proxy, the xtries). Therefore the work done by the client with Algorithm 3 will be quite a bit less than the work done by the owner and the query processor.

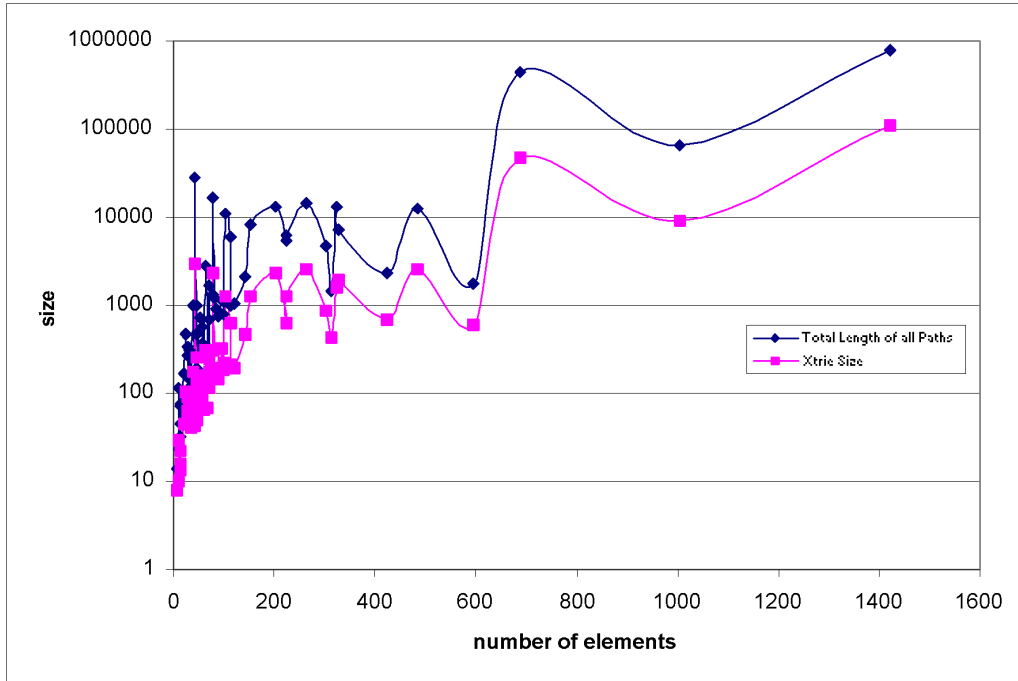


Figure 7: A log plot showing the relative sizes of the naive approach (a table of all paths) versus the xtrie approach. The xtrie approach always has significantly lower storage requirements (on the average, by a factor of 5), *and* proportionately reduces query processing effort, particularly for larger, more complex DTDs; however, we note that there are few DTDs with over 500 elements. This graph includes 62 data-points. One outlier, with over 3 million possible paths, and a compression factor of over 15, was omitted.

3.3.4 Xtries for recursive and large DTDs

When the DTD is recursive there is no bound on the xtrie size. However, for any given document D conforming to the DTD, an xtrie representing all paths in the document is of finite size (and is typically much smaller than the document). Even for non-recursive DTDs, a document D conforming to the DTD may have paths corresponding to only a small fraction of the xtrie. To handle these settings we now present an alternate approach where the publisher proves that certain parts of the xtrie do not match any paths in the document D , and thus the client need not expand those parts of the xtrie.

The process we describe can be viewed as a special case of the algorithm described in [19], which shows that whenever a query on a data-structure can be viewed as searching a directed acyclic graph (DAG), we can digest such a data-structure and then verify answers to queries. We also note

here that if an xtrie node has no actual paths in D , then the set of matching subtrees associated with that node (as described in the prior section) is empty.

For this verification process we assume that the client knows the DTD, the root digest of the xtrie, and the path query to be answered. Using this, the client can start building the xtrie as in Algorithm 3.4. The first step is to produce all the children of the root of the xtrie. The client will be provided the digest values associated with each of these child nodes (which she can hash together to verify they match the root digest value). If child node (c, i) has a digest value "null" (a special value for no matching subtrees), then the client knows that D has no paths to (c, i) in the xtrie, and there is no need to expand (c, i) further. For example, in our sample xtrie of Figure 6, if the D had no `preparer` then the node `(preparer,0)` would have a null digest value, and this would be proved to the client (who would then avoid expanding the children of `(preparer,0)`).

Next, for each child node with a non-null digest, the client expands that node using the DTD. Again, the client will be provided the digest value for each of these nodes (e.g. for `(town,0)`, `(county,0)` and `(state,0)` as children of `(filing,0)` in our sample xtrie). Again, the client can hash these together with their parent's subtree digest and then verify that this matches the digest value of the parent. Using this method, we can readily instantiate the generalized proof from [19] to show that the client accepts the answer only if the correct xtrie and associated digest values are provided, or the publisher has found a collision in the hash function.

This expansion continues until we hit the leaves of the xtrie, or until every unexpanded node has been shown to have a null value. The client now has an xtrie representing all paths which actually occur in D . This lets the client continue as in the prior section to match relevant xtrie nodes to the path query using the path automaton, and verify that the actual subtrees returned by the publisher hash to the digest values for the selected nodes.

We note that a further reduction in the size of the xtrie expanded can be achieved by having the client use the path automaton to prune the xtrie so that only xtrie nodes which are part of a path matching the current query are expanded. For example, with the path query of Figure 4 applied to the xtrie in Figure 6 the client can determine that `(bequeath,0)` is the only child of `(will,0)` which can match the query, and thus needs to be expanded.

Thus we considered three approaches: compute the full xtrie from a DTD, compute an xtrie for D stopping at null nodes, and compute the part of the xtrie which matches the path query (this last could also be combined with stopping at null nodes for a specific D). Each of these approaches could be best depending on the DTD, the document D , the path query q and the number of queries being done on a specific document or DTD.

We note in conclusion that there are two parts to authenticating an answer: *correctness* (i.e., is the answer really contained in the original document ?) and *completeness* (i.e., does the answer include every part of the document satisfying the query ?). As noted earlier, DOMHASH already enables correctness verification. We introduce the xtrie for completeness verification. Finally, we also describe some ways of extending xtries to handle recursive DTDs, and also present some further optimizations for dealing with large documents, and/or documents that don't contain all the subtrees and paths allowed for in the DTD.

3.4 Certifying Answers to Selection Queries

We now consider selection queries as presented in Definition 2.2. To recapitulate: given a document D , and a query $select(D, q, s, p)$, where p is a comparison predicate, we seek to return a set of subtrees

T_1, \dots, 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 label path s starting at the root of T_i such that the string value at that leaf node value satisfies predicate p .

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

```
<!ELEMENT records (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 a leaf element such as `ssno`. Clearly, an 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 answer to a query such as:

```
select(C, records.trafvio, *.drlic.ssn, text = 11111111)
```

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

```
(records.trafvio.*.drlic*.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 `11111111`. 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.trafvio` from the root element.

To answer such queries quickly, and provide compact certifiers, we need only slightly modify the algorithms shown in Section 3.3. First, we note that an efficient index structure over the leaves can support efficient searching. In addition, an index structure such as a binary tree can be Merkle-hashed, as shown in Figure 5, to provide compact certifiers for correct search procedures.

We modify the digesting procedure given in Section 3.3.2. If the given document (that is to be subject to selection queries) has leaf nodes corresponding to a given xtrie node, we build a search tree (see [11]) over the set of leaves, lexicographically ordered, and Merkle-hash the resulting search tree to give a root hash. This root hash gets associated with the xtrie node, rather than just the set of raw leaf values (see the second paragraph in Section 3.3.2).

Second, when evaluating the query $select(D, q, s, p)$, we compute the xtrie nodes X_s for the path query $q.s$. First, for each element of X_s we search the index tree for its leaves, retrieve the set of leaf values satisfying p , and build a certifier 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 that it occurs under an element of S ; the use of DOMHASH for this purpose was reviewed in Section 2.2.

Finally, the verification procedure is modified slightly to check the certifiers described above. The selection process using the Merkle-hashed search tree for the leaves reached by paths satisfying $q.s$ guarantees that we retrieved *all* the leaves that satisfy the answer. The hash values of each individual leaf, along with the DOMHASH structures, guarantees that these leaves occur under sub-trees rooted at paths that match q .

There are several factors that contribute to the size of the certifier. First, the number of subtrees satisfying the query (which is $|S|$). Second, the number of entries in the xtrie matching $q.s$ (call this T , which is the same as $|T_s|$). Third, the total number of leaves satisfying the selection criterion (call this M). Each entry in T_s has a Merkle-hashed index tree over the leaves, which is searched. This induces a certifier of size $O(M_i + H_x + \log L)$, where M_i is the number of leaves matching the selection in this particular entry, H_x is the height of the xtrie, and L is the number of leaves indexed in the tree under this entry. Clearly, L is bounded by the size of the document $|D|$. In addition, each leaf carries a DOMHASH chain of height $O(H_d)$ certifying that the leaf occurs under the desired subtree, where H_d is the total height of the document tree. So we get a certifier of size $O(H_d * |S| + M + T * (H_x + \log |D|))$. We note that the entire document (e.g., the list of all traffic violations) is likely to be very large relative to the size of the answer $|S|$ or the height H_d (level of nesting) of the document.

4 Conclusion

XML is rapidly gaining in strength as the data model of choice for representing and exchanging information on the Internet. Certifying the correctness of XML documents is clearly an important problem. The current XML signature recommendation 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. Thus far, there has 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 works best on non-recursive DTDs. 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. A similar approach to authenticated publication of XML data where no DTDs are given is presented in [18].

References

- [1] S. Abiteboul, P. Buneman, D. Suciu: Data on the Web: From Relations to Semistructured Data and XML. Morgan Kaufman, 2000.
- [2] A. Bonifati, S. Ceri: Comparative Analysis of Five XML Query Languages. *SIGMOD Record* 29:1, 68–79, 2000.
- [3] A. Buldas, P. Laud, H. Lipmaa: Accountable Certificate Management using undeniable Attestations. In *Proc. 7th ACM Conference on Computer and Communications Security*, 9–17, ACM, 2000.
- [4] T. Bray, J. Paoli, C. Sperberg-McQueen: Extensible Markup Language (XML) 1.0. W3C Recommendation, February 1998.
- [5] D. Chamberlin, J. Clark, D. Florescu, J. Robie, J. Simeon, M. Stefanescu: XQuery 1.0: An XML Query Language. W3C Working Draft, 2002.
- [6] J. Clark: XSL Transformation (XSLT), Version 1.0. W3C Recommendation, Nov 1999.
- [7] J. Cowan: XML Information Set. W3C Recommendation, October 2001.
- [8] J. Clark, S. DeRose: XML Path Language (XPath). W3C Recommendation, Nov 1999.

- [9] 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: Data and Applications Security*, 39–50, Kluwer Academic Publishers, 2000.
- [10] E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, P. Samarati: Securing XML Document. In *6th International Conference on Extending Database Technology (EDBT)*, LNCS 1777, 121–135, Springer, 2000.
- [11] P. Devanbu, M. Gertz, C. Martel, S. Stubblebine: Authentic Third-party Data Publication. In *14th IFIP 11.3 Working Conference in Database Security: Data and Applications Security*, 101–113, Kluwer Academic Publishers, 2000.
- [12] Digest Values for DOM (DOMHASH). RFC2803, <http://www.landfield.com/rfcs/rfc2803.html>, April 2000.
- [13] L. Dongwon, W.W. Chu: Comparative Analysis of Six XML Schema Languages Sigmod Record 29:3 (September 2000), 76–87
- [14] D. Eastlake, J. Reagle, D. Solo: XML–Signature Syntax and Processing, W3C Recommendation, February 2002.
- [15] D. C. Fallside: XML Schema Part 0: Primer. W3C Recommendation, May 2001.
- [16] M.T. Goodrich, R. Tamassia, and A. Schwerin: Implementation of an Authenticated Dictionary with Skip Lists and Commutative Hashing, In *DISCEX II*, 2001 (also *U. S. Patent Filing*).
- [17] D. Gusfield: *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*, Cambridge University Press, 1997.
- [18] A. Kwong and M. Gertz: Authentic Publication of XML Document Data. In *2nd International Conference on Web Information Systems Engineering (WISE 01)*, 331–340, IEEE Computer Society, 2001.
- [19] C. Martel, G. Nuckolls, P. Devanbu, M. Gertz, A. Kwong, S. Stubblebine: General Model for Authentic Data Publication, www.cs.ucdavis.edu/~devanbu/files/model-paper.pdf
- [20] R.C. Merkle: A Certified Digital Signature. In *Advances in Cryptology–Crypto ’89*, 1989.
- [21] M. Naor and K. Nissim: Certificate Revocation and Certificate Update. In *Proceedings, 7th USENIX Security Symposium*, 1999.