UPDATING ENCRYPTED XML DOCUMENTS ON UNTRUSTED MACHINES

Prakash Reddy, Robert N. Mayo, Eamonn O'Brien-Strain, Jim Rowson, and Yuhong Xiong
Hewlett-Packard Labs

Abstract: With XML and other data types becoming increasingly used in distributed systems, we have a need to update this data in a way that preserves privacy and integrity. Prior work has developed ways of encrypting XML documents for privacy, and adding integrity codes to ensure that the data is not tampered with. In this paper we present an algorithm that allows XML documents, or other tree-structured data, to be updated without decrypting them. In our model of a distributed system, several trusted machines have access to the decrypted form of a document and may request changes to it. These change requests are encrypted and sent to an untrusted update machine for processing. The update machine is able to take the original encrypted document, apply the encrypted changes, and produce an updated encrypted document. In addition, an integrity code is produced that proves the untrusted machine performed the update correctly. In practice, our algorithm allows trusted machines in a distributed system to send incremental updates to a storage server, even if that server is not allowed access to the clear text.

Key words: XML, Security, Incremental Cryptography, XOR MACs, Incremental change support

1. INTRODUCTION

Distributed systems often have the need to cache data at many locations, and then propagate incremental changes to the data [8]. For instance, consider the case of several mobile computers that each cache a shared calendar of meetings, with the master copy of the shared calendar stored at an Internet storage provider that should not be allowed to inspect the calendar. When a user of a mobile computer changes a meeting time in his copy of the calendar, it needs to tell the storage server to update the master calendar appropriately.

In our model, we consider the storage server to be untrusted, in that it is not allowed access to the clear text meeting times. The mobile computers, however, are trusted as they are part of the workgroup that has access to the calendar. We would like to store our data in encrypted form, to prevent snooping, on the storage server, and add an integrity code to verify that the storage server hasn't modified the data inappropriately.

However, there needs to be a way for the storage server to update the documents it holds. Our algorithm presents a way for the mobile computers to send encrypted update requests, called *deltas*, to the storage server. Deltas are an application-generated set of consistent changes. Deltas are encoded such that the storage server can perform the updates without needing access to the clear text. This is done using a form of incremental cryptography [2] [3]. Incremental cryptography is a set of algorithms that can re-compute cryptographic functions in time proportional to the amount of change in the document, rather than in proportion to the overall size of the document. Our technique, however, is novel in that the updates can be processed by an *update server* without knowledge of the key.

2. BASIC CONCEPTS OF XML

XML, or eXtensible Markup Language [5], is a popular way of representing hierarchical data using a textual representation. The standardization of the syntax and the textual representation make it a useful and neutral interchange format between systems and organizations.

Each XML document contains a single tree of nodes, each node called an element. Each element begins and ends with a tag, as in "<employee> ... </employee>. Data for the element may be associated with the element tag as name-value pairs called attributes, and/or it may be included as free-form text between the tags. Child elements may also appear between the tags.

In the example below, `web="lime.org"` is an attribute containing data for the element `base`. The text `OK rock.` is free-form text associated with the `base` element, while `color` and `value` are child elements.

```
<base web="lime.org">
        Ok rock.
        <color>White</color>
        <value basic="cheap">
                <dollars>29.33</dollars>
        </value>
</base>
```

3. OUR SYSTEM MODEL

In our model of the system, we have trusted *update generators*, untrusted *update processors*, and trusted *verifiers*. In the introduction, we gave an example where the update generator was a mobile computer with a user's calendar, while the update processor was a storage server. In addition, in that example the mobile computers are also trusted verifiers, in that they verify the integrity of a document when they receive it.

Update generators request modifications by preparing *deltas*, or lists of changes. These are sent to update processors. For example, our storage servers store the documents safely. But, acting as update processors, they also incrementally update the documents with deltas from the mobile computers. Trusted verifiers verify that the updates were done properly.

4. ALGORITHMS

XML documents in our system are encoded, encrypted, and then appended with a Message Authentication Code (MAC). We present algorithms for each of these three steps. In addition, documents can be updated by applying deltas. In the process of applying deltas, it is possible for conflicts to occur. We discuss options and algorithms for handling these.

4.1 Encoding and encrypting the XML

XML documents are tree structures. Our encoding and encryption scheme is designed to protect the content as well as the hierarchy. Hiding the hierarchy prevents unauthorized persons from inferring the structure of the information from the encrypted document.

We will describe our process in two phases: encoding and encryption. In the encoding phase, the tree-structured document is flattened into an unordered list of elements, or nodes. The encryption phase then encrypts these elements.

In XML, every node has some data (tags, attributes, and text), possibly some child pointers, and a parent pointer (unless it is the root node). An example document is shown below:

```
<base web="lime.org">
      Ok rock.
      <color>White</color>
      <value basic="cheap">
            <dollars>29.33</dollars>
```

```
          </value>
     </base>
```

We will describe the encoding process using approximate textual descriptions. An implementation of the algorithm would bypass these textual descriptions, working directly on the data structures instead.

The first step in our encoding is to assign an arbitrary ID to each element. This is shown below in brackets:

```
     <base[109] web="lime.org>
          Ok rock.
          <color[558]>White</color>
          <value[971] basic="cheap">
               <dollars[623]>29.33</dollars>
          </value>
     </base>
```

Now that elements have IDs, we can flatten the hierarchy into an unordered list of nodes, in no particular order. Each line contains its own node ID, the node ID of its parent, and then the contents of the node with IDs substituted for the children:

```
[971] 109, <value basic="cheap" [623]>
[623] 971, <dollars>29.33</dollars>
[109] null, <base web="lime.org>Ok rock.
[558][971]</base>
[558] 109, <color[558]>White</color>
```

To encrypt the document, we encrypt each line independently. We leave the initial node ID in clear text, and add a random number to the start of the line's encrypted data. Thus, using encryption function *E ()*, we have:

```
[971] E(10290304, 109, <value basic="cheap" [623]>)
[623] E(98740123, 971, <dollars>29.33</dollars>)
[109] E(57093489, null, <base web="lime.org>Ok
rock.[558][971]</base>)
[558] E (95347892, 109, <color [558]>White</color>)
```

The random number strengthens the encryption. This is needed because XML documents have certain properties: a document often has multiple identical nodes, and the amount of data in a node may be small. Adding the random number allows each node to be unique, and also lengthens the amount of data to be encrypted. The choice of the encryption algorithm and the length of the random number are not discussed here. They should be chosen to provide sufficient security for the application, while still meeting any size and performance constraints imposed by the application. We will rely on the uniqueness of each node, provided by the random number, in the conflict resolution algorithm, presented later.

4.2 Encoding the Deltas

Given a document, it is possible for a trusted update generator to request modifications to the original document by preparing a *delta*, which is a list of commands, each of which is one of the following:

- **ADD *id*, E(*txt*)**
  This adds a new node to the XML document. The encrypted data contains the same data used to represent a node, explained previously.

- **DELETE *id***
  This simply deletes the specified node from the document.

- **REPLACE *id*, H(E(*txt_old*)), E(*txt*)**
  The specified node is replaced with the specified data. The ID of the node is not changed. We include a hash of the old encrypted value of the node, to allow the update processor to uniquely identify the data in the node. This command is similar to a DELETE of the node, followed by an ADD using the same node ID.

Addition and deletion will affect the parent nodes, in that their child pointers change. Therefore ADD and DELETE commands will be followed by REPLACE commands to replace the parent nodes with correct ones. Since the node ID of the parent does not change, no further propagation of the changes is needed. Since we require that deltas be consistent, it is assumed that when an intermediate node is deleted then the command list also contains commands to delete all the descendent nodes.

While the delta containing these commands is generated on a trusted machine, they can be applied to the encrypted document on an untrusted update machine, which is the primary benefit of our system.

4.3 Applying the Deltas

Applying the delta involves executing the ADD, DELETE, and REPLACE commands. For ADD, the untrusted machine simply adds the ID and encrypted text into the document as a new line. For DELETE, it simply deletes the line with the specified ID. REPLACE substitutes the specified data for the data that is found in the file for that node. If the hash of the data that is found in the document does not match the value specified in the REPLACE command, we consider it a conflict (described later).

4.4 Document Integrity

Since our documents are modified by untrusted machines, we need to verify that the document was properly processed. MACs have been used in other systems for a similar purpose: verifying that the document was not altered in transit. In these systems, a MAC is generated by a sender using the document and a shared key, and the document and MAC are transmitted to the receiver. The receiver generates its own MAC from the received document and the shared key, and then verifies that it matches the transmitted MAC.

A traditional MAC must be regenerated using the shared key whenever the document is changed. Our system, however, allows an untrusted machine to update a document, and this machine does not have access to the key. Furthermore, we would like the amount of computation done by the machine to be proportional to the size of the delta, not the size of the document.

To meet these requirements, we use what we call an *integrity code (IC)* that is similar to a MAC. Like a MAC, a shared key is needed to generate an IC and to verify it. Each document and each delta contains an IC, produced using the shared key. The ICs are designed such that they can be combined when applying a delta, without knowledge of the shared key. The resulting IC matches what would be produced by computing an IC by a pass over the entire updated document, allowing verification at a later time.

4.4.1 Integrity code generation for a complete document

To compute the *integrity code (IC)* for a document, we compute the IC for each line of the document separately and XOR these together to produce the IC for the overall document. As a special case, if the document has no lines then its IC is 0.

To compute the IC for a line, we decrypt the line using our shared key and then apply a one-way hash function, H (). For instance, to compute the IC for this line in the encrypted document:

```
[558] E (95347892, 109, <color[558]>White</color>)
```

We compute

```
IC = H ([558] 95347892, 109, <color[558]>White</color>)
```

We do not require the use of a specific hash function. All we require is that it be a one-way function (that is, no H^{-1} exists such that $H^{-1}(H(x)) = x$) and that the function be largely collision-free (that is, H(x) is unlikely to equal H(y)). In practice, MD5 or a member of the SHA family will work fine.

4.4.2 Verification of integrity codes

It is simple for a trusted machine to verify the IC of a document. It simply recomputes the IC of the document and compares it with the IC attached to the document. If they match, the document is valid. A mismatch indicates improper processing.

4.4.3 Integrity code generation for incremental changes

In order to support incremental changes to documents, we need to allow the incremental update of integrity codes. We do this by attaching to each delta, or set of update commands, an *incremental integrity code (IIC)* that can be applied to a document's IC to reflect the updates. Neither generating nor applying the IIC should require access to the entire document, nor should it involve re-generation of the IC of the entire document.

In our method, the bitwise XOR function is used to apply the IIC. That is, when the delta has been applied to the document by an untrusted machine, that machine may update the integrity code for the entire document by XORing the document's IC with the delta's IIC, producing a new IC for the document.

To compute the IIC for a delta, we compute an IIC for each line in the delta and XOR them together, producing the IIC for the overall delta. Deltas contain a list of commands, and we generate an IIC for each one depending upon its type:

- **ADD *id*, E(*txt*)**
  This adds a new node to the XML document. The IIC for this change is just H (*txt*). *Txt* is the clear text for a node. As described previously, this contains a random number, the ID of the parent and children, and the text data for the node. When the untrusted machine adds this line to the document and XORs in this IIC, the new IC will match the updated document.

- **DELETE *id***
  This deletes the specified node from the document. The IIC for this is simply the IC of the old node. To compute this, we find the node

to be deleted in the existing file and compute the IC for that line. When the untrusted machine deletes this line from the document and XORs in this IIC, the new IC will match the updated document. This is because the IIC, when XORed, effectively "backs out" the old node's contribution to the overall IC. We are relying on the property that if x = a XOR b, then we can "back out" b with another XOR, as in a = x XOR b.

- **REPLACE *id*, H(E(*txt_old*)), E(*txt*)**
  The specified node is replaced with the specified data. The ID of the node is not changed. We compute the IIC as if this was a DELETE command followed by an ADD command, XORing those two IICs together. In other words, the IIC consists of the XOR of two values: the IC of the line to be deleted, and the IC of the new line.

Once the IICs for each line of the delta have been computed, we XOR them together to produce the IIC for the overall delta. The individual line IICs are not saved. The overall IIC is attached to the delta and sent to the machine that will process the delta. When the untrusted machine performs the update, it executes each line in the delta and then XORs the IIC of the delta with the IC of the document, producing a new IC that matches the contents of the updated document.

4.5 Data freshness and conflicts

Since we allow multiple applications to make incremental modifications to XML documents we have to address the issue of data freshness. Our primary goal is to prevent applications from updating data based on stale copies. In systems that allow multiple applications to make changes to a document the granularity of freshness is often the entire document. In our case the granularity is at the level of a node. This finer level of granularity allows us to support changes to documents which would normally be considered stale. For example if two applications (A and B) cache the same document and if application "A" makes a change and updates it, the copy held by application would be considered stale in the traditional sense. In our case it is possible for application "B" to make changes to the document as long as these changes do not conflict with the changes made by application A. Applications in our system are free to make changes and any conflicting changes would be recognized by the update processors.

Conflicts are when a set of changes made to a document do not agree with another set of changes made to the same document. Conflicts can be one of the following two types, *real* or *false*. For example if a document has an attribute called color and if Bob changes it to green, while at the same time Jim changes it Red, the system will not be able to decide which of the changes to accept. This is a real conflict. The other type of conflict is false; in this case the two sets of changes may appear to be inconsistent because of the granularity of the encoding. However a system that understands the semantics of the document may be able to automatically resolve this type of conflict. For example if we have a node P that has two children b and c. If Bob inserts a new child before child b and Jim inserts another child after c, both changes may be acceptable, however our particular encoding method would view this as a conflict in the children of node P.

In any system that supports incremental updates by multiple applications, conflict can occur and is not specific to encrypted documents. An example of a system that supports incremental updates is a source code control system used by a group of developers. Such systems need to detect when conflicts occur and resolve them. The systems that understand the semantics of the data deal with false conflicts automatically and resort to policies or human intervention for resolving real conflicts. In our case since the update processors deal with encrypted data they will not be able to distinguish between real and false conflicts.

In this paper we are primarily concerned with detecting conflicts. The procedure for resolving conflicts is beyond the scope of this paper, but would most likely enlist application-specific interpretation of the data to enable a meaningful conflict resolution.

4.5.1 Conflicts from multiple applications

Several applications may need to access and update the data in an XML document, which results in multiple deltas being sent to the update processor. This may lead to conflicts. It would be possible to avoid conflicts by restricting data access to one application at a time. However this would be an impractical policy. Typically applications tend to operate on different parts of the data and their updates are non-overlapping. We offer a solution that is optimized for this common case but we handle the exception situation.

4.5.2 Detecting conflicts

When applications make updates to documents, they generate deltas which are sent to the update processor, which is responsible for applying the deltas. It detects a conflict if a delta replaces a node that has been replaced by a previous delta. Add and delete may also cause conflicts, but these will be caught by the associated replace command. Recall a replace command includes a hash of the node's original encrypted data, plus the encrypted version of the modified data. When processing a replace command, the update processor finds the node data associated with the node to be replaced and compares its hash with the hash sent as part of replace and, if they match, replaces it with the new data. If there is a mismatch it marks it as a conflict and appends the delta to the document. The update processor, given a delta, computes if there are any conflicts before applying the delta. A single conflict will disable the merge of this delta, and instead the delta is simply appended to the end of the document.

5. OTHER WORK

There is considerable body of work that is related to our proposal. Most of this work is complementary to what we are proposing. Our solution encompasses the areas of XML encryption, incremental change support for XML, incremental cryptography and XOR based MACs. We give a brief overview of some of the work in these areas.

5.1 XML Encryption

Since XML documents are widely used to store important data, security is a major concern. Hirsch [6] gives a broad overview of the issues associated with XML security. Current work in this area includes two developing standards, XML-Encryption [9] [1] and XML-Signature [10] [1].

XML-Encryption and XML-Signature are going through the standardization process as part of W3C. In brief, the XML-Encryption standard aims to specify a process for encrypting data and representing the result in XML. The XML-Encryption supports encryption of arbitrary data, an XML element, or an XML element's content. The specification defines how the encrypted data is to be represented and does not specify what to encrypt. On the other hand our solution specifies how an XML document is encrypted and also how incremental updates can be performed. The syntax proposed by the XML-Encryption can be adopted to represent our encryption scheme.

XML-Signature provides integrity, message authentication and signer authentication services for any type of data. XML Signature is a method of associating a key with a block of data, and representing it as XML. While this specification is an important component of secure XML applications, it is not sufficient to address all application security and incremental update issues.

Our solution generates MACs that are used to verify the integrity of the document and also support incremental updates to the document as well as the MAC itself. Since XML-Signatures map one XML document into another, they have no effect on our processing of documents.

5.2 Incremental change support for XML

Increasingly, distributed systems are being built to allow multiple users to update to a single data set simultaneously [7]. The key issue in supporting this is the ability to merge changes as they are made. The proposal made by Fontaine [7] aims to support merging of updates at a much finer level of granularity (attributes) than our solution of merging at the element level. However, if attribute-level of merging is required, our method could be adapted to provide it. The motivation behind our decision to support a coarser level of granularity is based on the assumption that applications may operate on a single data set simultaneously but they tend to be non-overlapping. Our solution works well under this assumption. Another difference of our work is that it operates on encrypted versions of the data rather than the clear text.

The key issue in a merge process is the node matching algorithm. Fontaine et al [7] proposes a tree matching algorithm. The algorithm walks through the corresponding nodes in each input XML tree and performs a tree structured comparison. This tends to be time consuming and requires work not proportional to the amount of change. In our model, identification is a trivial algorithm that involves a simple look up of the node's unique ID. In our system a node is assigned a unique ID and that is associated with the node for the node's lifetime.

The fact that there are several commercial entities developing tools to support concurrent changes to XML data validates the importance of this field. Since there is no clear solution that addresses incremental support combined with security, this is a promising and relevant field of study,

5.3 Incremental Cryptography

Using cryptographic algorithms to protect the security of data is a well known technique, however the algorithms used tend to operate on the entire data and any time the data is modified the entire document would have to be re-encrypted. In systems that support incremental changes and simultaneous updates, this technique would not be practical. The goal of incremental cryptography is to design cryptographic algorithms with the property that having applied the algorithm to a document, it is possible to quickly update the result of the algorithm for a modified document, rather than having to re-compute it from scratch. Incremental cryptography can provide large efficiency gains when small changes are frequently made to large documents [2].

Our solution to support incremental changes to XML documents and securing them lends itself very well to incremental cryptographic techniques. Incremental cryptography and its application to virus protection were proposed by Bellare et al [3] and they base their techniques on previous work on XOR MACs [4]. Our solution proposes a technique for incrementally encrypting changes specific to XML documents and more generally to hierarchical oriented or record oriented data structures. The Bellare proposal is aimed at supporting incremental changes to documents by breaking them up into smaller units and encrypting them. It does not specify the exact process by which a document is broken down and what happens if the change crosses the unit boundary. However they highlight the same issues we address. We came to the same conclusion that they have reached in that we cannot hope to support incremental encryption and at the same time hide the amount of difference between the two documents. Our solution also supports concurrent changes and merging of these changes on untrusted servers.

5.4 XOR MACS

The XOR MAC scheme was also originally described Bellare et al [4]. This scheme was designed to support incremental changes, however these changes were limited to just text replacement operations. This XOR scheme was subsequently enhanced to support add and delete operations [3]. The enhancement included the introduction of the chaining technique. The algorithm involves three steps. 1) Computing a hash for each subsection of the document. 2) Combining adjacent hash values using another hash, and then accumulating these results with the XOR function. Combining adjacent values ensures that the order of the subsection is reflected in the final MAC. 3) Generating the MAC of the document by hashing the result of the XOR accumulation.

Incremental updates including deletions and additions involve computing the hashes of the subsections being added or deleted, and adjusting the accumulated value to reflect these additions or deletions. Finally the accumulated value is hashed to produce the MAC.

Our initial MAC computation is similar to this scheme, however the definition of subsections is not defined, and they do not specify how changes across subsections would be handled. In our model, the blocks for which we compute the integrity code are based on the structure of the document. Computing incremental MACs in our proposal is fairly straight forward. We do not need the chaining since our encoding of the document is immune to the reordering of subsections. Also, in the above mentioned scheme it is assumed that the final document's MAC can be computed only on trusted machines and is assumed not to contain conflicts. It therefore cannot be used to support merging of changes on untrusted machines.

5.5 Threats

The proposed system could be subjected to several kinds of threats. We have grouped these into three main categories. We will briefly identify the threats and address some of the possible solutions

5.5.1 Encoding related

Since we encode the XML document on a per node basis any party that has access to the encrypted document could easily infer the number of nodes in the document. If this is information is deemed important it is possible to add additional dummy nodes which can be easily filtered out during document decryption phase.

The choice of the encryption algorithm is not specified and the security of the document depends on the specific encryption scheme. Users have the flexibility in picking the appropriate encryption scheme to ensure the level of security.

5.5.2 Update related

As incremental updates are supported through command lists, it is possible for an adversary to reconstruct partial hierarchies by monitoring the command lists. This will reveal the structure of the document but not the contents of the document. One way to deal with this would be to generate additional commands which would make the reconstruction difficult but not impossible. This threat can be prevented by using a secure channel to transfer the command lists.

Unauthorized parties can generate false updates which would be accepted by the un-trusted server and this may result in preventing real trusted updates. This technique can be used to launch a denial of service attack. Again a secure channel can prevent this threat. This threat does not affect the integrity of the document as changes to documents are accepted only after the changes have been validated.

5.5.3 Standard network related

The common network related threats can be launched and most of these will lead to some sort of denial of service. None of these attacks can be used to make a legitimate modification to the document. If the update command lists are sent over public networks, they can intercepted and modified, however the integrity code checking would catch any such violation. Hosts in the middle can do traffic analysis, determine the size and frequency of changes, perhaps compute the size of the document etc. They may even alter the commands list to remove some valid changes. We offer no specific solutions to these threats. If these threats are considered important, they can be prevented by using a secure channel to transfer the command lists.

6. STATUS AND FUTURE WORK

We have implemented most of these algorithms in support of an application we are building, but the conflict detection and resolution parts are not yet finished.

We are investigating the application of this technology to other (non-XML) representations. In particular, we believe a variant of our method could be used to do similar operations on directed graphs, which are more general than trees.

In order to avoid the need for trusted update generators to have access to the entire document, we are working on methods of retrieving subtrees of documents and computing integrity codes on them. This would allow even more general incremental operation.

We are experimenting with different granularities of encodings, in order to support conflicts better. In particular, we are experimenting with refining our encoding so it doesn't encode a single node's data as one encoded node. Rather, each attribute (name-value pair) and each block of free-form text would be encoded separately. This should reduce the number of false conflicts.

More ambitiously, we are investigating the ability to do queries on encrypted data.

7. CONCLUSIONS

We have applied past work and created new algorithms to solve an important distributed systems problem: the incremental modification of data on untrusted machines. We have structured our system to work on the popular XML format, although the underlying techniques apply to other representations.

The problem solved has wide applicability to distributed systems. Our model of an untrusted storage server is just one example. There are other patterns of data flow in other distributed systems that could benefit from our algorithms.

8. REFERENCES

[1] IBM Corporation. alphaWorks XML Security Suite, *As described at http://www.alphaworks.ibm.com/tech/xmlsecuritysuite, December*, 2003.

[2] M. Bellare, O. Goldreich, S. Goldwasser, Incremental Cryptography: The case of Hashing and Signing – Crypto 94 Proceedings, Vol 839, Springer-Verlag.

[3] M. Bellare, O. Goldreich, S. Goldwasser. Incremental Cryptography and Application to Virus Protection, Proceedings of the 27th ACM Symposium on the Theory of Computing, May 1995.

[4] M. Bellare, R. Guerin, P. Rogaway. XOR MACs: New Methods for message authentication Using Finite Pseudorandom Functions, Oct 1995.

[5] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler. *Extensible Markup Language (XML) 1.0 (Second Edition)* , W3C Recommendation, October 6, 2000.

[6] Fredrick J. Hirsch. Getting Started With XML Security, *As described at http://www.fjhirsch.com/xml/xmlsec/starting-xml-security.html* .

[7] Robin L. Fontaine. Merging XML files: a new approach providing intelligent merge of XML data sets, *As described at http://www.deltaxml.com/pdf/merging-xml-files.pdf* .

[8] Andrew S. Tanenbaum, Maarten van Steen, Distributed Systems: Principles and Paradigms, Prentice Hall, 2002.

[9] World Wide Web Consortium. XML Encryption Syntax and Processing. *As described at http://www.w3.org/TR/xmlenc-core/*, December 10, 2002.

[10] World Wide Web Consortium. XML-Signature Syntax and Processing. *As described at http://www.w3.org/TR/xmldsig-core/*, February 12, 2002.