Ensuring Secure and Robust Web Service Interactions

Ph.D. Thesis submitted by

Gyan Prakash Tiwary

Discipline of Computer Science and Engineering

INDIAN INSTITUTE OF TECHNOLOGY INDORE

August 2021

Ensuring Secure and Robust Web Service Interactions

Thesis submitted by

Gyan Prakash Tiwary

under the guidance of

Dr. Abhishek Srivastava

in partial fulfilment of the requirements

for the award of the degree of

Doctor of Philosophy

Discipline of Computer Science and Engineering

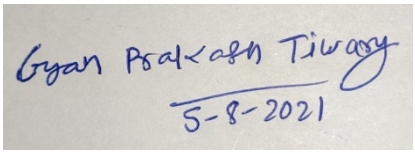
INDIAN INSTITUTE OF TECHNOLOGY INDORE

August 2021

INDIAN INSTITUTE OF TECHNOLOGY INDORE

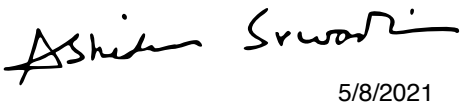
I hereby certify that the work which is being presented in the thesis entitled **Ensuring Secure and Robust Web Service Interactions** in the partial fulfillment of the requirements for the award of the degree of **DOCTOR OF PHILOSOPHY** and submitted in the **DISCIPLINE OF COMPUTER SCIENCE AND ENGINEERING, Indian Institute of Technology Indore**, is an authentic record of my own work carried out during the time period from February 2015 to August 2021 under the supervision of Dr. Abhishek Srivastava, Associate Professor, Indian Institute of Technology Indore, India.

The matter presented in this thesis has not been submitted by me for the award of any other degree of this or any other institute.



signature of the student with date
(Gyan Prakash Tiwary)

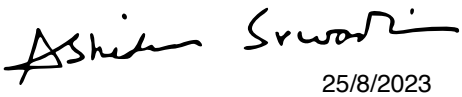
This is to certify that the above statement made by the candidate is correct to the best of my/our knowledge.



5/8/2021

Signature of Thesis Supervisor with date
(Prof. Abhishek Srivastava)

Gyan Prakash Tiwary has successfully given his/her Ph.D. Oral Examination held on **August 25, 2023**.



25/8/2023

Signature of Thesis Supervisor with date
(Prof. Abhishek Srivastava)

ACKNOWLEDGEMENTS

I am very grateful to my supervisor Dr. Abhishek Srivastava for his invaluable guidance, encouragement, and direction throughout this work. Working with him ultimately resulted into a great deal of enjoyment in my dissertation research.

I would like to express my heartfelt gratitude towards my PSPC committee members Dr. Neminath Hubballi and Dr. Vivek Kanhangad for their interesting discussions and suggestions towards my research.

I would also like to thank my parent, wife and family member for their continuous support during the tough phases of my Ph.D. Their suggestions and their words motivated me to continue the hard work during the course of this thesis.

I want to thank everyone who have, in one way or another, helped me to conduct this research. I express my appreciation and indebtedness to my friends Dr. Dheeraj Rane, Dr. Rohit Verma, Dr. Tanveer Ahmad and all who helped me in many ways during my thesis work.

Abstract

Web services are a programmable way to access the web. Unlike websites, web services do not return human-friendly content such as HTML, CSS, or JavaScript in response to a request. Web services mostly support the transfer of data in XML or JSON formats. Owing to this, the efficacy and security of XML and JSON documents is important. XML and JSON documents are intended for consumption by computer application and not humans. The main issue with XML and JSON documents is that they are verbose. They are both structured documents, meaning that their content can be divided into structure and data parts. In XML, the tag-name and attribute-name constitute the structured parts of the document whereas values between the tags and values of attributes are the data parts. JSON documents comprise structure parts as names whereas values are data part. Compressing these documents increases the communication efficiency between web services. We review several prominent techniques of XML and JSON compression. The idea behind XML and JSON compression techniques is to compress by appropriately harnessing their structured nature instead of just compressing them like normal text. Subsequently, we propose an XML and JSON compression technique that performs better than existing techniques in the terms of compression ratio, especially in the case of web services. The method of XML and JSON compression proposed is simple yet effective, especially on small documents that constitute the bulk of such communicated content on the Internet.

In addition to efficacy in XML or JSON compression, we also need to ensure security of such documents. There exists a W3C recommendation for XML encryption. The W3C recommendation prescribes the use of conventional encryption algorithms to encrypt such content and present it using special XML elements. The idea is to encrypt the entire XML document or parts of it with existing encryption algorithms and represent the encrypted data under special XML tags. W3C's recommendation on XML encryption has been perhaps the most prominent work in this direction until now. We propose a technique of XML and JSON encryption wherein we extend existing techniques of XML compression and employ the same for security.

Owing to this, the proposed approach meets all the standards for security and in addition to this compresses the document as well.

XML and JSON encryption makes a document secure, but encrypting a document in certain scenarios is not enough. One such scenario is that of data service composition. It is common nowadays for data service providers to provision their database services through web service interface. Such data service providers accept queries to execute through these interfaces and communicate the responses also in the form of web service messages. Often a data service provider cannot on its own fulfil a client's request. In such cases, several data service providers compose their services and address the queries of the client. This is called data service composition. In a data service composition, a query to a service provider can be based on the output of another service provider. When the output of one data service provider is an input to another data service provider, the receiving provider invariably has access to substantial information on the former. Hence. the privacy of data service providers in a composition is often compromised and is an important issue. A mediator based data service composition approach and value generalization achieve privacy to an extent in such compositions. The major issue with such an approach is that mediators themselves are often untrustworthy and their intervention leads to compromise of the composition.We propose an approach to data service composition that is able to address security and privacy issues in data service composition whilst working with an untrustworthy mediator.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

Two totally different worlds of the World Wide Web exist on the Internet. A world of the World Wide Web in which content such as HTML, CSS and Javascript is shared that are called websites. The focus of content shared on websites is largely on the presentation of data. The content of websites is meant to be consumed by humans. A parallel world of the World Wide Web exists in which the focus is on data and not its presentation. This world of the World Wide Web is called Web Services. The content of web services mostly comprises XML or JSON which is meant for consumption by computer programs. The importance of web services today has greatly increased the share of XML and JSON in the content of the Internet.

The increase of XML and JSON content on the Internet has made the efficacy and security of these documents an important topic of research. Examples of XML and JSON are shown in listings 1.1 and 1.2 where the two represent the same content. XML and JSON are structured documents with their content being in tag-value or name-value pairs. The main problem with XML and JSON documents is that they are verbose in nature. The size of the XML shown in

Listing 1.1: API Response in XML

```
<Airport City="Edmonton">
  <Arrival-time>16:02:30</Arrival-time>
  <Departure-time>16:22:30</Departure-time>
  <Date>2019-12-24</Date>
  <Gate>6</Gate>
  <Gate>7</Gate>
</Airport>
```

Listing 1.2: API response as a JSON

```
{ "Airport":
  { "Arrival-time": "16:02:30",
    "Departure-time": "16:22:30",
    "Date": "2019-12-24",
    "Gate": "6",
    "Gate": "7",
    "_City": "Edmonton", }
}
```

Listing 1.1 is 169 bytes and the size of the JSON shown in Listing 1.2 is 154 bytes. The size of the actual content that both documents are representing is only 55 bytes. Owing to this, XML and JSON documents often need to be compressed.

XML and JSON documents are text files and can be compressed with regular text compressors like bzip2 [52] and ppm [19]. Such compressors ignore the structured nature of these documents. However if the structured nature of XML and JSON is taken into account during compression, there is potential for superior compression. The parts of XML and JSON documents that do not normally change, such as tags in XML, attribute-names, and names in JSON, are called structure parts of the document. Whereas the values between the tags in XML, the values of the attributes and the values in the JSON are called data parts and these change frequently. Structure parts define the structure of a document whereas the data parts contain the actual information. In the example shown in Listings 1.1 and 1.2 “Edmonton”, “16:02:30”,

"16:22:30", "2019-12-24", "6" and "7" constitute the data part, i.e., the actual information to be communicated. The element and attributes tags, i.e., "Airport", "City", "Arrival-time", "Departure-time", "Date" and "Gate", constitute the structure of the document. Compression techniques that take advantage of the structured nature of XML are called XML-conscious compressors [49]. It is important to note that the majority of compression research is done on XML and not JSON. However, both documents can be separated into structure and data parts in much the same way, every XML compression method is also equally effective on JSON as well. In this thesis also, we will mostly discuss our work in terms of XML but the same would apply equally well to JSON.

Almost all existing XML-conscious compressors use a dictionary in which the structure part of the document is mapped to a unique number of the least possible digits. All XML-conscious compressors, on the other hand, treat the data part of the XML differently, and obtain different compression ratios for different types of XML. XML-concious compressors are classified into two groups based on whether or not they produce queryable compressed documents. Queryable XML compressors maintain the structure and order of the original document in the compressed form (partially), allowing retreival of data without having to decompress the entire document. Queryable XML compressors are quite useful in XML-storage systems and allow space-saving storage and quick access. XGrind [59] is the best-known queryable XML compressor. XMill [37] is a well known non-queryable compressor.

Aggregation and clustering of XML documents is another approach to XML compression, in which a large number of small XML documents are combined to make a bigger one, which is then compressed in an XML-conscious manner. SMCA [26] is a state-of-the-art compression technique that takes this approach. SMCA is one of the latest works in XML compression. The best know XML technique and often regarded as the most important benchmark in compression is XMill. Chapter 2 delves into the taxonomy of XML compressors and associated studies.

In this thesis, we introduce a new XML compression technique that is especially useful in reducing the size of XML or JSON documents transmitted as requests and response across web APIs. The work is driven by the requirement to enable bandwidth-efficient interactions within service-oriented systems. The size of these documents is often small (less than 1MB), and the content has a large proportion of structured content (element and attribute tags) to data content.

In addition to compression of XML and JSON documents during web services interac-

tion, another important aspect is security. Encrypting an XML document is a step towards securing it and XML can be encrypted using existing block cipher or stream cipher. The W3C-recommended XML encryption approach [30] is exactly this and encrypts XML documents using existing block or stream ciphers (e.g. [31] and [20]). Everything in the XML document is encrypted and subsequently the encryption is encoded with specific XML tags. This approach necessitates XML canonicalization [17] in order to maintain namespace prefix bindings and attribute values in the XML namespace. "Canonicalization [17] of XML comprises consistently serialising XML into an octet stream as is required prior to encrypting XML," according to the W3C.

In this thesis, we propose a new XML and JSON encryption technique by modifying the well-known XML compression technique, XMill [37]. While XMill is primarily concerned with XML compression, it can be tweaked to work with JSON as well. The proposed encryption technique works with both XML and JSON. It builds upon and extends our work on compression that we discuss in this thesis.Such encryption approaches that are built upon a compression techniques don't just safeguard the documents, but also appropriately compress them. The proposed method of encryption is especially meant for structured documents that can easily be separated into structure and data sections like XML and JSON. The structured nature of XML or JSON is harnessed effectively for superior encryption. XML canonicalization (or serialisation) also play an important part in XML encryption and introduces a degree of randomization that aids in the reduction of size of the final document.

The security and compression of XML and JSON documents ensure effective and secure interactions between web-services. In spite of this, one category of web-services that remains vulnerable are data services. These are web-services that provide data over technology agnostic interfaces. Secure interactions between data services is a challenge owing to compromised privacy of data exchanged between such services. In this thesis, we also propose an approach to ensure secure and privacy data service compositions.

There are several types of data service providers in the world: medical databases, retail databases, demography databases, and a variety of others. The data is accessed from these and used by clients based on their needs. The web-service based interfaces provided by data service provides makes it much easier for clients to access databases. Furthermore, it allows a service provider, unable to offer all of the data requested, to seamlessly access databases

of other service providers and respond to a client's request. The phenomenon known as data service composition occurs to cater to queries that require data from multiple sources over a single interface.

A data service composition plan is created when a query requires data from more than one data service provider. Each query has its own strategy for service composition. As depicted in Figure 1.1, a service composition plan is a directed acyclic network in which each node contributes to the response to the query for which it was created [55]. If the composition plan has a directed edge e_{ij} from a service provider *DSi* to *DSj*, then *DSi* is referred to as the parent of *DSj*. This means that in a query response, *DSi* comes before *DSj*. *DSi* executes first, and subsequently *DSj* executes. For example, in Figure 1.1, *DS1* is the parent of *DS2* and *DS3*.

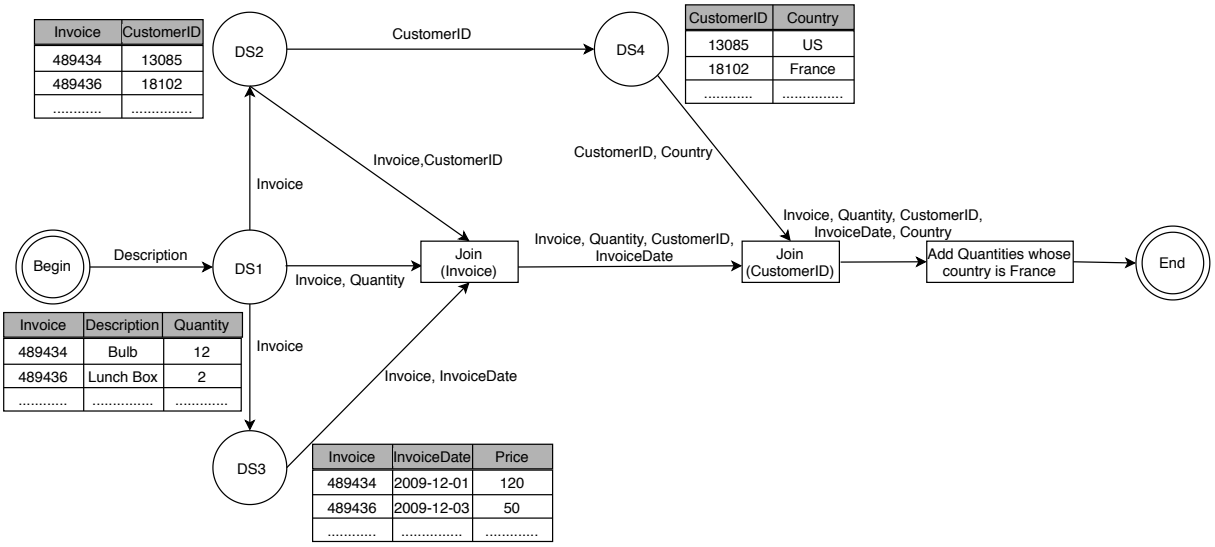


Figure 1.1: Service Composition Plan (Data flow between two services always happens through mediator)

The major problem with data service composition, as mentioned earlier, is ensuring privacy of component services. In Figure 1.1, suppose a query is performed on DS1; the result from DS1 is then used as the input to execute a query on DS2. When the result from DS1 is used as the input to execute a query on DS2, DS2 becomes aware of the data from DS1. In certain situations, however, DS1 would not want its data to be shared with DS2 or anyone else. If data is not shared then data service composition becomes impossible. To overcome this, a mediator based solution is proposed [45]. To ensure the privacy of DS1 in the earlier scenario, the mediator first performs value generalization of the data that is sent to DS2. Value general-

ization implies executing a query for K inputs on DS2 in place of just the one relevant query that has data from DS1. This is done so that DS2 does not readily come to know about DS1's data. The probability of DS2 knowing about DS1's data becomes 1/K. K-Anonymity [56], and/or its variants such as L-Diversity [39] and T-Closeness [35] are techniques that facilitate value generalization.

Value generalization solves only a part of the problem. A mediator cannot necessarily be trusted and therefore before sharing data with a mediator, it is secured in a manner that the mediator cannot see it but can compare it. Further, an unethical mediator can tamper with a service composition plan and wrongly include an alien service provider into the service composition plan. To overcome this, each parent-child pair in a service composition plan is required to authenticate each other's messages. We propose an approach for data service composition that carefully considers each of these issues and addresses them effectively.

1.2 Research objectives

- To propose an XML compression technique aimed at reducing the size of XML documents transmitted as requests and response between web APIs, motivated primarily by the requirement to enable bandwidth-efficient communications within service-oriented systems.

- To develop an XML and JSON encryption technique that is secure as well as keeps the encrypted document as small as possible. We create encryption techniques for XML and JSON documents that extend the ideas of existing XML compression techniques.

- To develop a mediator based technique for data service composition that maintains the privacy of the component service providers. The proposed method ensures that an unethical mediator cannot change the shared service composition plan or introduce an alien service provider without the permission of the other legally participating service providers.

1.3 Thesis outline

The subject matter of the thesis is presented in the following five chapters,

- ✓ Chapter 1 provides an overview of the compression and encryption approaches proposed for XML and JSON. The chapter also provides an overview of provisioning security and privacy to an important and rather vulnerable category of web-services, namely data services.

- ✓ Chapter 2 discusses existing literature on on compression and encryption aspects of XML documents. Most existing research endeavours are devoted to encryption and compression of XML and these ideas are in most cases equally valid for JSON.

- ✓ Chapter 3 describes the proposed method of XML and JSON compression in detail. This chapter also presents a comparative study of the proposed method with respect to existing methods. This comparative study clearly shows that the proposed method outperforms existing techniques in terms of compression ratio.

- ✓ Chapter 4 discusses encryption techniques proposed for XML and JSON documents which are mostly extensions compression techniques discussed in Chapter 3. The chapter also includes details on experiments performed to demonstrate the efficacy of the proposed method in securing against various XML based and generic attacks.

- ✓ Chapter 5 discusses dwells upon perhaps the most vulnerable web-service offerings, data services. Specifically, the chapter works on the data privacy aspects in data service composition. A mediator based approach is taken to maintain privacy between service providers and the assumption is that the is untrustworthy.

- ✓ Chapter 6 concludes the thesis and includes important pointers to future endeavours in this direction of research.

This page was intentionally left blank.

Chapter 2

Review of Literature

In this Chapter, we extensively discuss existing literature on the three main components of the thesis i.e. XML and JSON compression, XML and JSON encryption, and privacy preserving data service composition. Literature on the three components is discussed in separate sections for better clarity and comprehension.

2.1 Literature Survey on Compression of XML and JSON docuemnts

Most prominent research endeavours in compression of structured documents is XML with very little work done on JSON. It should be noted, however, that most of work on XML is equally effective on JSON as well. In this thesis, we develop a compression technique that works on both XML and JSON documents. Some significant contributions on XML compression are discussed in the succeeding paragraphs.

In principle, compression techniques belong in one of two categories: *general-text com-*

pressors and *XML-conscious compressors*. XML is a text file, and like any other text file, it can be compressed by any general text compressor such as bzip2 [52], ppm [19]. A general text compressor, however, does not take advantage of the special features of XML and, therefore, tends to result in inferior compression ratio. XML-conscious compressors, on the other hand, utilise the structure of XML documents and therefore are able to compress XML documents with better compression ratios. XMill, XGrind and SMCA [26] are examples of XML-conscious compressors.

XML-conscious compressors are next divided into three categories: *schema-dependent*, *schema-independent* and *XML tree-based* compressors. Schema-dependent compressors require the schema of the XML documents to effectively compress a document. rngzip and [25] are examples of schema-dependent compressors. Schema-independent compressors, on the other hand, do not require the schema. XMill, XGrind and SMCA are examples of schema-independent compressors. Finally, XML tree-based compressors take advantage of the fact that XML documents are organized in a tree-like structure and aim to make this tree smaller. This is done by representing the XML trees as a DAG (Directed Acyclic Graph) or a FCNS (First Child Next Sibling) list: [18], [38] and [16] are examples of XML tree-based compressors.

Furthermore, based on whether or not the resulting compressed documents support queries such as XQuery, compressors may be categorized as *queryable XML compressors* and *Non-Queryable XML Compressors*. XGrind is a queryable compressor while XMill and SMCA are non-queryable compressors. Queryable compressors can be *homomorphic* and *non-homomorphic*. The former type retain the original structure of the XML even after compression, whereas non-homomorphic compressors do not. XGrind is a homomorphic compressor while XMill and SMCA are non-homomorphic compressors. The XML compressor proposed in this thesis is XML-conscious, schema independent, queryable, and homomorphic.

Much of the research done so far on XML compressors is around XMill. XMill separates the data and structure parts of the XML and inserts the data values into different containers based on the path and data type. A dictionary is created to assign a unique number to every unique element and attribute name. Each container is then compressed using a back-end compressor specialized for that data type. Most general-purpose compressors can be used as back-end compressors. XGrind is a two-pass compressor: the output of XGrind is structured along the lines of its input XML. Querying or parsing thus becomes possible even on the compressed

documents. XGrind encodes the element names and attributes by numbers followed by 'T' and 'A' ('T' for tag, 'A' for attribute) and compresses the values using Huffman coding. XML does not lose its basic structure due to compression with Huffman coding, and this feature makes XGrind queryable.

XGrind, like XMill, is a seminal piece of work on XML compression and several research endeavours have built on it. Prominent among these are [36], [41] and [54]. Al-Shammary *et.al* [5] was the first attempt at XML compression keeping in mind SOAP-based web services. In this work, the XML tree is first converted into a binary tree, using a FCNS (First Child Next Sibling) transformation. Subsequently, a sibling state of two bits to each node of the binary tree is attached. Finally a Huffman code is assigned to each node and the entire XML is compressed. The resulting compressed document is non-queryable.

This was followed by an era of aggregation-based research on SOAP messages. This line of research is based on the principle that, if a group of XML documents is compressed instead of compressing each XML message separately, the compression ratio will be better[1], [2], [4], [6], [7], [8], [9] [10]. SMCA [26] is perhaps the most recent major research endeavour in grouping based XML compression, outperforming most approaches of XML compression in terms of compression ratio. It is the only technique that aggregates and compresses multiple XMLs in one pass and generates a single non-queryable output. However, RESTful services are known for single request and response, so aggregating multiple XMLs is not relevant in the case of RESTful services.

Various XML compressors discussed in this section are not as effective at compressing relatively small ($<$1MB) XML documents in terms of compression ratio. XML documents of this size are mostly utilised in API requests and responses. The norm in ReSTful web services is that a single request is followed by a single response. Therefore the technique of grouping responses to several requests together is somewhat impractical. The method we propose in this thesis overcomes the shortcomings of earlier approaches and is especially effective for relatively small XML documents and is therefore ideal for use with API request and responses (especially ReSTful Web services). Further, the proposed technique creates queryable compressed documents.

2.2 Literature Survey on XML and JSON Encryption

By far the most effective documentation on XML encryption is the W3C recommendation [30] that has been talked about in the Chapter 1. According to W3C's recommendation, any part of XML or the entire XML, is encrypted with the Existing Encryption algorithms like AES [31], DES [20] etc. The encrypted data is then represented in the form of XML under special security-tags. Special XML security-tags with the encrypted data inside it can be made an element of plain XML. W3C's recommendation is not just about encrypting XML data but also encrypting any data and representing that encryption in the form of XML with the help of special security-tags.

Another work on XML encryption is "XML Pool Encryption" [24]. In XML pool encryption, each node of XML is encrypted separately. After that those encrypted nodes are kept in a pool. This pool can either be an element of the plain XML, or it can be a separate document.Elements of XML that are encrypted are removed from their place in the plain XML and placed in the pool. While decrypting, the elements present in the pool go into their place in plain XML.

Hashizume *et. al.* [27] have made W3C recommendation as base of their research. They have worked on encrypting different tags of XML with different keys. When there are multiple receivers of the same XML, it is necessary to encrypt different tags of that XML with different keys. XBMRSA [44] is another method of XML encryption. They have also considered W3C recommendation as the base of their work. They have used Multi-Prime RSA [15] technique to achieve better and fast XML encryption. S.C. Seak *et. al.* [50] have also followed the W3C recommendation in their research. They have implemented digital certificate based XML encryption requires Public Key Infrastructure (PKI) technology [28] to secure symmetric key that is used to encrypt the data.

There is a special standard for the encryption of JSON documents called "JSON web encryption (jwe)" [32]. Encryption standard of JSON also requires serialization. There are two types of serialization used by JWE, 1) JWE Compact serialization and 2) JWE JSON Serialization. JWE also depends on existing encryption algorithms. JWE defines 19 steps to perform encryption, from determining the key management mode to creating the desired serialized out-

put. If there is no dependency between input and output of the steps, then the order of steps does not matter.

2.3 Literature Survey on Privacy Preserving Data Service Composition

System of data service composition have always been a major topic of research [47]. Achieving the privacy of data sources has always been a challenging task in this area [12]. As discussed in Chapter 1 a mediator based solution is a better approach for data service composition because a mediator takes many responsibilities of data service composition itself. In a research proposed by Yau e. al.[61], the mediator acts as a data repository. The data useful for the query is collected by the mediator from different data service providers in hashed format. The collection of these data in hashed format prevents data from being reused in another query. However, this technique does not prevent information leaks between different data service providers.

S. E. Tbahriti et. al. [57] have coined terminologies like “Privacy Policy" and “Privacy Requirement". What operations can a data service provider perform on its own data, this policy is called the privacy policy of that data service provider. Whereas the privacy requirement of a data service provider implies how it wants others to use its data. Through privacy requirements, data service providers applies their the right to conceal their data (i.e., output). The two data service providers will integrate with each other only when the privacy requirement of the source data service provider is compatible with the privacy policy of the destination data service provider. A formal privacy model is defined to allow a service to define a set of privacy policy and a set of privacy requirements. If the source and destination service providers’ privacy requirement and privacy policy are compatible only then their composition happens. Here the mediator sets a compatibility in the privacy requirement of the source and the privacy policy of the destination. If this compatibility is not set, then service composition will also not be possible. In order to set compatibility, the mediator negotiates between source and destination service providers to ease their privacy policy and privacy requirement. While there is no provision of real security (i.e encryption and authentication) between service providers and service consumers, it only works with negotiation. The techniques proposed by [61] and [57] rely on centralized entity that can

be trusted by all the data service providers in the data integration plan.

In research works proposed by B. C. M. Fung et. al. [23] and N. Mohammed et. al. [42] create an atomized view of data using K-Anonymity. However, service provider and service consumer can know each other's data in common which is not acceptable. There are few other researches (i.e [60] and [43]) using K-Anonymity, but they all lack somewhere in efficiency and cost effectiveness.

The research work proposed by M. Barhamgi et. al. [45] is the latest in the field of privacy in data service integration. The mediator is not considered trustworthy in the technique proposed by [45]. Data service providers share their data with the mediator in encrypted form. The mediator uses the data received in encrypted form for the join operation. In this research, Order Preserving Encryption Scheme (i.e OPES) algorithm [3] has been used to encrypt identifier attributes. OPES encrypts numeric data values such that between two encrypted values can be compared without decrypting the values, that which of them is bigger, which is smaller, or both are equal. Authors have used dubbed K-protection mechanisms (introduced by authors) to prevent data leakage between two service providers, which is the improved versions of k-Anonymity and Private Information Retrieval (PIR) [53] mixed together. PIR provides a primitive for accessing outsourced data from a server by a client while preventing a server to learn about client's access pattern. One problem with this research is that only numerical values are encrypted here. Nothing has been said explicitly about what to do if the identifier attributes are not numerical.

As discussed in the Chapter 1, in the all above discussed researches, the data of one service provider is not authenticated by other data service providers in the composition plan. In such situations, the mediator may not follow the shared service composition plan. The mediator can introduce a foreign data service provider in the service composition plan without consent of other service providers in service composition plan. We feel that in the above researches the mediator has been trusted more even considering it to be a non-trusted entity.

Chapter 3

Compression of XML and JSON API Responses

Effective web-service interactions are largely dependent upon compression of XML and JSON documents. Significant literature exists on compression of these documents through generic compression technique. Very little work is meant for XML and JSON documents specifically. In this chapter we propose a technique that harnesses the structure of XML and JSON for more effective compression. We first analyse the problem associated with existing methods of compression and subsequently explain the working of the proposed technique in detail. We finally compare the working of the technique with existing state of the art compression techniques and demonstrate the superiority of the proposed technique.

3.1 Understanding the Problem

To understand the problem statements of the proposed research, it is necessary to understand the workings of XMill [37] and SMCA [26], the most significant XML compression techniques.

SMCA is largely based on XMill and the points below summarize the common features shared by both of them.

- Both techniques treat the data and structure parts of XML differently.
- In both techniques, a ‘tag dictionary’ assigns numbers to the structure parts of the XML based on their position in the document. For example, the tag dictionary corresponding to the XML in Listing 4.1 comprises six tag names, i.e., Airport, City, Arrival-time, Departure-time, Date and Gate, and their respective codes, i.e., 1, 2, 3, 4, 5 and 6.
- In both techniques, a ‘path’ to each XML data (content) point is worked out. The path to a data point in XML is the sequence of tag names from the root tag to the parent tag of the data element in question. For example, the data point “2019-12-24" in Listing 4.1 has the following path: “Airport/Arrival-time". Data points “6" and “7" have the same path i.e., “Airport/Gate". An important assumption of both XMill and SMCA is that data points with the same data path are similar.
- Both XMill and SMCA create a new container for every new path and insert all the data that share a common path a common corresponding container.
- The major premise of both XMill and SMCA is that compressing similar data together in a container with compressors like bgip2 [52], provides better compression.

Assuming that compressing an individual XML documents does not yield a sufficient compression ratio, SMCA works on several similar XML documents, aggregated together. In SMCA, the contents from multiple XML elements with the same path are put together in one container to get better compression. Both XMill and SMCA suffer from some the following drawbacks.

- Certain types of XML documents result in the creation of many containers, each with a small number of data elements, a phenomenon which adversely affects the overall compression ratio. The XML documents used in Web APIs (either SOAP or RESTful) tend to suffer from this problem, since they are typically short and their data elements do not share common paths. XMill and SMCA are, therefore, less effective in compression of short XML documents.

- Assigning numbers to the structured parts of the XML document based on their order in which they appear in the document may result in infrequent structured elements being assigned smaller numbers, i.e., numbers with fewer digits, than more frequent elements. This results in bulkier than necessary path containers, and therefore reduces the effectiveness of the compression process.
- Neither XMill nor SMCA maintain the XML structure in the compressed format, and, therefore, the compressed document is not queryable.

In view of the drawbacks of existing XML compression techniques, we raise the following hypotheses.

- A small number of containers with a large number of similar data elements can result in better compression.
- Better compression can be achieved if the structure tags of the XML document are assigned the smallest possible number (starting from 1) in the tag dictionary based on their probable frequency. More frequent tags should be given small numbers, and vice-versa.

Using these hypothesis, in this thesis, we describe a new XML compression technique motivated primarily by the need to enable bandwidth-efficient communications within service-oriented systems and aims to reduce the size of the XML documents exchanged as requests and responses between web APIs. The size of these documents is typically not very large (generally less than 1MB) and their content typically involves a high ratio of element and attribute tags to data content.

3.2 The Proposed Technique of XML and JSON Compression

In this section, we describe in detail our method for XML compression. The proposed method deals with all the problem statements discussed in Section 3.1. It is designed to examine the hypotheses discussed in Section 3.1. Our method converts the entire XML into a special type

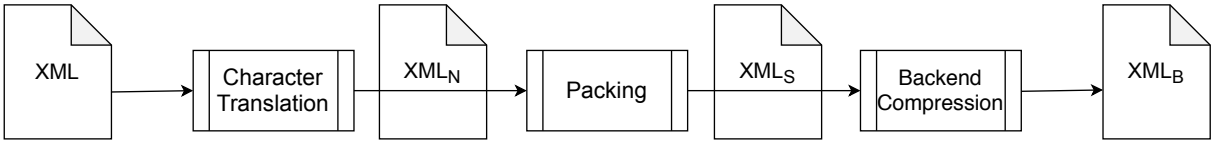


Figure 3.1: The three steps of the compression process.

of binary encoding in a single container. A back-end compressor is then applied to the entire container. The structure parts of XML are assigned with a smallest possible number (starting from 1) in the tag dictionary according to their probable frequency in XML. The tag dictionary is refereed to as “tag table" in this article. The tag table for the XML in Listing 4.1 is shown in Table 3.1.

Note that our method relies on a simple key idea: each element and attribute tag is mapped to a number; all closing tags are also mapped to another single number; finally, each distinct symbol of the language, in which the free-text of the document is written, corresponds to its own number. This translation process applies equally to XML and JSON documents, since the only difference between the two notations is in the lack of pairs of opening/closing tags in JSON.

The method involves a *configuration* and a *compression* phase. The *configuration* phase creates two dictionaries, the *symbol table* (i.e. Table 3.2) and the *tag table* (i.e. Table 3.1), which are used for the *compression* process, shown in Figure 3.1.

The first step of the *compression* process is the *character translation*, wherein the original XML document is converted into XML_N, a form comprising only 13 symbols that can be represented by just 4 bits. In the subsequent *packing* step, every two contiguous symbols of XML_N are accommodated in one byte. This is done for all symbols contained in XML_N and the resulting document is called XML_S. Finally, the third *back-end compression* step involves compression of XML_S using any one of the various back-end compressors like PPM [19] to form the XML_B document.

3.2.1 Configuration

The *configuration* phase creates the *symbol table*, which maps each character present in the data part of the input XML document to a unique number, and the *tag table*, which maps each

Table 3.1: The Tag Table corresponding to the flight-description XML document

Airport	2
Arrival-time	3
City	4
Date	5
Departure-time	6
Gate	1

element and attribute tag-name into a unique number.

According to a recent survey [46], 60.7% content on the web are in English. The second most used language is Russian, which is 8.3% of total and third most used language on the web is Turkish which is just 3.9% of total. In addition, several other languages use the Latin script. We are not entirely sure about the programmable web, but realistically we expect the language distribution of APIs to be along similar lines. Therefore, assuming English as the main language of XML documents, the document characters are in the range of 32 to 127 in the ASCII table. The symbol table, therefore, is a mapping of each character in this range of ASCII values to a unique number. Each character in the symbol table is mapped to a two-digit number, computed as the *(ASCII value of the character - 31)*. Table 3.2 shows a subset of the symbol table. The symbol table is independent of the specific XML document under compression; instead it only depends on the language of the document. Therefore, the same symbol table is used to compress all XML documents in a given language. Given the total number of characters in the English alphabet, commonly used symbols, and the Arabic numerals, the range of the Symbol Table values would be covered by two-digit numbers.

Tag tables depend on the XML schema underlying the to-be-compressed document, which, in turn depends on the web API. In principle, a tag table could be constructed once and reused for all documents emitted by the API. However, to accommodate XML documents that are the result of text annotation and to take advantage of the possibility that a particular response may not use all the tags in the schema, the tag table is constructed by scanning the XML document and mapping every unique element and attribute to a unique number. More frequent tags are mapped first to smaller numbers, whereas tags that appear less frequently are mapped to subsequent larger numbers.

Table 3.2: A subset of the Symbol Table

.	..
E	38
d	69
m	78
o	80
.	..

To illustrate the construction of the tag table, we use the XML shown in listing 4.1. Table 3.1 shows the corresponding tag table. As “Gate" is the most frequently appearing tag in the XML, it is mapped to the smallest number; all tags and attributes are also mapped to unique numbers.

Considering the practicalities of the deployment of our compression method, we propose that service providers share the symbol table and tag table of their XML response documents, along with the documentation of their APIs.

3.2.2 Compression

The actual compression process is carried out in three steps: *character translation*, *packing*, and *back-end compression* as shown in Figure 3.1.

3.2.2.1 Character Translation

In this step, the symbol table is used to translate the data part of the XML document, one character at a time. The structure part of the XML is translated using the tag table, with an entire element or attribute tag substituted by their corresponding number in the tag table. The intermediate file produced by this step is called XML_N. The XML_N corresponding to the XML document in Listing 4.1 is shown in Listing 3.1. In the XML_N ‘T’ denotes the tag name, the number 2 following T is the tag table entry corresponding to the word “Airport” in the XML. Similarly, ‘A’ represents an attribute name and the number following A, 4 in this case, is the tag

Listing 3.1: Example of the XML_N

```
T2 A4 1314150616070616 T3 2526272829273028 0
T6 2526272929273028 0 T5 29282534232529232935
0 T1 23 0 T1 24 0 0
```

table entry corresponding to the word "city" in the XML. Zero in X_N indicates the closing tag for the last (or most recently opened) tag. The remaining numbers that are neither zero nor have an 'A' or 'T' before them represent the data part of the XML. Each character of the data parts of the XML is substituted with the help of the symbol table. In this way, XML_N is made up of only 13 types of symbols, the Element Start Symbol (T), the Attribute Symbol (A), Space, and the digits 0 to 9. Element End symbol is denoted by a single 0 preceded and followed by Space. Therefore no extra symbol is required to denote the end of an element.

The character-translation process is described formally in Algorithm 1. Every token of XML is picked up one by one in different iterations (Line 3). If the chosen token in the current iteration is a data part of the XML (Line 4), each character of that token is translated with the help of the symbol table, and stored in the variable *translate* (Line 5). If the chosen token is a structure part of the XML (Line 6), the entire token is translated to a minimum possible number with the help of the tag table, and stored in the variable *translate* (Line 7). If the chosen token in the current iteration is a tag of the XML (Line 8), the character 'T' is prepended to the current value of *translate* (Line 9). If the chosen token in the current iteration is an attribute-name of the XML (Line 10), the character 'A' is prepended to the current value of *translate* (Line 11). If the chosen token in the current iteration is a closing tag of the XML (Line 12), the variable *translate* is set to 0 (Line 13). After obtaining a proper token it is concatenated with the variable XML_N followed by a space (Line 14). As many iterations are performed as is necessary to translate all the tokens in the XML document, and return the value of XML_N.

3.2.2.2 Packing

As the total number of symbols in XML_N is just 13, the symbols may be represented using 4 bits. In the packing step, every two contiguous symbols of XML_N are accommodated in one byte. Doing this for the entire content of XML_N results a 'squeezed' binary file called XML_S.

As both XML_N and XML_S retain the tree structure of the original XML, they may be parsed with the help of the symbol and tag tables and Xquery queries (or similar) can be executed on them.

Packing is described formally in Algorithm 2. Everything starts with mapping all the 13 different symbols used in XML_N to a unique 4-bit binary (Line 2). A cursor, denoted by the variable *cursor*, points to the first symbol present in XML_N (Line 3). If this cursor is not the end of the XML_N (Line 4) and there exists another symbol next to where the cursor is pointing (Line 5), then the symbol next to the current cursor is pointed to by another cursor denoted by the variable *cursor1* (Line 6). The 4-bit binary mappings of the symbols being pointed by *cursor* and *cursor1* are accommodated in one byte (Line 7). However, if the variable *cursor* is pointing to the last symbol of the XML_N (Line 9), then the byte will be created by assuming the next 4-bit binary as *0000* (Line 10). The created byte is appended to the variable XML_S (Line 8 and 11). Because in an iteration we are picking two values from the XML_N, so for the next iteration the cursor has to be advanced by two symbols. Finally the algorithm returns XML_S.

3.2.2.3 Back-end Compression

The this final step of the compression process, the XML_S created during the packing stage is passed to any one of various back-end compressors. In this work, we choose PPM [19] as the back-end compressor because of its superior compression ratio in our case. Any other general purpose compressor may also be used for back-end compression. The output of this step is called XML_B. Parsing XML_B or executing queries on it is not possible. SMCA, XMill, and many other XML-compression techniques use back-end compressors. We too have used a back-end compressor to show the effectiveness of the proposed technique against existing techniques.

The XML schema is required to create the tag table. However, once this table is created, the entire XML document need not be handled together for effective compression. Different parts of an XML can be compressed separately using the tag and symbol tables. Just one tag name of a large XML document can be substituted using the symbol table and tag table to get the corresponding XML_N. This XML_N that represents a very small portion of the document can be converted into XML_S followed by XML_B.

Algorithm 1: Character_Translation_Algorithm

Input: Plain_XML

Result: XML_N

```
1  var XML_N;
2  var translate;
3  for All the tokens in Plain_XML do
4      if The token is a "data" part of the XML then
5          translate ← Use Symbol Table to translate each character of the token;
6      if The token is a "structure" part of the XML then
7          translate ← Use Tag Table to translate whole token;
8          if The word is a "tag" in the XML then
9              translate ← 'T' concatenate(+) translate;
10         if The word is a "attribute-name" in the XML then
11             translate ← 'A' concatenate(+) translate;
12     if The word is a "Tag Closing" then
13         translate ← '0';
14     XML_N ← XML_N + SPACE + translate;
15 end
16 return XML_N;
```

Algorithm 2: Packing_Algorithm

Input: XML_N

Result: XML_S

var $XML_S \leftarrow null$;
Assign all the 13 symbols in XML_N to an unique 4-bit binary;
var *cursor* $\leftarrow$ First Symbol in XML_N;
for *cursor* is not end of XML_N **do**
    if A character at *cursor* $+ 1$ exists **then**
        var *cursor*1 $=$ *cursor* $+ 1$ (i.e. next symbol);
        Accommodate 4-bit binaries equivalent of symbols pointed by *cursor* and *cursor*1 into a single byte;
        Concatenate the created byte with current value of XML_S;
    if If *cursor* $+ 1$ is end of the XML_N **then**
        Create a byte using the 4-bit binary of character at *cursor* and 0000;
        Concatenate the created byte with current value of XML_S;
    cursor $=$ *cursor* $+ 2$;
end
return XML_S;

3.2.3 JSON Compression

The JSON document shown in Listing 4.2 conveys the same information and exhibits a parallel structure to the XML document in Listing 4.1, both consisting of a “Data" and a “Structure" part. The two representations can be easily converted to each other. Therefore, the character translation technique discussed in the section 3.2.2 can be applied to both representations and the output would be the same. However, most related research on compression is in terms of XML, and only XML datasets are available for comparative experimentation, therefore we have chosen to explain our work in the terms of XML, fully cognizant of the fact that mostly web APIs exchange information in JSON.

3.2.4 Compression of non-English based XML

The effectiveness of the proposed XML compression technique depends to a large extent on the number of digits in the entries of the symbol table. This number depends on the number of unique characters that the XML is composed of. Each character in the symbol table is assigned a unique number. The number assigned to these characters solely depends on number of unique characters used in XML and is not related to the encoding of characters (e.g., UTF-8, UTF-16, or UTF-32). Similarly, the tag table contains mappings of element and attribute names of the XML to the smallest possible number based on the frequency of occurrence the concerned tag and is also not related to the encoding of characters in the tag. In the proposed method, we have only considered the English alphabet because of two reasons.

- Our work is motivated by the need to improve the performance of web applications through the compression of API responses. Given that 60.7% of the content on the Internet is in English[46] it stands to reason that most of the API responses use English characters. With this, one can safely assume that most XML documents comprise content only in English and this significantly helps compression as it makes the entries in the symbol table be of just two digits.

- If we consider XML documents with non-English characters, the number of digits re-

Table 3.3: A subset of the Symbol Table having character to 3-digit number mapping

.	..
E	038
d	069
m	078
o	080
.	..

Listing 3.2: Example of the XML_N when Symbol Table has character to 3-digit number mapping

```
T2 A4 013014015006016007006016 T3 025026027028029027030028 0 T6
   025026027029029027030028 0 T5 029028025034023025029023029035 0 T1
   023 0 T1 024 0 0
```

quired for the symbol table entries increase and the compression is not as effective. Finally, because earlier research on XML compression assumes English based content only, we too show compression results on XML documents in English.

The proposed method can be easily updated to work on non-English characters. If the maximum number of unique characters that an XML can have is increased to 999, which is enough to accommodate most non-English characters, then each character in the symbol table would be mapped to a number of three digits. The details on which three-digit number is to be mapped with which character in the symbol table, can be worked out at the time of implementation. One possible approach to render some efficiency could be to assign a small number to a character with a small Unicode and a large number to a character with a larger Unicode. A subset of such a symbol table is shown in Table 3.3:

The value of XML_N ((i.e., result of character translation)) in this case is given in Listing 3.2. The compression ratio corresponding to a three-digit symbol table is shown in Table 3.4. It is clear from table that even with a 3-digit symbol table, the proposed method is superior to XMill and SMCA in compressing small size XML documents.

3.2.5 Compression of XML Containing Binary Data

XML documents may also contain binary data. It is important, therefore, that the proposed technique for XML compression works on binary data. To denote the start of binary data in an XML document, we introduce the fourteenth symbol in XML_N as 'B'. 'B' like other contents in XML_N is also converted into a unique 4-bit binary in XML_S. To convert binary data to XML_N, we apply a technique like Base-64 [33]. This involves dividing the binary data into 6-bit chunks. If the two-digit number that a 6-bit chunk represents is N, then the number N+1 is calculated and the two digits of N+1 are appended to XML_N. While creating groups of binary data of size 6 bits, if the last group cannot be made of 6 bits owing to very few bits remaining, then extra 0s are added to make it 6 bits long. Binary data is always represented in terms of a number of bytes. Let us assume, for example, that the binary data here is 3 bytes long, that is, 24 bits. We make four groups of 6 bits from these 24 bits. A group of 6 bits represents a number having two decimal digits. In our approach, we use a four-bit binary (i.e., a nibble) to represent each decimal digit in the packing stage of compression. Two decimal digits obtained from a group of 6 bits will, therefore, be accommodated in 8 bits. This means that it would take 8 bits in XML_S to represent binary data 6 bits long and thus it will take 32 bits to represent binary data of 24 bits. The number of bytes required in XML_S to represent the binary data therefore becomes 4/3 times the actual size of the binary data. This expansion is mediated by the back-end compression. Representing binary data with the proposed method permits 10 symbols in a nibble. Therefore, only 100 different values are possible in a byte instead of 256 values. Being able to accommodate only 100 values in a byte increases the probability of repetition of a byte. Repeating bytes in a binary document are very well handled by back-end compressors like PPMD [19]. Since previous research does not consider compression of binary data, we have not been able to conduct a comparative study of our technique in this regard.

3.3 Evaluation

Experiments are conducted in this section to evaluate the hypotheses raised in Section 3.1. In the experiments conducted, we demonstrate that better compression is achieved if fewer containers

are used to compress an XML. The experiments also show that in large size XML documents with a relatively small number of containers, the number of containers does not influence the compression ratio. The experiments demonstrate that, given the independence of the proposed technique from the use of containers for compression, the proposed technique performs very well on compression of small size web API XMLs.

3.3.1 Experimental Design and Dataset

To assess the efficacy of the proposed technique, we apply it to two benchmark datasets: the first is the one used in SMCA [26] (Section 3.3.2); and the second is the one used in the survey paper of S.Sakr *et al.*[49] (section 3.3.3). The proposed compression technique is implemented using Python3. It is important to note that in the experiments, we calculate the compression ratio as the fraction of the size of the original XML document over the size of the compressed document ($CompressionRatio(CR) = XMLSize/CompressedSize$); a higher compression ratio, therefore, implies a better compression. The configuration of the system used for the experiments comprises: an i3-4005U CPU @ 1.70 GHz processor and a 4 GB DDR4 RAM.

3.3.1.1 The First Benchmark Dataset

The first benchmark that we compare our technique against is the SMCA [6] method. We choose SMCA because it is the leading XML compression technique in terms of compression ratio. We were unable to find an implementation of SMCA and therefore compared the compression ratios achieved for our proposed technique with the ratios reported in SMCA [6]. The results are included in Section IV-B. The dataset used to compute the compression ratios of our technique is the same used by SMCA.

The dataset used by SMCA [26] comprises 160 XML documents, 1.xml, 2.xml, and so on, divided into four groups of 40 XMLs each, labelled "small", "medium", "large", and "very large", to indicate four different size ranges. The XML documents in each of these groups are then aggregated to form four new XML documents, i.e., small.xml, medium.xml, large.xml, and vlarge.xml. In addition, 16 more XML documents, S1.xml to S16.xml, are created by combining the first 10, 20, 30 and 40 XMLs of each of the above groups; i.e., S1.xml combines

the first 10 documents of the small set, 1.xml to 10.xml. The XMLs S5.xml to S8.xml are created by combining all 40 XMLs of the small group and respectively the first 10, 20, 30 and 40 XMLs of the medium group (for example: s5.xml is a combination of the 40 XML documents of the small group and the first 10 documents of the medium group; s6.xml is a combination of the 40 XML documents of the small group and the first 20 XML documents of the medium group and so on until s8.xml). Similarly, S9.xml to S12.xml are combinations of all the XMLs of the small and medium groups and respectively the first 10, 20, 30 and 40 XMLs of the large group. Finally, S13.xml to S16.xml are combinations of all the XMLs of the small, medium, and large groups and respectively the first 10, 20, 30 and 40 XMLs of the vlarge group. Therefore, in this way the first dataset contains a total of 180 XMLs. The original 160 documents, small.xml to vlarge.xml and S1.xml to S16.xml.

As Table 3.4 shows, the dataset contains XML documents ranging from 4673 bytes to 2072248 bytes. The number of containers in each of these XML documents depends on the number of independent unique paths in each document. This number (18, in this dataset) is the same in all XML documents in spite of large variations in terms of amount of data. For a small XML document, 18 containers have a negative impact on compression because each container has very little data. Therefore, the compression results of SMCA which utilizes containers for compression is inferior for small XML documents. The proposed technique, on the other hand, only uses one container (i.e., XML_S) for compression and thus does much better than SMCA with small XML documents. As the size of the XML increases, the data content in the 18 containers used by SMCA starts to increase, and so does the compression ratio. The 18 containers soon become insignificant when the size of the XML becomes very large. Conversely, the proposed technique continues to use one container for large XML documents and its compression ratio starts to converge with that of SMCA. Figure 3.3 shows a comparison of compression ratios of the proposed compression technique with SMCA. The proposed technique maps frequently appearing XML tags with numbers having a small number of digits in the tag table. This further improves the compression as tags appearing a large number of times are represented by short numbers that take less space. In this way the Hypotheses raised in Section 3.1 are verified.

3.3.1.2 The Second Benchmark Dataset

The second benchmark dataset comprises 24 XML documents, three of which could not be downloaded. We use 21 of these documents, therefore, and the range in size from 533,579 to 137,538,931 bytes. This dataset was used to evaluate an earlier generation of XML compressors [49]. We choose the XMill XML compression algorithm from this generation of compression algorithm. This is because XMill is widely considered to be a standard XML compression technique and consistently maintains a good compression ratio. Furthermore, the implementation of XMill is easily available. We downloaded the XMill implementation from the following url: *https://sourceforge.net/projects/xmill/* and compared the compression ratios and run-times of both methods. The results of these experiments are reported in Section 3.3.3. Figure 3.6 shows a comparison of the proposed method and XMill on the second dataset.

3.3.2 Compressing API-response XMLs

Certain compression techniques are specifically designed for compression of XML documents, produced and consumed by SOAP-based web services. Today, the state-of-the-art in such techniques is SMCA[26]. The thesis demonstrates the efficacy of SMCA using a dataset comprising SOAP requests and responses from various sources. These requests and responses are systematically aggregated to form documents of varying sizes. The dataset is now considered a standard and several compression algorithms have been tested against it, e.g., [5] and [1], with SMCA leading in terms of compression ratio.

According to [26], the SMCA technique was implemented using Visual Basic on an Intel(R) Xeon(R) CPU E5-1630 v3@ 3.70 GHz, 16 GB RAM. Tables 3.4 and 3.5 include data on the performance of SMCA provided in [26]. The same tables include results on the performance of our technique and these are based on actual execution on our system using this dataset. The other significant compression technique XMill, as stated earlier, was downloaded by us on our system (configuration of our system is provided in the previous section) using the executable file downloaded from the sourceforge website and made to work on the same dataset.

Table 3.4 provides data on the compression ratio of our technique and that of SMCA and XMill. The same table is graphically illustrated in the Figures 3.3 and 3.2. The compression

ratios of the three techniques are compared on 20 XML documents from the dataset namely small.xml, medium.xml, large.xml, vlarge.xml and XMLs from S1.xml to S16.xml. The compression ratio results clearly indicate that the proposed technique outperforms both SMCA and XMill for XML documents of all sizes except very large ones where our compression ratio is somewhat equal to that of SMCA. Our compression ratio is much superior when compared to SMCA for small size XML documents but the gap progressively reduces as the size of the XML document increases. SMCA mostly outperforms XMill except in the case of very small XML documents.

In web services the XML documents exchanged are mostly small in size and thus the proposed technique is the most appropriate. SMCA, on the other hand, does not perform well with small XML documents and hence is not suitable to be used with web-services. As both SMCA and XMill use text compressors at the back-end, we compare their compression results with XML_B.

In terms of the time required for compression, Table 3.5 shows that our technique takes longer than SMCA and XMill. This is mainly because our technique supports parsing and querying even in the compressed form (more precisely, parsing and querying are supported until the stage that XML_S is formed as described in the relevant section earlier). SMCA, on the other hand, forms compressed documents that do not support parsing and querying and hence the shorter time for compression.

Comparison of the compression ratio of our proposed technique with that of XMill is shown in Figure 3.4. The graphs have been traced over all the 180 XML document of the dataset. The proposed technique achieves a much better compression ratio across the dataset. Most XML documents in the dataset are smaller than 60000 bytes and therefore for better visualisation and more detailed analysis we compare the two techniques for XML documents smaller than 60000 bytes in Figure 3.5. Such documents make up 168 of the total 180 XML documents in the dataset and our proposed technique is seen to perform much better than XMill.

Table 3.4: Compression Ratio Comparison on SMCA dataset

XML file	**Size (bytes)**	**Our method CR (2 Digit Symbol Table) (Only English)**	**Our method CR (3 Digit Symbol Table)(Non-English)**	**SMCA CR**	**XMill CR**
S1	4673	6.49	5.45	4.07	4.69
S2	8567	8.61	7.40	5.45	6.38
S3	12147	9.72	8.53	6.24	7.22
S4	16475	10.84	9.68	6.93	8.02
S5	33563	13.59	12.41	9.39	10.04
S6	50716	15.09	13.79	10.79	10.06
S7	71528	16.33	15.00	12.13	9.68
S8	90785	17.13	15.78	13.20	9.06
S9	211795	19.61	18.05	17.04	10.41
S10	336198	20.63	19.01	18.85	10.97
S11	456942	21.15	19.47	19.91	11.36
S12	555800	21.54	19.82	20.56	11.42
S13	939150	22.31	20.54	21.96	12.46
S14	1339834	22.75	20.88	22.59	13.08
S15	1717316	22.96	21.05	22.84	13.43
S16	2072248	23.15	21.20	23.10	13.69
small	16475	10.87	9.68	6.94	8.02
medium	74311	16.54	15.20	13.03	8.59
large	465016	21.45	19.38	20.66	11.31
very large	1516449	22.85	20.79	23.36	13.26

Table 3.5: Time Comparison (ms)

XML file	**XML Size(bytes)**	**Our method**	**SMCA**	**XMill**
S1	4673	71	13	27
S2	8567	70	15	25
S3	12147	80	20	24
S4	16475	71	27	25
S5	33563	90	45	26
S6	50716	130	62	28
S7	71528	140	76	33
S8	90785	142	78	32
S9	211795	150	91	43
S10	336198	226	138	45
S11	456942	240	187	51
S12	555800	312	219	58
S13	939150	420	311	70
S14	1339834	688	405	90
S15	1717316	812	500	102
S16	2072248	1013	655	125

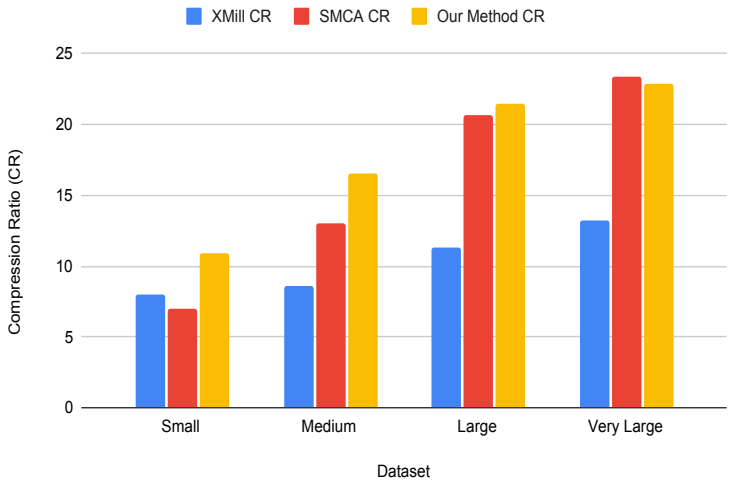


Figure 3.2: Compression Ratio Comparison on Small, Medium, Large and Very Large Group (SMCA dataset)

3.3.3 Compressing Document-style XMLs

The 24 XMLs used by [49] exist in two forms each, the original documents and the structured documents. Original XML documents are full XML documents comprising both structure and data parts. Structured XML documents, on the other hand, consist of only opening and closing tags. These documents, therefore, only have structure information and no data part.

We compare the compression ratios of the proposed technique with that of XMill using only the original XML documents, the ones with both structure and data parts. The results are shown in Figure 3.6. It is clear that our technique outperforms XMill in terms of compression ratio with every XML document except two. These two documents have a large number of tags with relatively small sized free text data between the tags.

3.3.4 Evaluating the queryable feature of the proposed technique

The proposed technique maintains the characteristic of being queryable up to the stage of XML_S (the various stages of compression are shown in Figure 3.1). After applying back-end compression, however, the compressed document is no longer queryable. The compression ratios for

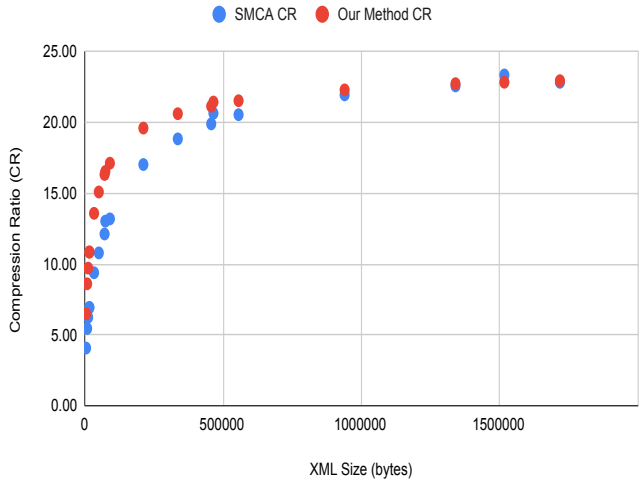


Figure 3.3: (SMCA vs Proposed Method) Compression Ratio Comparison on 20 XMLs of SMCA Dataset

documents at the stage of forming XML_S are naturally smaller than the final compression ratios which also incorporate back-end compression. The significance of the documents at this stage, as stated earlier, is that they are queryable as opposed to the document formed after back-end compression.

XGrind is perhaps the oldest and most significant research endeavour in the direction of Queryable XML Compressors, whereas QRFXFreeze [51] is a more recent technique and outperforms most queryable compressors in terms of compression ratio. The compression ratio of the proposed technique, at the stage of XML_S, is compared with the compression ratio of QRFXFreeze. The dataset that is used to compare QRFXFreeze with other queryable compression algorithms in [51] is also used here to compare the proposed technique with QRFXFreeze. Table 3.6 shows this comparison and XML_S of the proposed technique has a better compression ratio than QRFXFreeze for four out of the five XML documents used. In [51] the compression ratio is calculated using the following formula:

$$cr = [1 - \frac{sizeof(compressed file)}{sizeof(original file)}] * 100$$

To make this consistent with our technique we calculated:

$$\frac{sizeof(original file)}{sizeof(compressed file)} = \frac{100}{(100 - cr)}$$

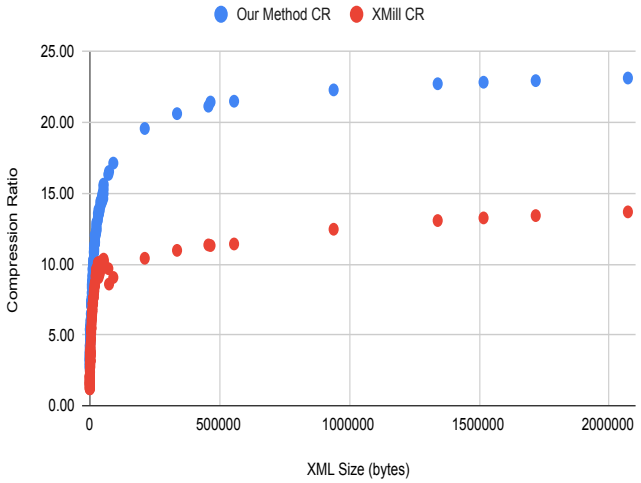


Figure 3.4: (XMill vs Proposed Method) Compression Ratio Comparison on all 180 XMLs of SMCA dataset

Table 3.6: Compression Ratio of XML_S (proposed method) vs QRFXFreeze

XML file	**XML Size (MB)**	XML_S **CR**	**QRFXFreeze**
XMark	118	2.45	2.38
DBLP	138	1.91	1.92
TreeBank	89	2.20	1.17
Shakespeare	7.9	2.38	2.22
SwissPort	109	2.18	2.13

Further

$$\frac{sizeof(originalfile)}{sizeof(compressedfile)}$$

is compared with the compression ratio of the proposed technique.

3.3.5 Evaluation of JSON Compression

A JSON document also contains a data part and a structure part as discussed in Section 3.2.3. Therefore, the same outputs, XML_N (subsequently XML_S and XML_B), are generated irrespec-

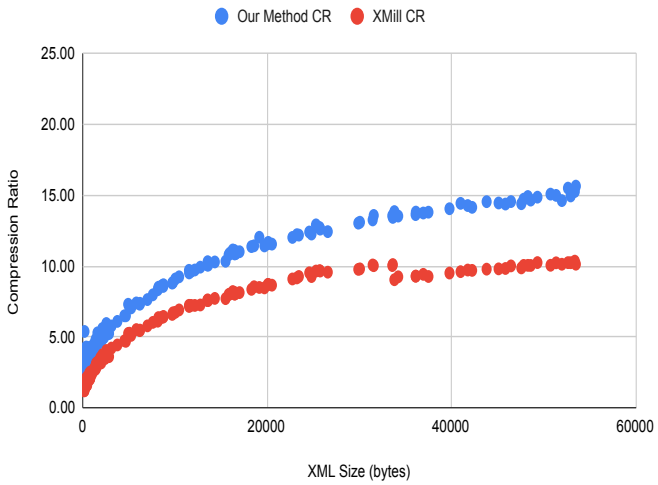


Figure 3.5: (XMill vs Proposed Method) Compression Ratio Comparison on 168 XMLs Having Sizes Less Than 60000 bytes using SMCA dataset

tive of whether XML and JSON representation is used. However, as major research endeavours use XML compression, we have also evaluated our research using XML.

3.3.6 Size, Information and Compression Analysis

XML documents may vary in their nature based on the number of characters dedicated to their data part and the number of characters dedicated to their structure part. The structure part of an XML comprises tags and attributes whereas the data part is the content between tags. An XML document may have a large data part with a large number of characters and very few tags and attributes. Another document may have a large number of tags and attributes with a large number of characters dedicated to these and very few characters for data. Certain XML documents may have a small number of tags and attributes but the size of each tag and/or attribute may be very large and hence a large number of characters dedicated to the same.

Our proposed technique for XML compression varies in its compression efficacy with different types of XML. In this section, we seek to understand the behaviour of the technique in this respect. To this end, we create synthetic XML documents of varying characteristics as described above. We start with a template XML document consisting of n slots. A slot can be filled with any one of the following elements of an XML document: an opening tag, a

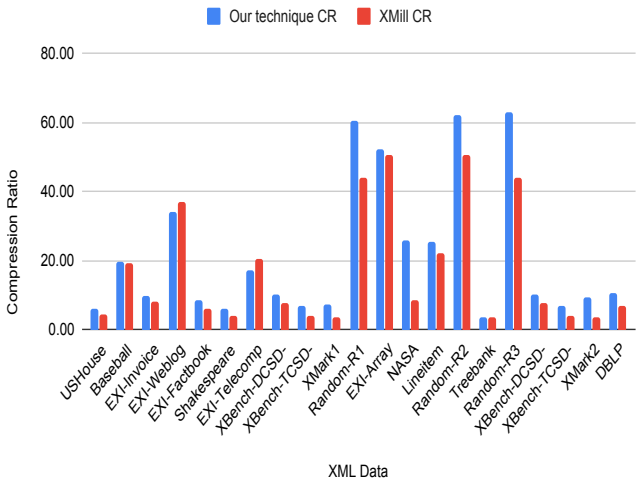


Figure 3.6: Compression ratio comparison on original XML documents (longer the bar better is the compression ratio)

closing tag, an attribute name, a single character of an attribute value, or a single character of data content. It is important to note that irrespective of the size of an opening tag, closing tag, and attribute value, each occupies a single slot; whereas one character of the data occupies a single slot. This means that if a single element of data comprises 10 characters, 10 slots would be required to accommodate the data element. Conversely, if an opening tag, closing tag, or attribute name is of size 10 characters, it would be accommodated in just 1 slot. The intent of defining slots in this manner is to understand the varying nature of XML documents and the variation in the effectiveness of the proposed technique with such varying documents.

We start with a document comprising only opening and closing tags that fill all the slots, resulting in an XML of *n/2* opening and *n/2* closing tags. The next XML document is created by substituting one pair of opening and closing tags with two data characters (as stated earlier, one slot can accommodate an entire opening or closing tag but only one character of data). Next another pair of opening and closing tags is substituted with data characters. This is done with successive pairs of opening and closing tags being replaced with data characters until *n/2* XML documents are created with the first one comprising only tags and *n/2* empty elements, and the last one containing a single element (just one pair of opening and closing tags) and *n-2* data characters.

To understand the behaviour of the proposed technique, we express the nature of the XML documents in terms of *Information*. To do this, let us assume that an XML document with *n*

slots has a total of t tags with an average size of x characters per tag, a total of a attributes with an average size of y characters per attribute, and has a total of d characters in the data part of the XML. Following this assumption, the total size of an XML with n slots is $(t*x+a*y+d)$ and $2t+a+d=n$ as it is assumed that the n slots are filled with the given number of tags, attributes, and data characters. The ratio $(t+a+d)/(t*x+a*y+d)$ represents the *information* that the XML carries and this ranges between *0* and *1*. Among XML documents with n slots, the XML that has a smaller value of $(t*x+a*y+d)$, carries more *information* and we notice that such documents (that carry more information) perform poorly in terms of compression ratio with the proposed compression technique. When t and a are both 0, all the slots in the XML document are filled with single characters of data and exhibits the maximum *information*, i.e., 1. With such documents, the proposed technique returns the worst compression ratio. Similarly when $d=0$ and $t << 2t*x$, $a << a*y$, then all the n slots are filled with large tags or attribute names and the *information* that the XML document carries is small, i.e., close to zero. In such cases we notice that the proposed technique gives good compression ratios.

We seek to understand the behaviour of the proposed technique in a little more detail by studying the variation of its compression ratio with varying *information*. To do this, three sets of XML documents are created each with the number of slots n being set at 10000. Each set comprises 5000 XML documents created in a manner described earlier: the first document in each set comprises only opening and closing tags; in a document with 10000 slots, this works out to 5000 of each opening and closing tags. For successive XML documents, tags are replaced with data characters and the next document has 4999 opening and closing tags and 2 characters of data. This is continued until the last (5000th) XML document in each set has just 1 opening and closing tag and 9998 data characters.

Each set of XML documents created through this process is populated with tags from three different schemas. The first set of documents is created using the “XBench-TCSD-Normal_Structural.xml" document from the survey paper [49] and comprises very small sized tags of average length 2 characters per tag. The second set is created using the “S16.xml" document used with SMCA [26] that comprises medium-size tags. The average length of tags in “S16.xml" is 10 characters per tags. Finally the third set uses the “EXI-factbook_Structural.xml" document again from the survey paper [49] that comprises large tags with an average size of 24.56 characters per tag.

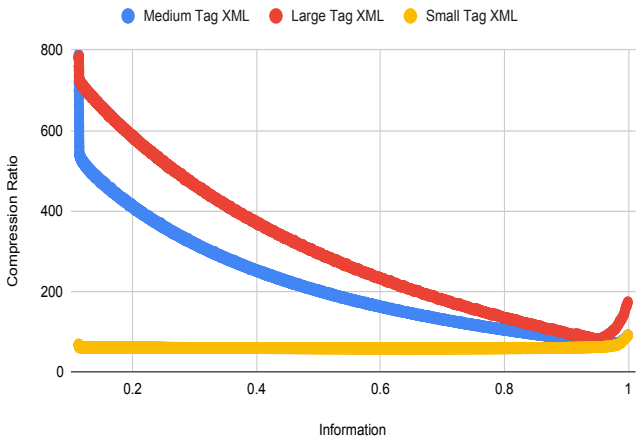


Figure 3.7: Behaviour of the proposed technique on various types of similar XMLs (5000 XMLs in each set)

Figure 3.7 demonstrates the behaviour of the proposed compression technique. For the same value of *information*, an XML with a large tag size results in a better compression ratio. The compression ratios for the XML documents of all three sets decrease with an increase in the value of *information*. This is very apparent for the sets with large and medium sized tags and less apparent for the documents with the set of documents with small tags. The same relation holds true for small tag documents but the graph appears like a straight line because of the small compression ratios. In Figure 3.7, a surge appears in the compression ratio for documents whose value of *Information* is close to 1. This is because XML documents are created for our experiments by successively reducing the number of tags one at a time and replacing these and thus increasing the data content of the document one character at a time. When the number of tags is sufficiently small and the data content sufficiently large, a further decrease in number of tags and their replacement with data characters does not significantly reduce the size of the new XML, while resulting in comparable improvement in compression. This results in what appears to be a little like a surge in the compression ratio.

3.3.7 Complexity Analysis

Decompression of compressed document is the reverse of compression and involves the following procedure: the back-end decompressor is first used on the XML_B document to get XML_S;

next the nibbles from XML_S are extracted one-by-one to get XML_N; and finally, the symbol and tag tables are used to get the original XML. The complexity of both the compression and decompression processes depends on: the number of slots in the XML document, and the complexity of the PPM algorithm, O(n) where n is the number of bytes in XML_S. Slots are the substitutable entities of an XML that are substituted using symbol and tag tables. These include tag names, attribute names, a character in data, and others. The slots of an XML are more comprehensively described and formally defined in Section IV-F. Both compression and decompression involve substitution followed by PPM compression or decompression. The number of slots is the same both in plain XML and in XML_S. Therefore, the complexity of both the compression and decompression processes is O(s+n).

3.3.8 Discussion

We compare the efficacy of our method against state-of-the-art competitors like XMill [37], QRFXFreeze [51], and the recent SMCA [26]. XMill is the pioneer among XML compression techniques and continues to serve as a benchmark. SMCA is the latest and the state of the art in XML compression. The Figure labeled “(SMCA vs Proposed Method) Compression Ratio Comparison on 20 XMLs of SMCA Dataset" i.e. the Figure 3 clearly indicates that our method is superior to SMCA in terms of compression ratio for small and medium size documents. For large documents, however, the proposed method performs similarly to SMCA, and in some cases SMCA performs marginally better. Figures 2 and 5 show that our method performs better than XMill on documents of all sizes. QRFXFreeze, unlike SMCA and XMill, is a queryable XML compressor. The proposed method is also queryable until the final back-end compression stage and hence we compare the queryable form (which we called XML_S, shown in Figure 1 of the proposed method with QRFXFreeze. QRFXFreeze has only been tested on XML documents of very large size and we, therefore, compare XML_S with QRFXFreeze for such documents. We notice that XML_S comfortably outperforms QRFXFreeze in terms of compression ratio as shown in Figure or Table 3.6.

Note that SMCA and XMill, as stated earlier, are non-queryable and hence we compare their outputs with XML_B, the final, non-queryable, output of our method, as produced by the last stage of back-end compression is done.

In Section 3.3.6, we try to gain some insight on the kinds of documents that are most amenable to compression and hence provide superior compression ratios. We find that our method gives inferior compression ratios for XML documents with a relatively large number of characters in the data part, as compared to the structure part. On the other hand, if the structure part of the XML documents has a large number of characters our method works better.

It is important to remember that JSON documents, like XML documents, are divided into structure and data parts, hence the proposed method is equally applicable and effective on JSON documents as well. Recently, a new encoding method, the Google Protocol Buffer [34], was introduced. This can be used to represent messages in a format that is shorter than JSON. In Google protocol buffer messages are represented in a binary form, in terms of key-value pairs of structured and free-text data. Different types of data are represented using different encoding schemes in Google protocol buffer. Although the Google Protocol Buffer is an interesting option for resource representation, but it is not as popular and is seldom used in web services. Messages between web services are still mostly transferred as JSON or XML.

3.4 Limitations

Although the proposed method performs better for small XMLs of web-based APIs, it has some limitations.

- As the number of characters in the XML increases, the digit count in the numbers used in the symbol tables increases. Therefore, the effectiveness of the proposed method in cases where a large number of characters is required is a little suspect. An example of this is its use for cases where non-English characters need to be used in the data part of the XML. This would make the compression less effective.

- When binary data is compressed with the proposed method, the size of the binary content in the intermediate XMLS increases by 1.33 times instead of decreasing. The back-end compression partially compensates for this expansion.

- The compression ratio of the proposed method on small XML documents is better than SMCA. However, in case of large XML documents, especially those larger than 2MB,

the compression ratio of the proposed method is mostly equal to that of SMCA.

3.5 Conclusion and Future Work

The key contribution of this work is a novel compression technique for structured documents like XML and JSON. The design is motivated by the need to improve the efficiency of network usage during service message exchange.

The proposed technique takes advantage of the structure of XML documents and relies on the intuition that the natural language underlying the messages exchanged requires fewer characters than the complete ASCII character set, under the realistic assumption that most service APIs are based on the English language. This, therefore, enables us to map structure tags and characters of the data elements into a much smaller character set. Beyond this translation step, the proposed technique further reduces the size of the document through a byte-packing step. This results in a significantly compressed document that importantly retains its queryable characteristic. The size of the document may further be reduced using a traditional compression algorithm but at this stage the document loses the property of being queryable.

Our experiments demonstrate that our method comfortably outperforms the current state-of-the-art compression technique, SMCA, in terms of compression ratio on files smaller than 1 MB (Table 3.4). On files larger than 1 MB, the proposed technique returns a compression ratio on par with that of SMCA. It is important to note that, for the most part, XML messages generated during interactions between ReST services are quite small. The proposed technique, therefore, is quite effective in improving the utilization of network resources in Service-Oriented systems which are mostly ReSTful in nature today, as compared to SMCA.

For document-style XML, the proposed technique outperforms XMill, long considered a standard compression technique for such XML documents, in terms of compression ratio (Figure 3.6). In this case, however, the final form of the compressed document using the proposed technique is non-queryable. This is because the final stage of compression involves utilising a traditional text-compression algorithm.

The stage of compression just before this final stage (as shown in Figure 3.1) results in

a document that is queryable. The compression ratios for these queryable compressed documents are compared with the compression ratios of the state-of-the-art query-friendly compressor XQRFFreeze [51]. The results (Table 3.6) show that the proposed technique compression ratios at the queryable stage outperforms the best in literature.

In the future, we will attempt to address the limitations mentioned above. Better compression of the binary data being transferred through XML or JSON can be an important avenue for future research. In addition, the compression of XML or JSON documents using non-English characters can be improved.

Chapter 4

A Size efficient Encryption Technique for Structured Documents Like XML and JSON

Security of XML and JSON documents plays an equal if not more important role than compression in ensuring effective web service interaction. In this chapter we proposed a new technique for encryption of XML and JSON documents. We extend the compression technique discussed in Chapter 3 in a manner that it is effectively harnessed for encryption.

4.1 Understanding the Problem

The need to secure contents of documents from unauthorized access is a fairly common issue nowadays. Security of documents comprises upholding the following three tenets: confidentiality, integrity and availability. Confidentiality implies that a conversation between the sender and receiver makes no sense to a potential intruder eavesdropping on the conversation or a stored

Listing 4.1: XML1

```
<root attr1="value1" attr2="value2">
<name>iiti</name>
<value>2</value>
</root>
```

Listing 4.2: JSON1

```
{
  "root": {
    "-attr1": "value1",
    "-attr2": "value2",
    "name": "iiti",
    "value": "2"
  },
}
```

document on a machine. Confidentiality is provided by encryption of a message; Integrity implies that a message in transit between a receiver and a sender remains unchanged during transit. Integrity is the assurance that the information is trustworthy and accurate. Digital signatures, checksum are examples of methods that ensure integrity and authenticity; Finally, availability is a guarantee of reliable access to the document by authorized personnel.

There are mainly two types of encryption algorithms: symmetric key encryption and asymmetric key encryption algorithms. In asymmetric key encryption, the sender maintains two keys of which one is 'public' and the other 'private'. The sender makes the public key available to all, and the private key is kept hidden. A message that is encrypted using the public key can only be decrypted using the private key. In symmetric key encryption, both the sender and receiver maintain a shared secret key between them called the symmetric session key. The symmetric session key is used both for encryption and decryption. To send a message, the receiver encrypts the message with this key and upon reaching the sender, the same key is used for decryption. A few examples of symmetric key algorithms include DES [20] and AES [31]. RSA [48] is the most popular asymmetric key cryptography algorithm.

XML and JSON are structured documents wherein elements are arranged in a parent-child or name-value relationship (Listing 4.1 and 4.2). An example of XML is shown in Listing 4.1 and its corresponding JSON is shown in Listing 4.2. XML or JSON are encrypted like a simple document and alternatively their structured nature is harnessed in the encryption process. There exists a W3C recommendation for XML encryption [30] that prescribes using existing blocks or stream ciphers to encrypt a document. Two such algorithms are Advanced Encryption Algorithm (AES) [31], and Data Encryption Algorithm DES [20]. The XML encryption recommendation also includes different versions of the Standard Hash Algorithm [22] for creating a message digest. The XML encryption recommended by W3C works on an XML document as a whole, on a part of the XML document, or even on binary documents. The idea of the W3C recommendation is to encrypt documents using existing algorithms and to represent the encryption in XML using special XML tags. The W3C recommended approach requires XML canonicalization [17] to preserve the namespace prefix bindings and the value of attributes in the "xml" namespace. According to W3C "A Canonicalization of XML is a method of consistently serializing XML into an octet stream as is necessary prior to encrypting XML." XML canonicalization is an approach of serializing XML to create a standard canonical representation of an XML document.

In this chapter, we develop a new technique for XML and JSON encryption by making certain changes to the well known XML compression technique, XMill [37]. XMill only deals with XML documents, but it can easily be modified to be applicable to JSON as well. The proposed technique extends the ideas of XMill and harnesses these for encryption in a manner that it is equally effective for both XML and JSON. The technique mostly builds upon the compression technique proposed by G.P. Tiwary *et. al.* [58] for XML and JSON documents generated by web APIs. The XML and JSON compression technique proposed in [58] outperforms XMill [37] on web API based XML documents in terms of compression ratio. The encryption technique proposed effectively encrypts the documents in addition to also compressing them.

The restriction with the proposed technique is that it only works on structured documents that are divided into *"structure"* and *"data"* parts (as in XML and JSON). In this chapter, we harness this structured nature of XML or JSON for more effective encryption. In the proposed method of encryption, serialization process of XML also introduces a degree of randomization in overall encryption and also helps to reduce the size of final result of encryption.

The major contributions of this chapter incldue:

- An encryption technique for structured documents like XML or JSON is proposed
- The proposed technique, in addition to encryption, very effectively compresses structured documents as well.

4.2 The Proposed Technique of XML and JSON Encryption

In this chapter, a method of symmetric key encryption is proposed to encrypt XML and JSON documents. The method both encrypts and compresses XML and JSON documents. The goal of this work is to come up with an encryption technique that not only makes XML and JSON documents secure but also generates encryptions that are much smaller in size compared to other encryption techniques. We make the following major changes to the well known XMill [37] compression technique and to the method proposed by G.P. Tiwary *et. al.* [58] with the intent of simultaneously encrypting and compressing XML and JSON documents:

- A dictionary that maps the structure parts of the document to numbers is generated separately with the help of a symmetric key, on both the sender and receiver sides.
- One more dictionary is generated in which every character in the XML or JSON document is mapped to a number. This dictionary is also generated with the help of a symmetric key, separately on both the sender and receiver sides.
- The entire XML or JSON document is converted into a special binary document in a manner similar to that in [58].
- The final binary is then passed through a byte level encryption process.
- In a manner similar to XMill and [58], text based compression (e.g. bzip, gzip) is performed on the final binary before the byte level encryption process.

There are mainly four stages of encryption and decryption for the sender and receiver respectively. For the sender these stages are Initialization, Substitution, Compression, and Byte-

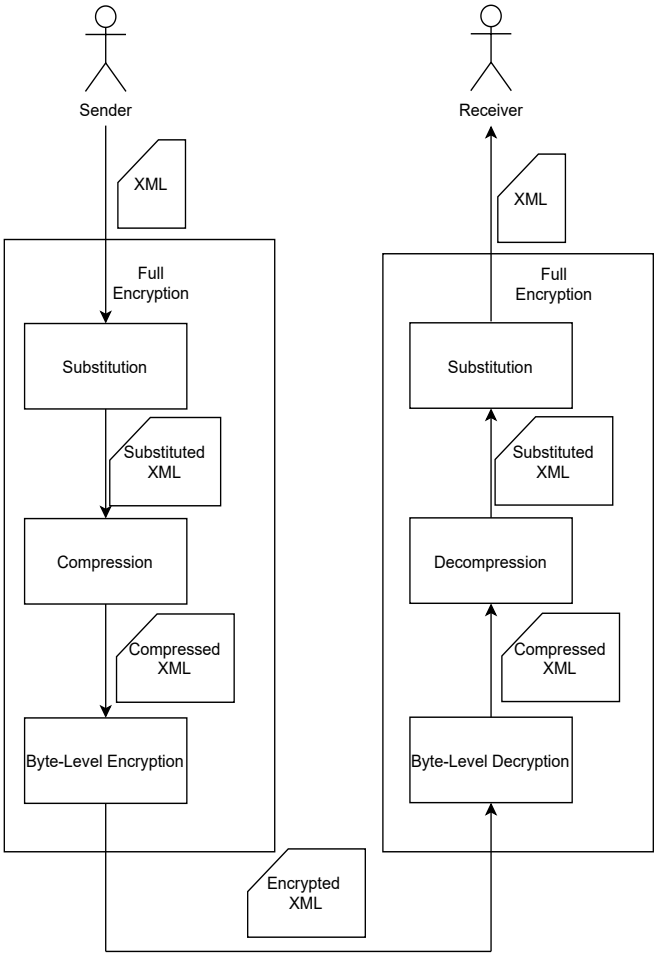


Figure 4.1: Post initialization stages of XML encryption and decryption

level encryption. For the receiver, these four stages are Initialization, Byte-level decryption, Decompression and Substitution. Figure 4.1 shows the stages barring the initialization stage.

4.2.1 Initialization (Initial stage for both sender and receiver sides)

The Initialization stage is the initial setup for both the sender and receiver. This stage is the pre-encryption or pre-decryption stage for both the parties. Before exchanging the actual XML document both the sender and receiver share a secret key called the *"10-element key"*. It is assumed that the receiver and the sender can securely share the secret key. Thereafter both the sender and receiver populate two dictionaries called the *"Temporary Table"* and *"Symbol Table"* with the help of the shared *"10-element key"*. The contents of tables created during initialization are the same at both ends. The steps of initialization are discussed in detail in

r	117
j	125
o	126
t	104
a	153
1	340
v	850
l	137
u	820
e	146
2	349
n	116
m	109

Figure 4.2: *A Subset of Symbol Table (ST)*

Appendix A. A Symbol Table (Figure 4.2) is a mapping of each character in the XML or JSON document to a unique number. Once the Symbol Table is created the Temporary Table is deleted.

In addition to the Temporary and Symbol Tables, both the receiver and sender maintain another table called the *"Tag Table"* (Fig 4.3). The Tag Table maintains a mapping between the *"structure"* parts in the XML or JSON document to a unique number. The Tag Table structure and creation are discussed in detail in Appendices A and B. The creation and use of the Tag Table happens at the substitution stage at both the sender and receiver ends.

root	4
name	5
value	6
attr1	8
attr2	9

Figure 4.3: *Tag Table (TAT)*

4.2.2 The Encryption (sender's) Process

After initialization, both the encryption and decryption processes execute at the sender and receiver ends respectively in the next three stages as shown in Figure 4.1. A detailed discussion of each post initialization stage at the sender side (i.e. stages of encryption) is included in Appendix B with only a high level discussion of each stage is included here.

1. **Substitution:** The substitution step has three purposes.

   (a) The first is to create a serialized form of the XML.

   (b) The second is to represent the entire XML with 15 different symbols.

   (c) The third is to introduce a degree of randomness in the final result of encryption.

   In the substitution step, the sender substitutes the characters of the *"data"* part of the XML using the Symbol Table, substitutes the characters of the *"structure"* part of the XML whose entry is not present in the Tag Table using the Symbol Table, and at the same time updates the Tag Table with the new entry. If an entry corresponding to the *"structure"* part of the XML is already present in the Tag Table, the entire *"structure"* part is substituted using the *"Tag Table"*. For example, in XML1 if the entry for the structure part "root" is not present in the Tag Table then every character of "root" is substituted using the Symbol Table (Figure 4.2) as *"T117126126104"*. At the same time the Tag Table entry for "root" is calculated. If the Tag Table entry for "root" already exists then "root" is substituted using the Tag Table (Fig 4.3) as *"T4"*. Here T represents Tag. Notice that the structure and the data parts of XML can be easily distinguished after the substitution stage. The document that is created in the substitution stage is called *"Substituted XML"*. The process of substitution is discussed in detail in Appendix B.

2. **Compression:** The purpose of this step is to decrease the size of the final encryption. The *"Substituted XML"* stage includes only 15 different symbols. One symbol is represented using 4 bits. The sender, therefore, can accommodate two symbols in the result of the substitution stage into one byte. The document that is created at the compression stage is called *"Compressed XML"*. The process of compression is also discussed in detail in Appendix B.

3. **Byte Level Encryption:** The *"Compressed XML"* has a good level of encryption but can still reveal several aspects of the original XML. Further, upto the substitution stage the proposed technique does not add a property of confusion and diffusion to the final encryption. That is why each byte of the *"Compressed XML"* is selected one by one and the selected byte is encrypted again according to the value of the key. The final result of the encryption subsequent to byte-level encryption never exposes the relationship between the key and the ciphertext or between the plaintext and ciphertext. The document

at the “Byte Level Encryption” stage is the final *“Encrypted XML”*. Details on byte level encryption are included in Appendix B.

4.2.3 The Decryption (receiver) Process

A detailed discussion of each of the post initialization stages at the receiver side (i.e. stages of decryption) is included in Appendix B and only a high level description of each stage is done here.

1. **Byte Level Decryption:** The receiver decrypts the received *“Encrypted XML”*. The result of Byte Level Decryption is called “Compressed XML”. The process of “Byte Level Decryption" is discussed in detail in Appendix B.

2. **Decompression:** The *“Compressed XML”* after the “Byte Level Decryption” stage is decompressed and converted into “Substituted XML” which forms the output of the substitution stage at the sender-side. The process of “Decompression" is again discussed in detail in Appendix B.

3. **Substitution:** The characters of the *“data"* part of the *“Compressed XML”* are substituted using the Symbol Table. The characters of the *“structure”* part of the *“Compressed XML”* are substituted using the Symbol Table for those characters that do not have a corresponding entry in the Tag Table. In such cases the Tag Table is also updated with the new entry. If an entry corresponding to a character in the *“structure”* part is already present in the Tag Table then the character is substituted using the Tag Table. The process of “Substitution” at the receiver end is discussed in detail in Appendix B.

4.3 Evaluation

We evaluate the working of the proposed approach along two aspects: 1) compression efficacy; and 2) security analysis against generic and XML/JSON specific attacks. These analyses are discussed in the following subsections.

4.3.1 Compression Effect of the Proposed Method

4.3.1.1 Dataset and methodology

We use three types of datasets for evaluating the proposed method. The first dataset comprises SOAP based requests and responses. This dataset is used by SMCA [26], a recent endeavour directed at aggregation and compression of XML. The authors of SMCA were very kind and shared the dataset with us. A second dataset was synthesized by us. This comprises XML documents of various types. The idea behind synthesising the dataset by ourselves was that the proposed method has different compression effects on different types of XML. We therefore included all kinds of XML documents in the synthetic dataset to comprehensively assess the working of the proposed method. The third dataset comprises XML or JSON that constitute the Request and Response documents for widely used REST APIs (these include APIs of Amazon, Google, and other popular web sites).

The size of encrypted XML depends on the value of $key[8]$ as explained earlier in Section 4.2. If less than 100 unique characters are used in the XML document, the value of $key[8]$ is at least 2, whereas if less than 1000 characters are used in the XML, the value of $key[8]$ is at least 3. Setting the value of $key[8]$ to the minimum possible does not affect the security, as discussed earlier. According to W3Techs survey on Jan 31, 2021, 60.6% of the content on the Internet is in English. We feel that this percentage is probably even larger in the case of the programmable web. If only English is used in an XML, then the number of unique characters in an XML is mostly less than 99. Further, it is practically impossible for more than 999 unique characters to be used in an XML. Therefore, in this chapter, we discuss the compression effect of the proposed method with the value of $key[8]$ being 2 or 3.

The proposed XML encryption technique is implemented using Python3. It is important to note that in conducted experiments, we calculate the compression ratio as the fraction of the size of the original XML document to the size of the compressed document as shown in Equation 4.1.

$$CompressionRatio(CR) = \frac{XML\ Size}{Compressed\ Size} \tag{4.1}$$

A high compression ratio implies a better compression. The experiments were conducted on an i3-4005U CPU @ 1.70GHz processor with a 4GB DDR4 RAM.

4.3.1.2 Compression ratio comparison against established XML compressors

The compression effect of the proposed encryption techniques for XML and JSON is similar to that discussed in Chapter 3. The validation of the compression effects are also similar and hence we do not include the same here.

4.3.1.3 Size comparison against 3-DES Encryption

We assess the working of the proposed method on 562 real world XML documents (Table 4.1). We encrypt the entire dataset using the proposed method and do the same with a 3-DES encryption. It is important to note that we do not use a backend compressor in this case. We use a value of $key[8] = 3$ for the execution. We find that in more than 90% of the cases, the proposed approach gives an encrypted document that is smaller size than that given by the 3-DES algorithm. The complete comparison between the two algorithms in terms of encryption size is available at: https://drive.google.com/open?id=1mm9ZOteOoafAvZspX8ZF-LFGGtTHxMaN. Also, if an API supports JSON then the JSON document is encrypted by first converting it to XML and subsequently following the same sequence of steps. The size of the resulting encryption is compared with the size of actual JSON. In a similar manner, the JSON is also encrypted using the 2-DES algorithm.

4.3.2 Security Analysis

The proposed encryption technique needs to be secure enough to protect the contents of the XML. In this section, the security features of the proposed algorithm are put to test. Our ap-

Table 4.1: Size comparison of proposed method vs 3-DES on real world XML documents

API Name	**Number of Tested Documents**	**Size of proposed method $<$ 3-DES**	**Size of proposed method $<$ 3-DES/2**
Amazon Product Advertising API	43	41	22
Amazon Alexa Music Skill API	62	62	28
Amazon Alexa Home Skills API	156	156	82
Paypal	133	130	42
AngleCam	96	54	34
Google Maps	13	13	9
YouTube	16	10	0
Gmail	18	15	10
CIMS California	10	10	10
CubeSensors API	4	4	4
Wise Orchard API	11	11	11
Total	562	506	252

proach uses three levels of encryption: the first level comprises substitution of the *"data"* and *"structure"* parts of the XML with numbers; the second level accommodates two numbers generated during the substitution phase into one byte ; and finally at the third level, the bit positions are swapped in each byte long data based on a function that depends on the value of the key and with proper XOR operations. The combination of these three levels ensures a high degree of security. In this section, security analysis of the proposed method is done by subjecting the encrypted content to various possible attacks.

4.3.2.1 Assumptions

Here we assume that the *10-element key* is exchanged securely and successfully between two communicating parties. The Symbol Tables and Tag Tables are also considered secure at both ends. It is important to note that both character substitution cipher (substitution stage) and block cipher (byte level encryption stage) are used in the proposed method of XML encryption. That is why the security of the proposed approach depends on both these techniques.

4.3.2.2 No relation between entries in the Symbol Table and Tag Table

Substitution, however, is the intermediate stage of the proposed method of encryption, and the substituted XML remains hidden from the rest of the world. Nevertheless, their security aspects have been tested in this section.

If the corresponding number of one character in *ST* is somehow found by an intruder, then it cannot guess the corresponding number of the other character in it. The reason behind it is a number corresponding to a character in the *ST* depends upon its *"row header(rh)"* and *"column header(ch)"* in the *"Temporary Table"* and the *"key[9]"* as: $(rh)^{key[9]}+(ch)^{key[9]} = STEntry$. The values of *"rh"* and *"ch"* of a character in the *"Temporary Table"* depends upon various elements of the *10-element key*. The result of above expression is further cut down to *"key[8]"* digits. Therefore just by knowing the value *"STEntry"* for one character it is impossible to find the values of *"rh"*, *"ch"* and *key[9]*. The Tag Table entry of a structure part of XML is calculated by summation of Symbol Table entries of its various constituent characters. The summation is further cut down according to number of entries already present in the Tag Table. Therefore Tag Table entries of structure parts cannot be related to each other. Therefore, looking at one entry

of a Symbol Table or a Tag Table, its other entries cannot be guessed.

4.3.2.3 No trivial relation between plain XML and encrypted XML

The lack of a trivial relation between the plain XMKL and the encrypted XML is explained better with an example. Let's assume a situation in which the value of the key is *"[12,6,1,1,1,14,4,1,3,2]"* and a text *"a1"* is to be encrypted. The steps of this encryption are shown below:

- Plain XML is a1
- Substituted XML is 152340
- Compressed XML is 00010101 00100011 01000000
- After swapping the content is 10101000 00100011 01000000
- Final encrypted XML is 10100010 01100000 01100100

The two numbers of the symbol table and the tag table are not at all related to each other, which introduces a fair amount of randomness in the substituted XML. Subsequently, the various swapping and XOR operations make the byte even more random. If 'a' is made into 'e' in the above example, then the encrypted XML with the same key is changed to *00100010 11100010 11100110*, which is significantly different from the previous encryption. Further, the Initialization Vector changes the final result for each new encryption done with the same key.

4.3.2.4 To Brute-force a block of byte level encryption

The size of a block is assumed to be 8-bit during byte level encryption (Figure B.2). Only 256 possibilities of this 8-bit block are possible. It seems easy to 'brute-force' a block. However, even after 256 different permutations (one correct byte with 255 wrong cases.), only a part of the substituted XML is found, and not even that of the plain XML. To get a part of the plain XML, the substituted XML has to be 'brute-forced' again.

The different possibilities of 'brute-forcing' a nibble (not the whole byte) are shown in Figure 4.4. There are infinite possibilities related to each nibble of the encrypted XML. Further, the entire encrypted XML is 'brute-forced' only when the correct value for each nibble in it

is obtained. The knowledge of the XML schema does help a little in brute force, and this is discussed in later subsections.

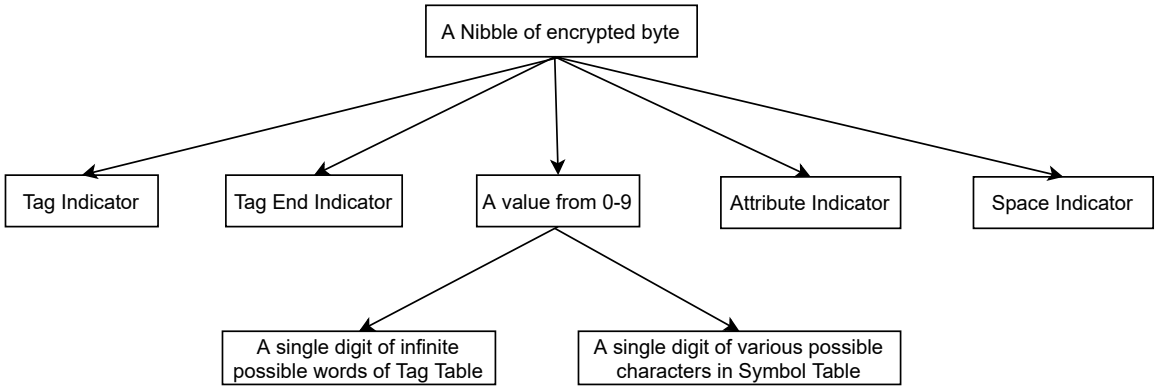


Figure 4.4: *Different possibilities when brute forcing a nibble of encrypted XML*

4.3.2.5 Intruder with XML schema tries to guess the encryption

The XML schema is often publicly available. In such cases, an intruder tries to get an idea of the various parts of the XML by looking at its encrypted form. For example, the binary of the encrypted XML corresponding to the XML in Listing 1 is shown below:

00100111 01111010 00100111 01100000 01011001 00001100 11001011 00010010 01000100 00000000 10001000 11000000 11000100 01001010 00000010 11111011 00100010 011110000 1011000 00001101 01101110 10011101

"0010" is the first nibble of the ciphertext. By assuming that it is an encryption of the tag start symbol (i.e 1011), the intruder tries to find this binary pattern (0010) in other places of the ciphertext. To do this, the binary pattern must either be the first 4-bits of the byte or the last 4-bits. Owing to shifting of bits and several XOR operations, 0010 appear at various places in the ciphertext that do not belong to the tag-start symbols. Even at positions where the ciphertext of the tag-start symbol is present, it may or may not be 0010. This binary pattern is found at 5 places in the ciphertext shown. The same happens with other binary patterns.

4.3.2.6 Intruder with XML schema tries to guess the data parts of the XML

The symbol table mostly contains numbers of only two or three digits. Therefore, if the XML schema is known, the start and end points of the data part of an encrypted XML can be easily

guessed. If the start and end points of the data part of the XML are known, the number of symbols in the data part of the plain XML can also be easily known. If the data in the plain XML is made up of a small number symbols, it can be 'brute-forced' and discovered.

There is a solution to this. Padding the data part (with nibbles representing "1010") in the ciphertext makes so big that it looks like having at least 17 different symbols in plaintext. If the size of the variable word in plaintext is less than 17, then padding is done to make it 17. 17 symbols is quite safe even when all symbols are digits in plaintext. Even if the intruder knows that all the symbols are digits, it will take 3 years to 'brute-force' 1 billion values per second. If more than one space is found in the Compressed XML during decryption at the receiver side, it is considered to be a single space.

For example, "iiti" only has 4 characters that are encrypted into 12 nibbles (if $key[8] = 3$). Depending on the value of $key[8]$ the result of the compression does some space-padding using 1010. It should appear to the intruder to be a 17 symbol data. If $key[8] = 3$ then three nibbles represent one plaintext symbol. Therefore 17 plaintext symbols require 51 nibbles in the ciphertext to be properly represented. In the case of "iiti" if $key[8] = 3$, we need $51 - 12 = 39$ extra padding bits of 1010. On the other hand, if $key[8] = 2$ then 2 nibbles in ciphertext represents one symbol in the plaintext. Therefore, 17 symbols in plaintext require 34 nibbles in ciphertext. In the case of our example, "iiti", with $key[8] = 2$, $34 - 8 = 26$ extra padding bits of 1010 are required.

If every variable part of an XML is encrypted in such a way that it appears to be of at least 17 symbols in plaintext, then it is very difficult to get the plaintext from ciphertext using 'brute-force' (even at the rate of 1 billion tries per seconds). Table 5.1 shows the time taken to 'brute-force' variable parts of different types.

Content of ciphertext	Brute Force Time
All 17 Symbols are digits	3 Years
All 17 Symbols are small alphabets	35953428 years
All 17 Symbols are capital letters	35953428 years
All 17 Symbols are alpha numeric	9.372428e+13 years
All 17 Symbols are ASCII characters	2.1074772e+19 years

Table 4.2: Brute force time depending upon type of symbols

4.3.2.7 No trivial relation between the key and ciphertext

A small change in the 10-element key results in a completely different ciphertext. A perfect example is shown in Table 4.3. A total of 15-bits change at the random points in the ciphertext of "iiti" because of a change in 1-bit of the *10-element key*. For even the slightest change in the key, a large number of bits are affected in the encrypted message. This is a desirable characteristic and attests the security provisioning capabilities of the proposed algorithm.

Plaintext	10-element key	Ciphertext in bits	**Bit difference from the key [12,6,1,1,1,14,4,1,3,2]**
iiti	[12,6,1,1,1,14,4,1,3,2]	010000101100101010 000010100001100000 100001000000	0
iiti	[12,6,1,1,1,14,4,0,3,2]	110000100100000010 100000101001000010 111011001110	15

Table 4.3: Relation between plaintext and key

4.4 Conclusion

In this thesis, a novel approach for XML and JSON encryption is proposed. The proposed method modifies existing XML compression algorithms to implement XML and JSON encryption. In addition to encrypting the document, the method is also compresses the same. In some ways, the proposed method is an efficient means of providing secure compressions. The efficacy of the approach is validated through a systematic analysis of the same against various possible security attacks both generic and specific to XML or JSON document. The anaysis clearly demonstrates the capabilities of the method to counter these attacks. In addition to this we also assess the compression efficacy of the method. It is similar to the approach proposed by us earlier in [58] and the method continues to provide superior compression ratios.

Chapter 5

Improving Privacy in Data Service Composition

To properly secure interactions between web services, concerted efforts are required to ensure security and privacy of web-service compositions. This is especially true for web-services that provision data more commonly known as data services. Ensuring security and privacy of such data is imperative as data that gets exchanged is often sensitive and critical. We look at this aspect of securing web-services in this chapter.

5.1 Understanding the Problem

Data of various types is available in a variety of domains, including medical databases, retail databases, demographic databases, and others. Clients can access and use this data, based on their specific application requirements, through web-service interfaces provisioned by the data owners. Typically, complex applications require information beyond what any single provider may own and offer; in such cases, multiple data services must be composed to provide a com-

plete solution meeting the client needs. The term "data-service composition" refers to a unified interface that invokes multiple data providers and synthesizes their data to deliver to the client all the data it needs in response to a single request message, as if it were available withing a single source.

U. Srivastava et. al. [55] formally define a “Service Composition Plan" as: “an arrangement of the web services in the query into a directed acyclic graph (DAG) $\mathscr{H}$ with parallel dispatch of data denoted by multiple outgoing edges from a single web service, and rejoining of data denoted by multiple incoming edges into a web service". Each query has its own service composition plan. A directed edge e_{ij} from a service provider *DSi* to *DSj* establishes *DSi* as the parent of *DSj* and implies that, while the plan is executed in response to a query, *DSi* precedes *DSj* and that *DSj* consumes the data produced by *DSi*.

An example service composition plan is diagrammatically depicted as a directed acyclic graph $\mathscr{H}$ in in Figure 1.1. In Figure 1.1, there are four data service providers (*DS1*, *DS2*, *DS3* and *DS4*) that provide details of a retail store. If a client requests information on “how many lunchboxes were purchased by people belonging to France on December 3, 2012", the execution of this query requires the involvement of all the four data service providers. The service composition plan for the query is shown in the Figure 1.1. *DS1* is invoked first with a “Lunch Box" description to retrieve the “Invoice" and "Quantity" of the purchases. Next, *DS2* and *DS3* are invoked with “Invoice" as the input to obtain “CustomerID" and “InvoiceDate" for the invoices corresponding to “Lunch Box". The outputs from *DS1*, *DS2* and *DS3* are joined together to produce a table *T1(“Invoice", “Quantity", “CustomerID", “InvoiceDate")*. *DS4* is then invoked with “CustomerID" as the input and results in *T2(“Custo-merID", “Country")*. T1 and T2 are joined to get *T3(“Invoice", “Quantity", “CustomerID", “InvoiceDate", “Country")*. Finally, the sum of the “Quantity” value in table T3 that corresponds to “Country” as France gives the final result to be returned to the invoking client.

The service-oriented architecture style dictates that providers, involved in a composition for the purpose of fulfilling a client’s needs, should not be aware of each other. Furthermore, data service compositions impose additional privacy-related constraints that further enforce this requirement [57]. More specifically, a service provider may encrypt the data it contributes to the service composition, in order to prevent other participants to gain any information. Alternatively, a service provider may allow the sharing of some attribute values, except in the case

of specific queries. Furthermore, a service provider may disallow the sharing of some attribute values with some specific service providers participating in the service composition plan. To support these requirements, it is essential that the participating service providers have a right to know the service composition plan, which should be shared before the execution of the plan.

Privacy is an important concern in data service composition: a service provider *DSi*, participating in a service composition plan, should be able to choose whether to hide one or more of its attributes from all, or some of, the other service providers in the composition plan. An attribute that a data service provider does not want to share is called a "privacy critical attribute".

Privacy is a major challenge to the smooth execution of a service composition plan. For example, in Figure 1.1, if “Invoice" is a privacy critical attribute of *DS1*, then this implies that *DS1* does not want to share its values with anyone else. However, according to the service composition plan, *DS1* needs to connect with service *DS2* and share the relevant invoice number (489436 in this example) in order to get the corresponding “customer ID” related to the lunchbox purchased with the above invoice. This sharing would be a direct infringement of the privacy of *DS1*, which raises the question: how can this service composition plan be executed?

An approach to address the issue of privacy in data service composition is through the use of privacy models such as K-Anonymity [56], and/or its variants such as L-Diversity [39] and T-Closeness [35]. K-Anonymity is explained as follows: in a service composition plan where *DSi* is the parent of *DSj*, a query is executed on *DSj* using as input information produced aa output of *DSi*. In such a situation, to preserve the privacy of *DSi*, the K-Anonymity approach prescribes that instead of querying for just the one attribute value of *DSj* corresponding to the output attribute value of *DSi*, *K* attribute values of *DSj* are queried. These *K* attributes include the value corresponding to the output of *DSi* and *K-1* other attributes. The parent (i.e., *DSi*) determines the value of K.

K-Anonymity can be explained using the example shown in Figure 1.1. *DS1 is queried for the “Invoice" of a lunchbox (invoice number 489436). Subsequently, database DS2 is queried for the “CustomerID" of the same “Invoice" (i.e., 489436). Conforming to the K-Anonymity approach, a query on the data service provider DS2 is not just executed for invoice number 489436 which is of interest here, but for K different values where K>1. For example in Figure 1.1, when K is 3, queries on DS2 are executed for 3 different values of the attribute “Invoice” (say 489435, 489436 and 489437), among which the actual invoice of interest,489436, is included. All the*

three values of "invoice" are such that they are present in the DS2 database. In such a scenario, therefore, DS2 knows that a certain value belongs to DS1 only with a probability of 1/K (in this example 1/3 as K is 3). DS1, that needs to protect the privacy of its "Invoice" data, determines the value of K. K-Anonymity is also referred to as K-Protection or the Value Generalization Method. Several research methods have been developed relying on K-Anonymity for privacy preservation and are discussed in greater detail in chapter 2.

Executing a data service composition involves several tasks such as, selection of service providers, development of the service composition plan, invocation of queries on the various data service providers, storing the intermediate query results, performing join operations on data received by different providers, and maintaining the privacy of service providers through value generalization. Responsible for the above tasks is a special service, called the mediator. Mediators can be trusted or untrusted. If a mediator is trusted, it can be used as a warehouse to place the intermediate results of the query in plain form. If a mediator is untrusted the providers' data is encrypted before it is stored in the warehouse. If the mediator is untrusted, there is little or no expectation that it should contribute towards privacy preservation. Most research today on mediators is towards improving the working of untrusted mediators.

Managing encrypted intermediate data is a major challenge with untrusted mediators because the mediator needs to handle encrypted intermediate data in queries or join operations without seeing the content. To overcome this, M. Barhamgi et. al. [45] developed OPES [3], an encryption method for numerical data in a manner that preserves the order of the encrypted values. This means that the comparison of two encrypted values will produce the same result as the comparison of the corresponding decrypted values. M. Barhamgi et. al. [45] use OPES for query execution, join operations, and deciding on the value of K for K-Anonymity. A major limitation with OPES is that it can only be used to encrypt numerical data. Using OPES, therefore, a service provider is able to secure only its numerical privacy-critical attributes. In this chapter, we propose to overcome this limitation by making use of hashing. Each parent-child pair in a service composition plan shares a secret string unknown to the mediator. This secret string is appended to each value of the privacy critical attributes and its hash (e.g., SHA-256) is calculated. As the secret string is the same for a parent-child pair, the hash of a certain value will be the same for both service providers. In this way, the hash values of non numerical data can also be compared lexicographically by the mediator without seeing their actual values.

Another major limitation of existing data service composition approaches is the assumption that the mediator is adhering to the service composition plan. Service providers do not, therefore, authenticate each other. They do utilise value generalizations (mostly using K-Anonymity) to protect data from each other by encrypting and sharing data with the mediator using OPES [3], but do not check whether the mediator is adhering to the shared composition plan. It is therefore possible that a mediator can share a false service composition plan and introduce a fake service provider in the composition. If a service composition, for example, consists of four services *DS1*, *DS2*, *DS3*, and *DS4*, and $DS1 \rightarrow DS2 \rightarrow DS3 \rightarrow DS4$ is the actual service composition plan. An unethical mediator can invoke another service provider *DS5* into the composition about which the remaining four service providers are unaware. Alternatively, an unethical mediator can also surreptitiously change the shared service composition plan (e.g. $DS1 \rightarrow DS4 \rightarrow DS2 \rightarrow DS3$) without the knowledge of *DS1*.

In this chapter, we describe a method for supporting data-service composition through an untrusted mediator, as follows.

1. During data service composition, the data of one service provider needs to be protected from other service providers. The mediator facilitates the maintenance of privacy between service providers in addition to taking on various responsibilities such as service selection, creation of the service composition plan, querying, joining, and communication between service providers. The mediator is untrusted and is not allowed to see any data; therefore, all data exchanges are encrypted.

2. Only numeric data can be secured using the current standard of OPES [3]. We propose concatenation of a secret string with data followed by hashing. Therefore, privacy critical attributes in our case do not necessarily need to be numeric. The proposed approach enables the mediator to perform operations such as join, query execution, or value generalization using K-Anonymity without seeing the actual data values.

3. It is the right of every service provider to review and agree on the service composition plan before participating in a composition. The service providers should be able to validate that they are participating only in the composition plan to which they have agreed. The service providers should be able to validate that their input data is originating from their parent service provider and that their output data is sent to their children service providers, based

on the agreed upon composition plan.

5.2 The Proposed Technique to Improve Privacy in Data Service Composition

We use a mediator-based approach for data service composition. We consider the mediator to be untrustworthy. Intermediate data is shared with the mediator in a secure manner such that the mediator can orchestrate the data composition plan, perform join operations on the collected data without accessing the data. Our approach is similar to that of M. Barhamgi et.al [45] but our method considers privacy-critical attributes of any type, not only numerical. Similar to [45], our methods uses K-Anonymity, more specifically the *"Dataset-based identifier generalization"* method discussed in [45], for value generalization to preserve the privacy of data service providers during data service composition. In addition, our approach also ensures authentication of the intermediate input and output data during data service composition. The service providers participating in the composition plan authenticate their respective parent(s) (the preceding data service in the composition plan) and child(ren) (the succeeding data service in the composition plan). The proposed approach also ensures that the mediator strictly adheres to the shared service composition plan and does not introduce illegitimate, external service providers.

In spite of being 'untrustworthy', it is the mediator's responsibility to orchestrate the data service composition. A client application queries the mediator and the mediator returns privacy sanitized results, without gaining access to intermediate privacy critical data. Algorithm shown in the Figure 5.2 describes the mediator's role in detail in data service composition; and Algorithm shown in the Figure 5.3 describes the part that the service providers play. The detailed steps followed by the mediator during the Service Composition are described in detail in the following subsections.

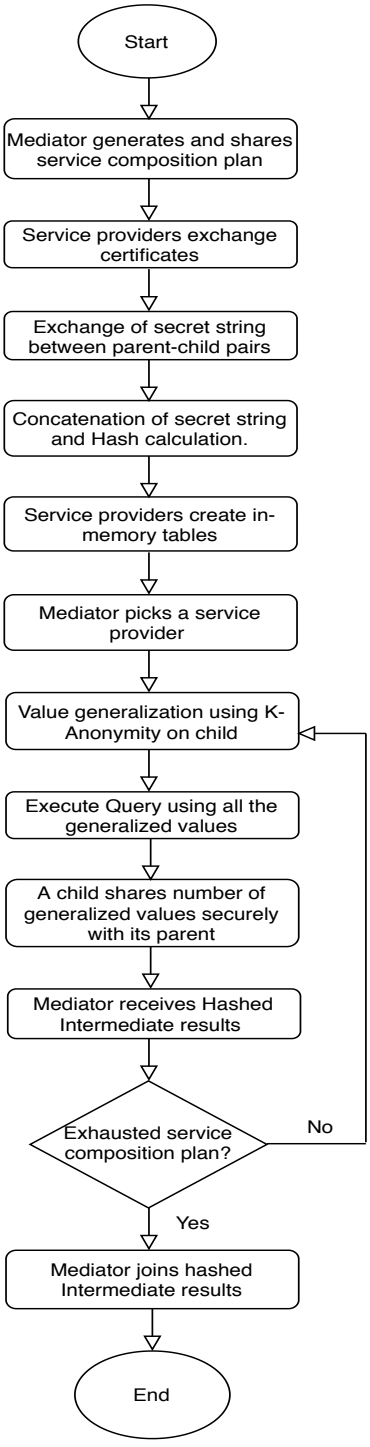


Figure 5.1: Steps in Privacy Preserving Data Service Composition

5.2.1 Mediator Generates and Shares Service Composition Plan

It is the responsibility of the mediator to prepare a service-composition plan to execute the client query (Steps 3 and 6 of Algorithm 5.2, Steps 1 and 2 of Algorithm 5.3). The mediator maintains a mapping table to keep a record of services along their properties such “service name", “input", “output", and the like (Step 1 of Algorithm 5.2). The mediator next selects a set of services from the mapping table to execute the requested query. The selection of services is based on their properties such as input, output, and content. The input query is broken into several sub-queries and the sub-queries are assigned to different service providers. This process is called query rewriting (Step 12 of Algorithm 5.2). In this chapter, our focus is only on improving privacy in data service composition and we do not dwell upon service selection and query rewriting. We adopt these from [14], [55], and [13].

Subsequent to service selection and query rewriting, the mediator decides the order of execution of sub-queries on the selected services. This ordering forms a directed acyclic graph of data service providers called the Service Composition Plan (Figure 1.1). An edge between two nodes *DSi* and *DSj* (nodes represent the service providers) in the Service Composition Plan indicates that the output of *DSi* is an input to *DSj*. For example, *In Figure 1.1 the input to DS2 and DS3 is the output of DS1*. We adopt the composition plan generation technique from Srivastava et. al. [55]. The mediator shares the Service Composition Plan with the selected data service providers (Step 6 of Algorithm 5.2).

5.2.2 Service providers exchange certificates

Service providers in a service composition plan exchange their respective authentication certificates with each other through the mediator (Step 7 of Algorithm 5.2, Step 2 of Algorithm 5.3). The mediator requests the X.509 certificates of all service providers in the composition plan [29] which is one of the possible certification mechanisms to authenticate the service providers in the composition plan. The mediator shares the certificate of each service provider with all the other service providers i.e. the certificate of each service provider *DSi* is shared with service providers *{DS1, DS2, ..., DSn}-{DSi}*.

5.2.3 Exchange of Secret Strings Between Parent-Child Pairs

All the parent-child pairs in the service composition plan share a common secret string between them (Steps 8, 9, and 10 of Algorithm 5.2, Steps 4, 5, 6, and 12 of Algorithm 5.3). The mediator facilitates the exchange of the secret strings between them, without it seeing the secret string. To create the secret string each parent service provider chooses a random secret string and encrypts it using the public key of its respective child(ren). This encryption is followed by another encryption with its own private key (i.e.,
$PRI_{parent}(PUB_{child}(SecretString))$). This doubly encrypted secret key is then passed on to the mediator that distributes it to the parent's child(ren). If a service provider happens to be playing the role of both a parent and child in the composition plan, it is associated with two secret strings: one that it receives from its parent and the other that it shares with its child(ren).

5.2.4 Concatenation of Secret String and Hash Calculation

All service providers append the secret strings of their respective parents to the entries of the privacy critical attributes (Step 13 of Algorithm 5.3). Next, an SHA-256 hashing of each value of the privacy critical attribute appended with the secret string is created. The same hash string is created for values that are present with the both the parent and child of the parent-child pair. The mediator does not know the attribute values nor the concatenations of the values and the secret string, but it is able to compare or execute join operations on the hashed values. Every new data service composition has a new secret string between a parent-child pair.

5.2.5 Service Providers Create In-Memory Tables

Each service provider maintains a table for its privacy-critical attributes (Steps 14 and 15 of Algorithm 5.3) and a second table comprising the "*hashed-value*" and the actual '*value*" of the privacy critical attributes. The *hashed-value*, as mentioned earlier, is created by concatenating the secret string from the parent with the value of the attribute and following it with applying

SHA-256 to it. This additional table for hash values ensures that the original table is untouched. For example, *in Figure 1.1 DS2 maintains an in-memory table comprising T(Invoice-Hash, Invoice)*. The table remains sorted (lexicographical sorting) with respect to the hashed-value (i.e., “InvoiceHash" in the case of DS1). When the mediator needs to execute a query for a value, it executes the same with the help of the hashed values (this is because the mediator only has hashed value). The service provider is able to know the actual value of the attribute with the help of tables in the memory and executes the query. There are several advantages of using an in-memory tables. First, the original table need not be sorted. Second, a service provider can maintain different in-memory tables for different composition plans. Different in-memory tables have hash values created, each with a different secret string. Third, the in-memory table protects our method from the privacy breech mentioned in detail in Section 5.2.7.

The use of the secret string is explained through the following example. In Figure 1.1, *DS1* encrypts a secret string, $SecretString_{DS1-DS2-DS3}$, with the public keys of *DS2* and *DS3* and passes these on to the mediator. The mediator shares these encryptions with *DS2* and *DS3*. All three service providers append the secret string to the values of their respective privacy-critical attributes such as “Invoice” and “CustomerID”. For instance, Invoice *“489434"*, becomes *“489434Secret"* if $SecretString_{DS1-DS2-DS3} = Secret$. Next, each service provider calculates the SHA-256 (hashing algorithm) values of their respective privacy-critical attributes appended with the secret string (e.g., *“489434Secret"* for the example invoice value). Each service provider maintains a sorted table in the memory comprising the privacy critical attributes and the hash values of their respective appended forms. in Figure 1.1 *DS1* maintains a table *T(InvoiceHash, Invoice)*, *DS2* maintains two such tables as *T(InvoiceHash, Invoice)* and *T(CustomerIDHash, CustomerID)*, where “*InvoiceHash*" is created using $SecretString_{DS1-DS2-DS3}$ and “*CustomerIDHash*" is created using $SecretString_{DS2-DS4}$. The mediator executes a query on the databases with the hashed appended values as the input instead of the actual attribute values. For example, the mediator queries *DS2* with the “*InvoiceHash*" as input instead the “Invoice" as input.

5.2.6 Mediator picks a service provider according to service composition plan

To execute an input query, the mediator executes a number of sub-queries on individual data service providers based on the service composition plan (Steps 12, 13, and 14 of Algorithm shown in the Figure 5.2). For example, *in Figure 1.1 a sub-query is executed on DS1 with "description" as input. Next, a sub-query is executed on DS2 by using the output of DS1 i.e* $Hash(Invoice + SecretString_{DS1-DS2-DS3})$. *A sub-query is also executed on DS3 using the same output of DS1. Finally, a sub-query is executed on DS4 using the output of DS2 as input,* $Hash(CustomerID + SecretString_{DS2-DS4})$.

5.2.7 Hashed Value Generalization using K-Anonymity

Value generalization is the process wherein the mediator queries a generalized (at least K values) set of values from a data service provider, instead of just querying one specific value of interest x (Steps 18 and 19 of Algorithm 5.2, Steps 16 and 17 of Algorithm 5.3). In other words, if a query is to be executed using an input value x on a Data Service Provider *DSj*, then generalization results in the query being executed using *K-1* values in addition to the query with x. All the K possible values belong to *DSj* and K is called the maximum protection factor required by the parents of *DSj* in the service composition plan (Figure 1.1). The mediator computes the generalization of a value with the help of *DSj* in such a manner that *DSj* is unable to know which of the K values is the value x for which the query needed to be really executed. This ensures privacy to the extent that *DSj* can only guess (with a probability of $1/K$ of being correct) the values of x.

Why Hashed Value Generalization? Assuming two service providers, *DSi* and *DSj*, where *DSi* is the parent of *DSj*. The mediator shares the service composition plan with all the participating service providers including *DSi* and *DSj*. If the mediator executes a query on *DSj* using an output from *DSi* (*DSj*'s parent), then *DSj* comes to know that *DSi* has information on this value. This is possible because both *DSi* and *DSj* have the hash for a common value. This infringes upon the requirement of privacy and needs to be avoided. Hence, *hash value generalization*

is used that results in *DSj* only being able to make a guess on the value of input data, with a probability of *1/K* of being correct. After a query is executed on a service provider with *K*, the mediator discards the *K-1* extraneous results and holds on to just the one actual result, by *filtering the false positives*.

Selectivity (Se) is an important factor that needs to be defined before a detailed description of Hashed Value Generalization is provided. Selectivity represents the number of output values from an in-memory table of a service provider over a given range. *Se(DSi, R)* is the number of output values in the range R of the in-memory table of *DSi*. For example, *the value of Se(DSi,* $[-\infty,+\infty]$*) is total number of elements in table whereas Se(DSi, [319e5, 6e093]) is total number of elements between "319e5" and "6e093".*

Hashed Value Generalization: For a given in-memory table of a child data service provider, the mediator follows the procedure below for hashed value generalization.

1. Mediator queries the selectivity of a service provider *DSi* with respect to the wide range of privacy critical values i.e Se(DSi, $[-\infty,+\infty]$) (i.e the whole table) along with the middle value value (say *mid*) in the range of $]-\infty,+\infty[$.

2. If the selectivity value returned is no less than *2*K*, and the *mid* value is lexicographically more than *x*, then step 1 is repeated with the new range $]-\infty, mid]$.

3. Else, if the returned selectivity value is no less than *2*K*, and the *mid* value is lexicographically less than or equal to *x*, then step 1 is repeated with the new range $[mid,+\infty[$.

4. Else, if the returned selectivity value is less than *2*K*, then the current range comprises the generalized values of *x*. Instead of just *x*, the service provider *DSi* is queried with all the values in the range.

5. The mediator removes every result except that of *x*. This is called the removal of false-positives.

We have largely adopted this method from - M. Barhamgi et. al. [45]. A limitation with their implementation is that it is susceptible to a privacy breach whereas we overcome it using an in-memory table and hashed values (discussed in 5.2.7).

Privacy Analysis of Hashed Value Generalization: According to [45] the only privacy issue with the method is that the result of generalization changes with a change in table. If the mediator does value generalization with a value (say x) twice from the same table with the data in the table being different the two times, then this results in value generalization with two different sets of data. However, one value remains common in both the results and that is the actual value being protected. This would enable the child service provider to guess that the common value is the actual value. The solution to this problem is that the result of value generalization should not change even when there is change in the table. To realise this, we use 'in-memory' tables to perform value generalization. The in-memory table remains unchanged for a given data service composition. Even when there is a change in the original table, the in-memory table remains unchanged for that data service composition. A new in-memory table is used for the next composition. Further, a child in a service composition plan shares with its parent the value of the number of queries executed on it by the mediator. This number, in an ideal scenario, is equal to K and represents the instructions given by the parent service to the mediator on the number of queries to be executed. If the value shared by the child is different from K, it indicates improper adherence to the K-Anonymity mechanism.

5.2.8 How authentication is achieved?

Hashed Value Generalization, as discussed in Section 5.2.7, only protects the value of a parent data service provider from the child data service provider in a service-composition plan. Authentication between two service providers is actually achieved using a secret string shared between the parent-child pairs (discussed in Section 5.2.3). Owing to the shared secret string between parent-child pairs, the mediator is not be able to introduce a service provider from beyond the agreed upon service-composition plan. If, in a service composition plan, service provider *DSi* is the parent of service provider *DSj*, then the following steps provision security between *DSi* and *DSj* and this security is denoted by *Sij*.

1. *DSi* picks a secret string and shares it with *DSj* by encrypting it using $PRI_{DSi}(PUB_{DSj}(SecretString))$.
2. Both *DSi* and *DSj* concatenate the secret string with the privacy critical attributes.
3. Both *DSi* and *DSj* compute the hash of the concatenation in the above step.

4. *DSj* shares the number of queries executed on it by the mediator with *DSi*.

Whenever the mediator executes a query on *DSi*, the same returns the result of the query to the mediator after securing it using *Sij*. By doing this, *DSi* and *DSj* authenticate each other in the following way:

1. The parent *DSi* gets confirmation that no one can use its output other than *DSj*.
2. *DSi* also gets to know whether the mediator is following the value of K or not.
3. *DSj* also gets confirmation that the input for which the value is being generalized using K-Anonymity, is coming from *DSi*. This is because all the hashed values in *DSj*'s in-memory table, on which value generalization is be performed, are secured by *Sij*.

5.2.9 How is everything working while maintaining privacy?

The operations of a mediator and the service providers in a service-composition plan are described in detail in Algorithms shown in the Figure 5.2 and 5.3 respectively. The following example describes the communication between these components done whilst maintaining security in Figure 1.1. In Figure 1.1 the security between *DS1*, *DS2* and *DS3* is denoted by *S123* for convenience of illustration. The three services share a common secret string because *DS2* and *DS3* are children of *DS1*. A query is executed on *DS1* with “Description" as the input. The result of *DS1* is Table T1[“Description", “S123(Invoice)", “Quantity"]. It is important to note that the mediator cannot see the value of “Invoice". The mediator performs value generalization (using K-Anonymity) of “S123(Invoice)" on both *DS2* and *DS3* based on the value received from *DS1*. It then queries *DS2* using the generalized values of “S123(Invoice)" as input. *DS2* creates table T2[“S123(Invoice)", “S24(CustomerID)"] for all the generalized values. Similarly, the mediator queries *DS3* using the generalized values of “$S123(Invoice)$” as input. *DS3* creates table T3[“S123(Invoice)", “InvoiceDate"] for all the generalized values. The mediator then performs value generalization (using K-Anonymity) of “S24(CustomerID)" on *DS4* based on the value received from *DS2*. Next the mediator queries *DS4* using the generalized values of “S24(CustomerID)" as input. *DS4* creates table T4[“S24(CustomerID)", “Country"]. In all the cases, the mediator eliminates the respective *K-1* data used for generalization and only retains

the actual data. Finally, the mediator joins T1, T2, T3, and T4 to get the final result of the query. A simplified sequence diagram of the proposed method having only one parent and one child is shown in the Figure 5.4.

5.3 Evaluation

In this section, the proposed service composition method is evaluated on the criteria of security, privacy, and performance. The privacy and security aspects of the proposed method are first evaluated by assessing the behaviour of the system under standard likely 'threat' circumstances. This is followed by comparing the proposed method with existing state-of-the-art techniques in terms of various aspects of security and privacy. Finally, the proposed method is evaluated in terms of its performance wherein the execution time and resource requirements are assessed and shown to be within reasonable bounds.

5.3.1 Dataset

To evaluate the proposed method for privacy in data service composition, we downloaded an online retail dataset from the UCI Machine Learning Repository (link of the dataset). There is only one table in this dataset that contains 1,067,371 records. The schema of this table is as follows: *R(Invoice, StockCode, Description, Quantity, InvoiceDate, Price Customer, ID, Country)*. From this table, we created four different schemas: *R1(Invoice, Description, Quantity)*, *R2(Invoice, CustomerID)*, *R3(Invoice, InvoiceDate, Price)*, *R4(CustomerID, Country)*. R1 includes the description and quantity of an invoice, R2 includes the customer identification associated with the invoice, R3 includes the price and date of an invoice, and R4 has information on the country that a customer belongs to. We create four tables corresponding to the four schemas.We insert 65K, 130K, 195K, 260K, 325K, 390K, 455K and 520K records in each of tables. That is, the first time that the query is executed 65 thousand records are inserted in each of the four tables, the next time 130 thousand records are inserted, and so on.

Algorithm : Mediator's steps of execution

Input: Query
Result: result_of_Query

```
1  var ServiceProvideres[] stores reference to all the
     service providers;
2  var query[] stores sub-queries of the original query;
3  Create Service Composition Plan;
4  Store reference of all the service providers in service
     composition plan in ServiceProviders[];
5  for Service_Provider s : ServiceProviders[] do
6  |   Share Service Composition Plan with s;
7  |   share certificate of s with everyone else in the
   |     ServiceProviders[];
8  |   if s is the first service provider in the composition
   |     then
9  |   |   for Service_Provider s1 : ServiceProviders[]
   |   |     do
10 |   |   |   Ask PRI_S(PUB_S1("secret")) from s
   |   |   |     and give it to S1;
11 |   |   end
12 end
13 Do query rewriting (i.e. divide the Query into
     sub-queries and store them in the array query[]);
14 for query q : query[] do
15 |   Find a service provider in ServiceProviders[] to
   |     execute q (Say SP);
16 |   if q does not have privacy constrained data from
   |     anyone else in ServiceProviders[] then
17 |   |   execute q;
18 |   else
19 |   |   Ask the value of K from the source of privacy
   |   |     constrained data of the query;
20 |   |   Do value generalization of privacy constrained
   |   |     data on SP;
21 |   |   Help SP to get confirmation from its parent
   |   |     (i.e source of privacy constrained data) about
   |   |     the value of K. Take the secured results of
   |   |     value generalization from the SP ;
22 |   |   Execute q on SP using all the results of value
   |   |     generalization;
23 |   |   Collect the secured results of query execution;
24 |   |   Remove false positive;
25 |   |   Help the SP to share the number of query
   |   |     executed on it with source of privacy
   |   |     constrained data ;
26 |   if q does not execute fine then
27 |   |   abort;
28 |   Do join operation if required;
29 end
30 return result_of_Query;
```

Figure 5.2: Mediator's steps of execution

Algorithm : Service provider's steps of execution

1 We refer the current service provider as SP;
2 Collect the service-composition plan from the mediator;
3 Store references of all the service providers (and their certificates) of the service-composition plan ;
4 Decide the value of K (its level of protection) for the value generalization using K-Anonymity and share it with the mediator;
5 **if** Mediator asks for a secret to be shared with others in composition plan **then**
6 | Select a secret string say "Secret";
7 | **for** All the service providers S in the service composition **do**
8 | | Share $PRI_{SP}(PUB_S(\text{“}Secret\text{”}))$ with mediator;
9 | **end**
10 **if** $PRI_{SP\_child}(PUB_{SP}(valueofK))$ received **then**
11 | **if** decided value of K matches with the received K **then**
12 | | send $PRI_{SP}(PUB_{SP\_child}(\text{“}OK\text{”}))$ to child via mediator;
13 | **else**
14 | | send $PRI_{SP}(PUB_{SP\_child}(\text{“}Abort\text{”}))$ to child via mediator;
15 Collect $PRI_{SP}(PUB_{SP\_parent}(\text{“}Secret\text{”}))$ provided by the mediator (Line 9 Algorithm 1) and decrypt it;
16 Suffix the privacy critical attributes with the string "ParentSecret" and calculate $Hash(PrivacyCriticalValue + Secret)$;
17 Create a table in memory as: $T(PrivacyCriticalAttribute, Hash(PrivacyCriticalValue + Secret))$;
18 Sort the table created in step 6 with respect to $Hash(PrivacyCriticalValue + Secret)$;
19 Help the mediator for value generalization based on $Hash(PrivacyCriticalValue + Secret)$;
20 Cache the result of value generalization;
21 Send $PRI_{SP}(PUB_{SP\_parent}(Count(\text{“result of value generalization"})))$ to parent via mediator;
22 Get confirmation from the parent;
23 **if** Confirmation from the parent is positive/ **then**
24 | Give the secured result of query executed by mediator as $Hash(result + Secret)$ for all the attributes in result;
25 **else**
26 | Abort the execution of query;

Figure 5.3: Service Provider's steps of execution

5.3.2 Experimental Setup

We simulate a data service-composition scenario by by providing the four schemas to four different data service providers to maintain, *DS1*, *DS2*, *DS3*, and *DS4*, correspondingly. The four services and a mediator are deployed as five standalone applications in Java on the same machine. The datasets are stored in a MySql server. All the processes described in Section 5.2 are implemented using appropriate Java APIs. For example, cryptography is implemented using packages “java.security", and “java.crypto". We run the experiments on an AMD Ryzen 7 3700U, 2.3GHZ processor running Windows 10 with 12GB RAM.

5.3.3 Security and Privacy Analysis

This section first examines how the proposed method overcomes certain standard security challenges in data service composition. We use sequence diagrams to depict the normal flow (Figure 5.4) of a composition and to depict the various threats that a data service composition has to face (Figure 5.5). These threats comprise unethical behaviour of the mediator. The idea of using sequence diagrams for these representations is taken from the AVISPA tool [11]. For simplicity, a scenario of one mediator and two data service providers is assumed where one service provider is the parent and the other is the child.

The threats depicted in the sequence diagram are described below along with a brief discussion on how the proposed method overcomes them.

- ***Threat:*** *The mediator attempts to insert an alien service provider into the composition*
  Resolution: The mediator is unable to do this because the latter is not aware of the common secret key shared between the services and therefore cannot share this with the alien service. The only way that the alien service can gain access to the secret key is by sharing its certificate with one of the members of the service composition.

- ***Threat:*** *The mediator attempts to surreptitiously change the service-composition plan by changes the relationships between the participating service providers, from Parent* $\rightarrow$ *Child* $\rightarrow$ *Alien to Parent* $\rightarrow$ *Alien* $\rightarrow$ *Child*

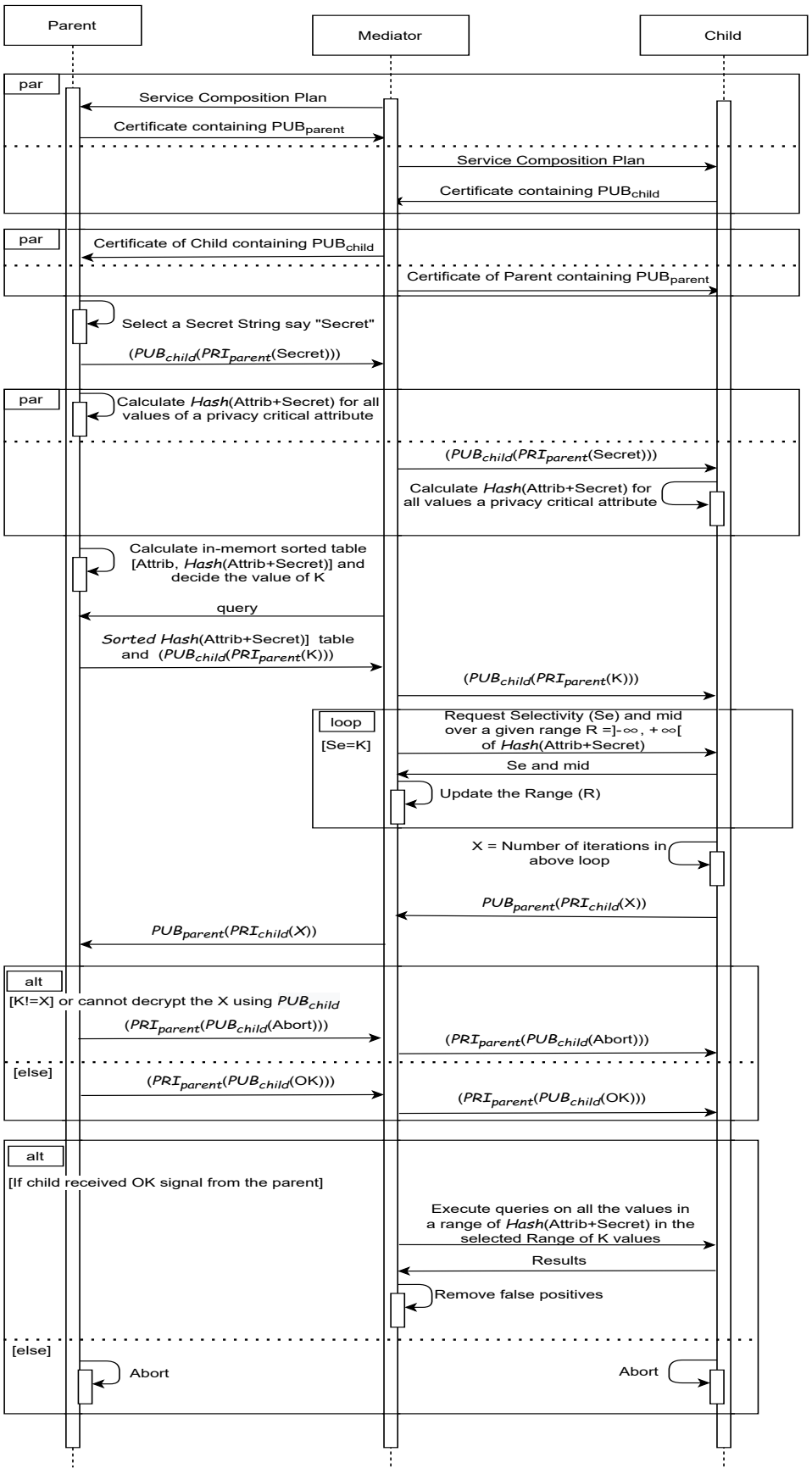


Figure 5.4: Sequence diagram of proposed data service composition with only two members

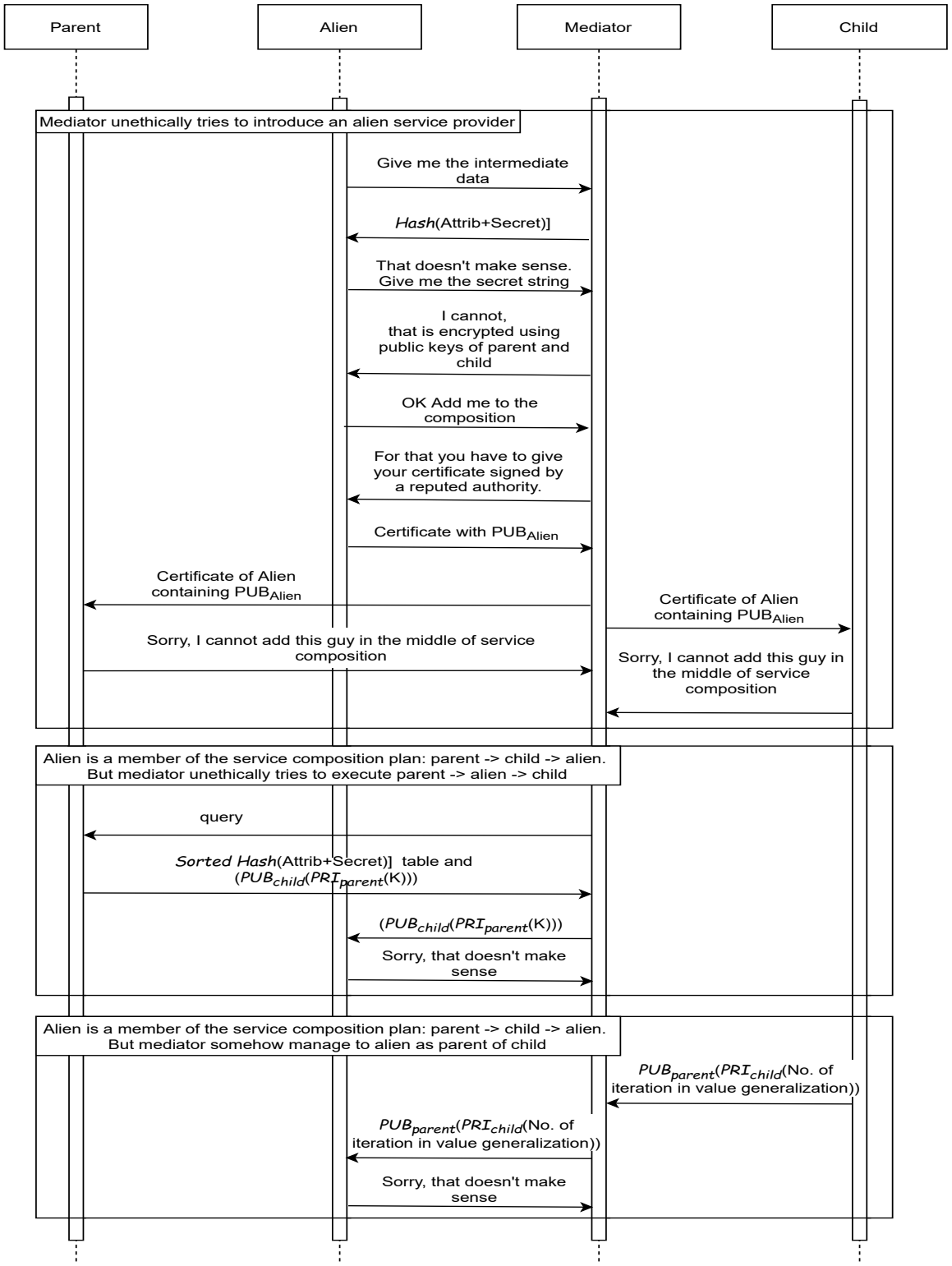


Figure 5.5: Security analysis of the proposed method having unethical mediator

Resolution: The legitimate Child of Parent, waits for a message (i.e $PUB_{Child}(PRI_{Parent}(K))$) from the Parent before supporting the mediator in value generalization. As the value is encrypted using Child's public key it does not make sense to Alien. Therefore the mediator is unable to change the service composition in this manner.

- ***Threat:*** *This is another scenario wherein the mediator attempts to surreptitiously change the service-composition plan in the following manner: here also, Alien is considered a legitimate member of the composition plan and the correct plan is Parent $\rightarrow$ Child $\rightarrow$ Alien. This is modified to Alien $\rightarrow$ Child by the mediator and this modification happens after the Child has received a message from the Parent and needs to respond. The Alien replaces Parent and becomes the new parent of the child.*

  Resolution: In this case, the Child still considers the Parent as its parent. The Child sends a message (i.e $PUB_{parent}(PRI_{child}($ No. of iteration in value generalization$))$) to the Parent after assisting the mediator with value generalization. As the value is encrypted using the Parent's public key it does not make sense to the Alien. Therefore the mediator is unable to change the service composition in this manner.

The vulnerabilities and threats discussed above are by no means comprehensive and represent a small sample of possible threats. They are included to demonstrate the working of the proposed method.

Apart from overcoming the threats described, the proposed method ensures complete privacy between communicating service providers. No message exchanged between these service providers gets leaked. The proposed method ensure this privacy through value generalization by the mediator. The only thing that the mediator is trusted with is the encrypted intermediate value of messages exchanged between service providers in the composition.

5.3.3.1 Comparison with existing techniques

Let us now compare our work, in terms security and privacy, against other related methods in terms of the functionalities they provide. The work proposed by Stephen S. Yau *et. al.* [61] on data service composition is quite similar to our method, with the exception of two main differences.

- Stephen S. Yau *et. al.* [61] use a unique random factor is between each parent-child pairs which is analogous to the Secret in the proposed method.
- The major difference between our method and [61] is that the latter does not do value generalization and it is not able to maintain privacy between two service providers.

S. E. Tbahriti *et. al.* [57] propose an approach that employs formal models for ensuring privacy. When two services lack compatibility in ensuring privacy, the approach prescribes that the mediator negotiate and make amends. This approach is quite different from ours in that there is no real cryptography involved.

The method proposed by M. Barhamgi *et. al.* [45] is the latest in the field of data service composition.

- The technique uses OPES [3] encryption instead of “Hashing with a common secret", to hide intermediate data from the mediator. OPES encrypts data in a manner that two encrypted values can be compared without the need to be decrypted. The limitation of this technique, however, is that it cannot be used for encrypting non-numeric data.
- Just like our method, this technique also ensures privacy between two service providers through value generalization.
- The absence of the system of a common secret in this technique makes it vulnerable to the illegitimate inclusion of an alien service into a shared composition plan by an untrustworthy mediator.
- Also, the absence of authentication messages between parent and child exposes the service-composition plan to illegitimate alteration by an mediator.

Table 5.1 summarises the endeavours of various approaches at ensuring security and privacy in data service composition.

5.3.4 Performance Analysis

To analyse the performance of the proposed method, we recorded the execution time of a query with and without the applying the proposed method on the experimental data service composi-

	Stephen S. Yau et.al. (2008)	SE Tbahriti et.al (2014)	M Barhamgi et.al (2019)	Proposed Method
Intermediate data is Visible to the mediator	No	Yes	No	No
Privacy between a parent-child pair in a service composition is preserved	No	No	Yes	Yes
Value generalization Used	No	No	Yes	Yes
Encryption of the Intermediate data	Yes	No	Yes	Yes
Common secret shared among participating services	yes	No	No	Yes
Supported type of Intermediate data	Any	Any	Integer	Any
Authentication between Service providers	Yes	No	No	Yes
Source service has assurance about destination of its output data	Yes	No	No	Yes
Source service has assurance about the proper use of K-Anonymity	No	No	No	Yes
Possibility of Replay attack by mediator	Yes	Yes	Yes	No
Trust on mediator	Average	Very High	Average	Low
Mutual trust among services	High	Very High	Average	Low

Table 5.1: Privacy features available with various methods in a data service-composition environment

tion set-up described earlier in this section. The query executed is: *how many lunchboxes were purchased by people of France on December 3, 2012?*. The execution of this query requires the involvement of all four service providers discussed in Section 5.1. The service-composition plan to execute this query is shown in Figure 1.1. In the performance analysis results reported, the *process time* graphs show the actual CPU time expended to complete the execution of a query, whereas the *elapsed time* graphs shows the total time interval between the start and finish of query execution.

Figures 5.6, and 5.7 show that irrespective of whether it is the elapsed time or the process time, the required execution time increases with the number of queries executed. As the value of K is large, a large number queries is executed and the time of execution is long. Further, while performing value generalization, the number records in the database plays an important role. The more the number of records there are in the database, the more time that value generalization takes for a given value of K. Therefore, for a given value of K, the execution time increases as the number of records increases. On the other hand if no privacy is incorporated, the execution time remains constant.

Execution time also depends on the number of privacy critical data used in service composition. A large number of privacy critical data results in more value generalization and hence a large number of queries that need to be executed. In Figure 1.1, DS1 has a privacy critical attribute “Invoice". Figures 5.8 and 5.9 show that a large number of outputs from DS1 results in longer execution time for a given value of K. The Execution time of the proposed method is better than M. Barhamgi *et. al.* [45]. Where to execute a query the method proposed by [45]

takes more than an hour in its most secure form, the same proposed method takes hardly few minutes.

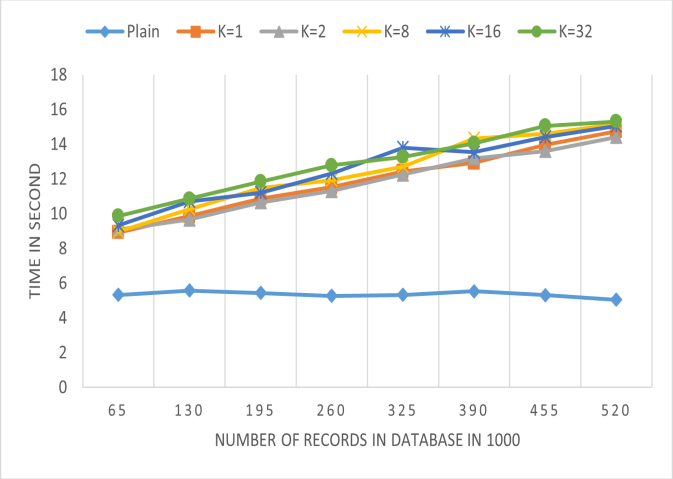


Figure 5.6: Process Time of Execution vs Number of Records (DS1 has 9 Records in Output)

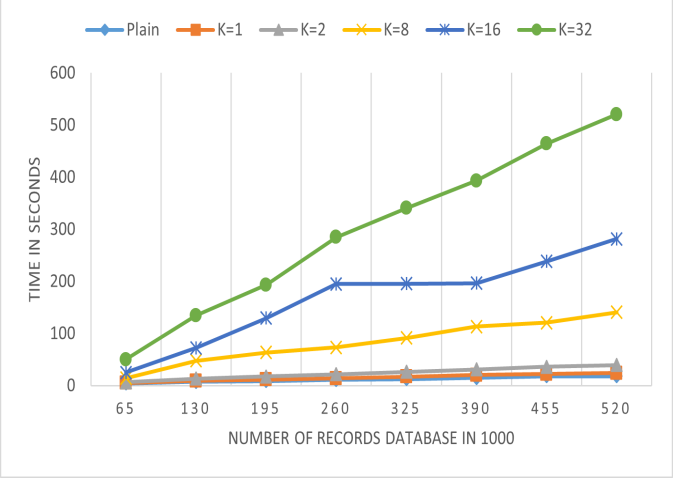


Figure 5.7: Elapsed Time of Execution vs Number of Records (DS1 has 9 Records in Output)

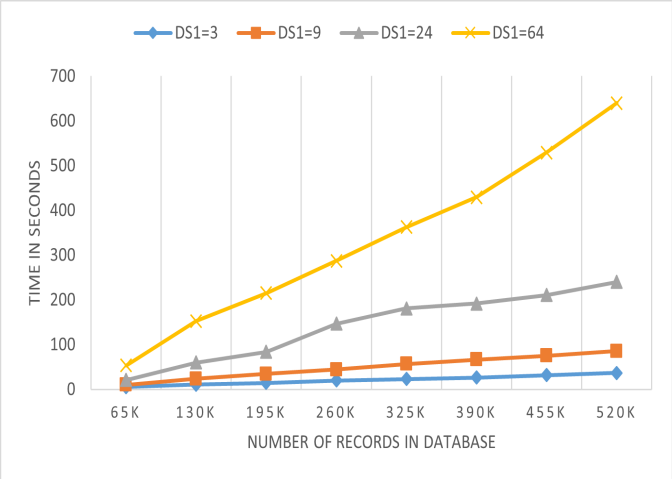


Figure 5.8: Elapsed Time of Execution vs Number of Records (K=4)

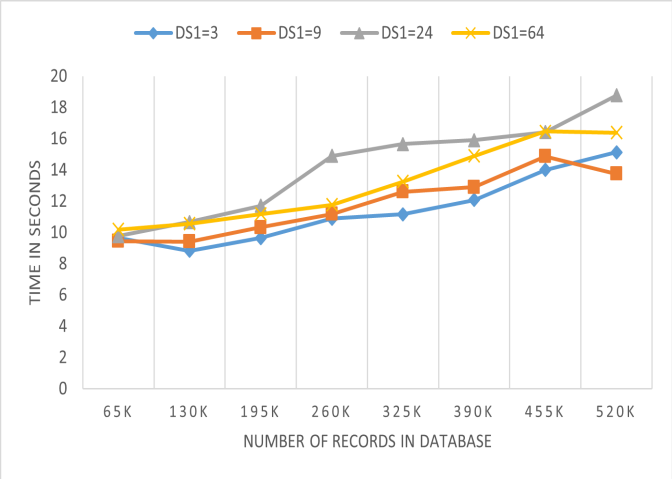


Figure 5.9: Process Time of Execution vs Number of Records (K=4)

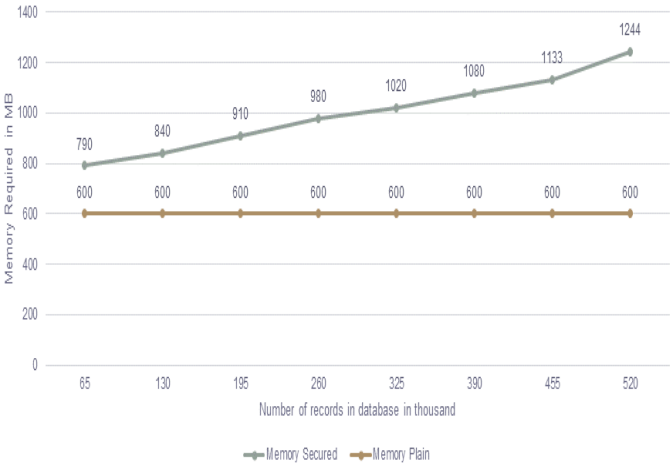


Figure 5.10: Memory consumption for query execution (K=4)

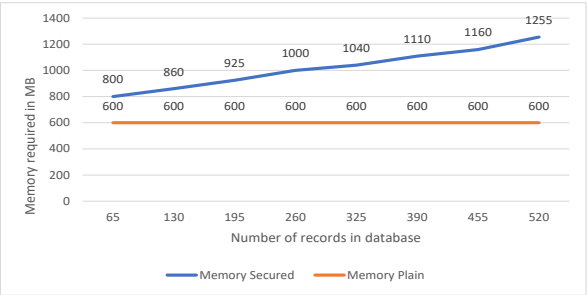


Figure 5.11: Memory consumption for query execution (K=16)

Therefore, the time required to execute a query depends on the following: 1) the number of actual execution instances, which is K times the number of privacy critical attributes; 2) the total number of service providers in the service-composition plan; 3) the total number of records in the database of each service provider, as the mediator performs a binary search on it for value generalization; and 4) the cost of the authentication process, which is a constant since it involves a fixed number of messages. If the number of privacy-critical attributes is P, a number of service providers in the service-composition plan is S, the average number of records per service provider is R, and K is the value of K in value generalization, then the time complexity of query execution is O(P*K*S + log(R)). This is the time complexity of the whole system of data service composition. The time taken to sort the in-memory table is not counted in the time complexity, as it is a separate one-time activity of each service provider. Each service provider also has to maintain a sorted table in memory, which contains all the records of that service provider. So each service provider has a space complexity of O(R).

The entire service composition set up is installed on the same Windows 10 system as discussed earlier. By executing the service composition, we are able to assess the total memory requirements. As is evident from Figures 5.10 and 5.11, if the value is not generalized, the memory requirements do not change with increasing number of records. With value generalization, however, an in-memory table needs to be maintained and therefore the memory requirements increase with increasing records.

5.4 Conclusion

This chapter proposes a method for data service composition that maintains privacy between data service providers. The method assumes an untrustworthy mediator and, relying on the *k-anonymity* algorithm for value generalization and basic authentication and encryption, it provides for effective privacy in data service composition, mitigating the risks of the mediator sharing participant data with external entities and violating the predefined and agreed upon service-composition plan. Our method meets the key privacy requirements of sensitive-data providers participating in data service compositions and its feasibility has been demonstrated through experimental evaluation with real-world data services.

This page was intentionally left blank.

Chapter 6

Summary and Conclusions

This thesis deals with the efficacy, privacy and security aspects of web-services and their composition. Most requests and responses between web-services are in the form of XML and JSON. To maintain effective interaction between web-services we propose new compression technique for XML and JSON documents. The proposed techniques provide compression ratio superior to existing state of the art techniques, especially those meant for web API based XML and JSON.

Subsequently, a novel XML and JSON encryption technique is proposed by extending the ideas of existing XML and JSON compressors. The proposed technique of encryption secures XML and JSON documents in addition to effectively compressing them. We conducted extensive security analysis of the proposed technique and analyzed the technique along varied security aspects. We also tested the technique against attacks that are especially meant for XML. We found the proposed technique to be secure in all these aspects.

We finally looked at security and privacy aspects of data services during compositions. This is especially important because this is one category of web-services that is especially vulnerable particularly in terms of data privacy. Data service providers are popular means of provisioning data and accept and respond to queries through web service interfaces. Often data

service providers have to form compositions amongst themselves to effectively execute a query. This phenomenon is called data service composition. The biggest challenge during data service composition is the privacy of the service providers. In this thesis, we provide a means for effective authentication, privacy and security between data service providers in a service composition plan.

6.1 Contributions

The points below depict main contribution of this thesis.

- In this thesis, we describe a new XML compression technique motivated primarily by the need to enable bandwidth-efficient communications within service-oriented systems and aims to reduce the size of the XML documents exchanged as requests and responses between web APIs. The size of these documents is typically small (less than 1 MB) and their content involves a high ratio of element and attribute tags to data content.
- By extending XML and JSON compression technique, we developed a novel XML and JSON encryption technique. The proposed encryption technique secures XML and JSON documents and compresses as well.
- The following are the contributions of this thesis in terms of data service composition. We describe a method for supporting data-service composition through an untrusted mediator, as follows.
  - During data service composition, the data of one service provider needs to be protected from other service providers. The mediator facilitates the maintenance of privacy between service providers in addition to taking on various responsibilities such as service selection, creation of the service composition plan, querying, joining, and communication between service providers. The mediator is untrusted and is not allowed to see any data; therefore, all data exchanges are encrypted.
  - Only numeric data can be secured using the current standard of OPES [3]. We propose concatenation of a secret string with data followed by hashing. Therefore, privacy critical attributes in our case do not necessarily need to be numeric. The

proposed approach enables the mediator to perform operations such as join, query execution, or value generalization using K-Anonymity without seeing the actual data values.

– It is the right of every service provider to review and agree on the service composition plan before participating in a composition. The service providers should be able to validate that they are participating only in the composition plan to which they have agreed. The service providers should be able to validate that their input data is originating from their parent service provider and that their output data is sent to their children service providers, based on the agreed upon composition plan.

6.2 Scope for future research

- For XML and JSON compression and encryption, coming up with appropriate compression techniques for binary data being transferred through XML or JSON is an important avenue for future research. In addition to this, the compression of XML or JSON documents using non-English characters needs to be worked upon and improved.
- There is plenty of scope to work towards reducing the time and memory required to execute data service composition.

This page was intentionally left blank.

List of Publications

International Journals

1. **G. P. Tiwary**, E. Stroulia, A. Srivastava. "Compression of XML and JSON API Responses. ", *IEEE Access. 2021 Apr 13;9:57426-39*, (Published)

2. **G. P. Tiwary**, E. Stroulia, A. Srivastava. "A Size efficient Encryption Technique for Structured Documents Like XML and JSON. ", *Under Prepration*, (In Press)

3. **G. P. Tiwary**, E. Stroulia, A. Srivastava. "Improving Privacy in Data Service Composition. ", *IEEE Access.*, (In Press)

This page was intentionally left blank.

Appendix A

Initialization Phase of XML encryption

A.1 Introduction

The sender and receiver make some initial settings before exchanging XML data, e.g. to create and share the key, creating *"Temporary Table"* and *"Symbol Table"* in the *"Initialization"* stage. XML encryption and decryption is done using a shared symmetric key among parties. It is assumed that a symmetric key can be securely exchanged between the sender and the receiver of the XML. We have chosen to call the shared key as *"10-element key"*.

A.2 10-Element Key and Temporary Table

In the proposed method of XML encryption, we have used a 59 bit symmetric shared key. The 59 bits present in the key are divided into 10 different parts discussed later in this section. In this way, we have called this key a *"10-element key"*. Therefore 10 integers of different range can be various elements of this key. For Example : *([12,6,1,1,1,14,4,0,3,2])* is a *"10-element*

	6	5	4	3	2	1
18	0	1	2	3	4	5
17	6	7	8	9	A	B
16	C	D	E	F	G	H
15	I	J	K	L	M	N
14	O	P	Q	R	S	T
13	U	V	W	X	Y	Z
12	a	b	c	d	e	f
11	g	h	i	j	k	l
10	m	n	o	p	q	r
9	s	t	u	v	w	x
8	y	z				
7						

Figure A.1: *Temporary Table (TT)*

key". The symmetric key is at the heart of the proposed method of encryption and comprises 10 elements. Here different elements of the *key* are referred to by using indexing starting from *0*, for example *key[0]* is 12, *key[1]* is 6 etc.

Using this key, both the sender and the receiver create two tables, *"Temporary Table"* (Figure A.1) and *"Symbol Table"*. (*"Temporary Table"* is discussed in the section A.2 whereas *"Symbol Table"* is discussed in the section A.3). The *"Temporary Table"* is created using only *"10-element key"* while *"Symbol Table"* is created using both *"Temporary Table"* and *"10-element key"*. The value of the *"10-element key"* determines the number of rows and number of columns in the *"Temporary Table"*. The *"Temporary Table"* will be filled with which characters and what will be the order of these characters in table also depends on the key itself. A fully prepared *"Temporary Table"* using the *"10-element key" ([12,6,1,1,1,14,4,0,3,2])* is shown in figure A.1. The coloured cells of the *"Temporary Table"* are used to store table headers, whereas the white cells are for normal entries. There are two types of headers in *"Temporary Table"* , one is a *"row_header"* and the other is a *"column_header"*.

Different element of the key has been discussed one by one in the points given below and its participation has been explained in the formation of *"Temporary Table"* and *"Symbol Table"*.

- **key[0]:** First 15 bits of the key i.e. *key[0]* specifies the number of rows in *"Temporary Table"*. The number of rows in *"Temporary Table" = key[0] + 1*.
- **key[1]:** Second 15 bits of the key i.e. *key[1]* specifies the number of columns in *"Temporary Table"*. The number of columns in *"Temporary Table" = key[1] + 1*.
- **key[2]:** As part of the encryption process, both *"row header"* and *"column header"* of *"Temporary Table"* need to be numbered. Numbering always starts with *1*. Numbering can start from either *"row header"* or *"column header"* depending upon the value of one

bit *key[2]*. The header numbering of *"Temporary Table"* starts with *"row header"* if the value of *key[2]* is *0* and it starts with *"column header"* if the value of *key[2]* is *1*.

- **key[3]:** If one bit *key[3]* is *0* then the *"row header"* numbering in *"Temporary Table"* starts from the top row (top to bottom) and if *key[3]* is *1* then the *"row header"* numbering starts from the bottom row (bottom to top).

- **key[4]:** If one bit *key[4]* is *0* then the *"column header"* numbering in *"Temporary Table"* starts from the the extreme left column (left to right) and if *key[4]* is *1* then the *"row header"* numbering starts from the extreme right column (right to left).

- **key[5]:** The key[5] is 6 bits in length. The non-header entries to be entered in *"Temporary Table"* are divided into 4 sets: "small English alphabet", "capital English alphabet", "digits", and "special symbols". The set "special symbols" contains characters from every language in the world except English. Different permutations of these sets will be entered in the *"Temporary Table"* for different values of the *key[5]*. Symbols of any one set will be entered in the *"Temporary Table"* according to their Unicode codepoints. The entry always start from the left top non-header cell of the *"Temporary Table"*.

- **key[6]:** After inserting different symbols into the *"Temporary Table"*, they are slightly shuffled into table using the *key[6]*. Various groups of consecutive elements in *"Temporary Table"* are created. Each group has a size indicated by *key[6]*. The last group of a *"Temporary Table"* can have fewer elements than *key[6]*. Figure A.1 shows the various groups as alternate consecutive underlined and non-underlined elements. The consecutive elements *0,1,2,3* (underlined) are in the same group and *4,5,6,7* (non-underlined) are in a different group.

- **key[7]:** If the value of one bit *key[7]* is 1, then all the elements of the group will be reversed in the respect of their old position in the group. If the value of *key[7]* is 0, there will be no such reverse operation. For *key = [12,6,1,1,1,14,4,1,3,2]* the groups are reversed and the corresponding *"Temporary Table"* is shown in Figure A.2.

- **key[8]:** The 3-bit *key[8]* has two purposes in this encryption process. It determines the number of digits of the numbers corresponding to characters in the *"Symbol Table"* (Figure 4.2). *key[8]* is also used as increment value of the initialization vector during the byte-level encryption and decryption (sections B.2.3 and B.3.1). Because the *key[8]*

decides the size of final encryption, therefore, its value depends on how much size the sender and receiver need for the encryption result.

- **key[9]:** The key[9] is 8-bits in length. For every non-header entry in Table-1, a number is calculated using this formula: $(rowheader)^{key[9]} + (columnheader)^{key[9]}$. Powering *column header* and *row header* with *key[9]* has a security benefit that: For the equation

  $(A)^n + (B)^n = C$

  By knowing a values of C its not possible to find A, B and n.

	6	5	4	3	2	1
18	3	2	1	0	7	6
17	5	4	B	A	9	8
16	F	E	D	C	J	I
15	H	G	N	M	L	K
14	R	Q	P	O	V	U
13	T	S	Z	Y	X	W
12	d	c	b	a	h	g
11	f	e	l	k	j	i
10	p	o	n	m	t	s
9	r	q	x	w	v	u
8	z	y				
7						

Figure A.2: *Temporary Table (TT) With Group Reverse*

A.3 Symbol Table Creation

After the creation of *"Temporary Table"* on both the sender and receiver sides, now it is the turn to create *"Symbol Table"* on both sides. The procedure for creating *"Symbol Table"* is discussed with an example. The *"10-element key"* (*[12,6,1,1,1,14,4,1,3,2]*) and the *"Temporary Table"* created using this *"10-element key"* (Figure A.2) have been used to create the *"Symbol Table"* here. The procedure for creating *"Symbol Table"* is discussed below:

1. From each *non-header* and *non-empty* cell of the *"Temporary Table"*, The character is taken and a unique integer corresponding to it is calculated as:
   $value = (row\_header)^{key[9]} + (column\_header)^{key[9]}$.
   For example *"Temporary Table"* in Figure A.2, character *'j'* has *row_header = 11* and *column_header = 2*. The calculated $value = (11)^2 + (2)^2$. Therefore the *value* corresponding to *'j'* in *"Temporary Table"* is *125* in *"Symbol Table"*.

2. The number of digits in *value* is made the same as the value of *key[8]*. In doing this, *diff = key[8]-(no. of digits in value)* is first calculated and the procedure advances to Step 3.
   For example if *key = [12,6,1,1,1,14,4,1,3,2]* (*key[8]* is *3*) then *diff = 3-3 = 0*.

3. The final value that is to be inserted in *"Symbol Table"* corresponding to the given *non-header* character of *"Temporary Table"* is calculated as follows: $final\_value = value \times 10^{diff}$.
   For example *final_value* of *'j'* is $125 \times 10^0 = 125$.

4. The final record to insert in the *"Symbol Table"* is the *(character, final_value)* pair. If a record in the *"Symbol Table"* exists with the same *final_value* then the procedure advances to Step 5 otherwise to Step 6 .
   For example *('j',125)* is inserted in *"Symbol Table"*. Before insertion of this record, it needs to be checked if a record in *"Symbol Table"* exists with the same *final_value* (i.e 125). For the running example, it is assumed that no such record exists, and the procedure advances to Step 6. If *('j',125)* is already in *ST* and *('o',125)* is to be inserted, the procedure advances to Step 5.

5. When a record with the same *final_value* already exists in the *"Symbol Table"*, the *final_value* of the record is incremented by *1* and checked for its uniqueness. This is continued until a unique *final_value* is found. Subsequently, Step 6 is executed.
   For example before inserting a record for character *'o'*, its *final_value* is increased to *126*. If a record with *final_value 126* also exists then *('o',127)* is inserted, otherwise the record is *('o',126)* gets inserted in the *"Symbol Table"* and procedure advances to Step 6.

6. The record *(character, final_value)* is inserted in *"Symbol Table"*.

7. Both the sender and the receiver delete the *"Temporary Table"* after creation of the *"Symbol Table"*.

A subset of *"Symbol Table"* is shown in Figure 4.2.

This page was intentionally left blank.

Appendix B

Sender's and Receiver's Process of XML Encryption

B.1 Introduction

As discussed in the Appendix A both the sender and receiver have executed the Initialization stage and have the same *"10-element key"* and the *"Symbol Table"*. Before sending the XML, the sender goes through three steps of encryption as shown in the Figure 4.1. All the three steps have been talked about one by one in this section. **Further in this document the *"data"* and the *"structure"* parts of XML are together addressed as *"words"*.**

B.2 The Sender's Process (The Encryption Process)

B.2.1 Substitution

Substitution converts whole XML in a document of just 15 symbols (i.e. 0 to 9, T (tag start), A (attribute name), "SPACE", E (end of latest opened tag)). We choose to call this document as the *"Substituted XML"*. In the **XML1** (Listing 4.1) the order of occurrence of the words are *"root"*, *"attr1"*, *"value1"*, *"attr2"*, *"value2"*, *"name"*, *"iiti"*, *"/name"*, *"value"*, *"2"*, *"/value"* and *"/root"*. During the substitution process, *"words"* of the XML document are taken in the same order in which they appear in the document. The substitution process is described in the Algorithm 3.

In order to convert XML to *"Substituted XML"*, the sender passes the XML as an input to algorithm 3 and executes algorithm 3. The sender always substitutes the characters of the *"data"* parts of an XML to numbers using the *"Symbol Table"* . The first time the sender finds a *"structure"* part in the XML that does not have an entry in the *"Tag Table"*, the sender first substitutes the characters of such *"structure"* parts to numbers using the *"Symbol Table"*, as well as it calculates the *"Tag Table"* mapping of new *"structure"* part. But if a *"structure"* part of an XML has already an entry in the *"Tag Table"*, then the sender substitutes it to a number using the *"Tag Table"*.

The *"Substituted XML"* is made up of only 15 types of symbols:

1. Digits from 0 to 9
2. SPACE
3. Tag indicator (T)
4. Attribute name indicator (A)
5. Tag end indicator (E)
6. Binary data indicator (B)

Algorithm 3: Substitution_Algorithm

Input: Plain_XML

Result: Substituted_XML

```
1  var substituted_xml;
2  var substituted_part;
3  var sum;
4  for All the words in Plain_XML do
5      if The word is a “data" part of the XML then
6          substituted_part ← Call Symbol_Table_Based_Substitution_Algorithm with
             word as input;
7      if The word is a “structure" part of the XML then
8          if The “structure" part has an entry in the Tag Table then
9              substituted_part ← Number Corresponding to this word in Tag Table;
10         if The “structure" part does not have an entry in the Tag Table then
11             substituted_part ← Call Symbol_Table_Based_Substitution_Algorithm
                 with word as input;
12             sum ← Call Find_Number_Equivalent_Of_Word Algorithm with
                 substituted_part as input;
13             Insert (“word", “sum") entry in the Tag Table;
14         if The word is a “tag" in the XML then
15             substituted_part ← ‘T′ concatenate(+) substituted_part;
16         if The word is a “attribute-name" in the XML then
17             substituted_part ← ‘A′ concatenate(+) substituted_part;
18         if The word is a “value" in the XML then
19             substituted_part ← substituted_part;
20     if The word is a “Tag Closing" then
21         substituted_part ← ‘E′;
22     substituted_xml ← substituted_xml + SPACE + substituted_part;
23 end
24 return substituted_xml;
```

The algorithm 4 describes steps to substitute a word of XML with numbers using the *"Symbol Table"*. This algorithm converts the input word of XML into a string of numbers using *"Symbol Table"* and returns that string of numbers. Algorithm 3 calls the algorithm 4 for all *"data"* words in XML and Algorithm 3 also calls algorithm 4 for all *"structure"* words whose entries are not yet present in the *"Tag Table"*. The variable *STEnc* present in the algorithm 4 is to store the final result of the algorithm whose initial value we have assumed to be *null*.

Algorithm 4: Symbol_Table_Based_Substitution_Algorithm

Input: Word

Result: Symbol Table Based Substitution of Word

```
1 var word ← Word;
2 var STEnc ← null;
3 for All the characters in word do
4 |   STEnc ← STEnc concatenate(+) Number corresponding to this character in
  |     "Symbol Table" ;
5 end
6 return STEnc;
```

Algorithm 5 finds *"Tag Table"* entry for *"structure"* words. Algorithm 5 calculates the minimum *"Tag Table"* number for the passed word and returns it to the algorithm 3. The Algorithm 3 further stores the (word, number) entry in the *"Tag Table"*.

In order to understand the substitution process better, it is assumed that the *"10-element key" key = [12,6,1,1,1,14,4,1,3,2]* already has been shared between sender and receiver and a *"Symbol Table"* shown in the figure 4.2 already has been created at both sides. It is also assumed that the entry of the *"data"* part *"root"* of **XML1** is currently not present in the *"Tag Table"*. Algorithm 3 is executed with XML1 (Listing 4.1) as input. For the first word of the **XML1** (i.e *"root"*) the flow of algorithms 3, 4 and 5 goes as below:

1. *"root"* is a *"structure"* part of the XML (step 7, Algo 3).
2. We have already assumed that the entry of any *"structure"* part is not in the *"Tag Table"* (step 10, Algo 3).
3. The algorithm 3 calls the algorithm 4 with *"root"* as input (step 11, Algo 3).

Algorithm 5: Find_Number_Equivalent_Of_Word

Input: $String\_of\_Number$

Result: Number equivalent of Given Word in Tag Table

1 var $word \leftarrow String\_of\_Number$;

2 var x stores the number of entries already present in the Tag Table;

3 var y number of digits required for this Tag Table entry;

4 var *sum* Number equivalent of Given Word in Tag Table;

5 var $diff$;

6 Divide the *word* into *key[8]* equal parts and covert all of them into integers;

7 Add all the integers found in the previous step and store the result of this addition in the variable *sum*;

8 $x \leftarrow$ "the number of entries already present in *Tag Table*" + 1;

9 $y \leftarrow \log_{10}(x+1)$;

10 $diff \leftarrow$ *y-(no. of digits in sum)*;

11 $sum \leftarrow sum \times 10^{diff}$;

12 **if** A record is already present in "Tag Table" with same value of "sum" **then**

13 Keep on increasing the value of *sum* as $sum \leftarrow (sum+1)\%\ 10^{y}$ until *sum* gets a unique value other than *0*;

14 return *sum*;

4. The word *"root"* consists of four characters *'r'*, *'o'*, *'o'* and *'t'*. The *"Symbol Table"* entries corresponding to these characters are *'117'*, *'126'*, *'126'* and *'104'* respectively .

5. These numbers get concatenated as *"117126126104"* (step 3 to step 5, Algo 4).

6. Algorithm 4 returns *"117126126104"* to algorithm 1 which will be stored in the variable *substituted_part* (step 11, Algo 3)).

7. Algorithm 3) calls algorithm 5 as *"117126126104"* as input (step 12, Algo 3)).

8. Divide the string *"117126126104"* in key[8]=3 characters each. i.e *117*, *126*, *126*, and *104* (step 6, Algo 3).

9. The value of the variable *sum* is: *sum = 117 + 126 + 126 + 104 = 473* (step 7, Algo 5).

10. *x* = number of entries already present in the *Tag Table* (*0*) + 1 = *1* (step 8, Algo 5).

11. $y = \log_{10}(1+1) = 1$ (step 9, Algo 5).

12. *diff = 1-3 = -2* (step 10, Algo 5).

13. $sum = 473 \times 10^{-2} = 4$ (step 11, Algo 5).

14. If another entry were indeed present in *"Tag Table"* with *sum = 4*, then the next value of *sum* would be $4+1=5$ (step 12 and 13, Algo 5).

15. Because no other record is present in the *"Tag Table"* with a value of *4* therefore algorithm 3 returns the value 4 ((step 14, Algo 5)).

16. The record *(root, 4)* is stored in *"Tag Table"*((step 13, Algo 3)).

17. *"root"* is a tag Therefore "T" will be prefixed in the variable *substituted_part*. New value of variable *substituted_part* is *"T117126126104"* (step 13, Algo 3).

18. *E* is used to represent the closing of a tag (step 21, Algo 4).

19. The variable *result_of_substitution* gets updated (step 22, Algo 3).

Like this, the algorithm 3 execute for other words of the XML. If it is assumed that no *"structure"* part was already present in the *"Tag Table"*, then the final result of the substitution will be this :

result_of_substitution = "T117126126104 A153104104117340 850153137820146340 A153104104111 850153137820146349 T11615310 122122104122 E T850153137820146 349 E E". At the same time, *"Tag Table"* entry for each *"structure"* part will also be created. The *"Tag Table"* is created along with substitution as shown in the Figure 4.3. The *"Tag Table"* stores the agreed upon integers corresponding to each *"structure"* part of the XML.

On the other hand if it is assumed that the entry of all the *"structure"* parts were present in the *"Tag Table"*, then the final result of substitution will be something like this. *result_of_substitution = "T4 A8 850153137820146340 A9 850153137820146349 T5 122122104122 E T6 349 E T116850 153340 E E"*.

B.2.2 Compression

The result of the *"Substitution_Algorithm"* (i.e the *"Substituted XML"*) is a document always made up of 14 symbols. Therefore every character present in the *"Substituted XML"* can be represented with just 4-bits. Using this property, *Compression_Algorithm* (i.e the Algorithm 6) compresses the *"Substituted_XML"* to create another document we choose to call *"Compressed XML"*. The *"Compressed XML"* is a binary document.

For example, in the message (*"T4 A8 850153137820146340 A9 850153137820146349 T5 122122104122 E T6 349 E T116850 153340 E E"*.), "T" and "4" can be collected together in a byte as "10110100" as shown in the figure B.1. This is how two symbols are represented in a single byte and that is how the above-encrypted data is represented in just 46 bytes. The size is, in fact, much smaller than the actual size of the XML (i.e 78 characters). After incorporating all the symbols of the encrypted data byte pairs if one last symbol remains unaccommodating, then it will be paired with "0000". The odd one remaining is always the end of the root tag (i.e. *E*) (i.e "1111"). Therefore it assumed that a byte "11110000" also denotes the end of the XML (Figure B.1). Here we assume each number of the symbol table to be three digits. If the symbol table was to map each character with a number of two digits, then this result would have been even smaller.

Algorithm 6: Compression_Algorithm

Input: Substituted XML

Result: Compressed XML

```
1  var compressedXml ← null;
2  Consider all characters between 0 and 9 in the document of 14 symbols as a 4-bit
     binary between 0000 and 1001 ;
3  Consider the SPACE in the document of 14 symbols as a 4-bit binary 1010 ;
4  Consider the Tag Start Symbol in the document of 14 symbols (i.e the prefix T) as a
     4-bit binary 1011 ;
5  Consider the Attribute name Symbol in the document of 14 symbols (i.e the prefix A)
     as a 4-bit binary 1100 ;
6  Consider the Closing of a tag in the document of 14 symbols (i.e the E) as a 4-bit
     binary 1111 ;
7  var cursor ← First_Symbol_in_the_Substituted_XML;
8  for cursor is not end of "Substituted XML" do
9      if A character at cursor + 1 exists then
10         var cursor1 = cursor + 1 (i.e. next symbol);
11         Accommodate 4-bit binaries equivalent of symbols pointed by cursor and
             cursor1 into a single byte;
12         Concatenate the created byte with current value of compressedXml;
13     if If cursor + 1 is end of the "Substituted XML" then
14         Create a byte using the 4-bit binary of current symbol and 0000;
15         Concatenate the created byte with current value of compressedXml;
16     cursor = cursor + 2;
17 end
18 return compressedXml;
```

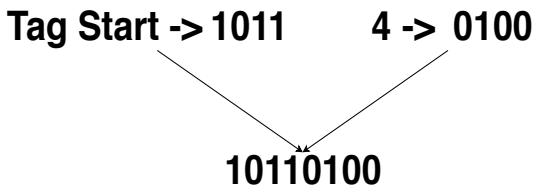


Figure B.1: *Accommodating Two Symbols in One Byte*

B.2.3 Byte level encryption of the Compressed XML

The substitution (section B.2.1) and compression (section B.2.2) stages of encryption introduce several security benefits (Chapter 4). Important security features that need to be incorporated in addition to the ones mentioned include “Diffusion" and “Confusion" [5]. Diffusion hides the relationship between the ciphertext and plaintext, whereas Confusion hides the relationship between the ciphertext and the key. The encryption procedures discussed until this point are all substitution ciphers that cannot provide confusion and diffusion effects. Apart from this, the documents “Substituted XML" and “Compressed XML" reveals the nature of the “Plain XML", i.e which part of the document is a tag, which part is an attribute name etc.

In order to make encryption more unpredictable and add “Diffusion" and “Confusion" properties to final encryption, we introduced byte level encryption as the third stage of proposed method of XML encryption. The same shared key 10-element key (for example *[12,6,1,1,1,14,4,1,3,2]*) is used for the byte level encryption. Steps of byte level encryption is described in the Algorithm 7.

The process of byte level encryption is depicted in the Figure B.2. Encryption is applied to every byte of “Compressed XML". Considering the first byte to be encrypted through byte level encryption is “00000100". Subsequent to the steps 9, 10, 11 and 12 of algorithm 7 byte “00000100" becomes “00100000". Rest of the byte level encryption procedure is shown in the figure B.3.

The permutation is followed by an XOR operation with an earlier byte of the encrypted stream (Step 20, Algo 7). For the first byte, same XOR operation happens with the least significant 8-bits of the second element of the key (i.e. *key[1]*) (Step 14 and 15, Algo 7). This is followed by an XOR operation on the byte with the least significant 8-bits of the first element of the key (i.e. *key[0]*) (Step 16 and 21, Algo 7). Figure B.3 shows byte level encryption of one

Algorithm 7: Byte_Level_Encryption_Algorithm

Input: Compressed XML, 10-element key

Result: Encrypted XML

var $encrypted\_Xml \leftarrow null$;

var $byte1 \leftarrow null$;

var $XOR1 \leftarrow null$;

var $XOR2 \leftarrow null$;

var $priviousByteEncryption \leftarrow null$;

for every byte of the "Compressed XML" **do**

  $byte1 \leftarrow byte\_selected\_in\_current\_iteration$;

  If *key[2]* is 1 then the first bit of $byte1$ is swapped with the eighth bit. And if *key[2]* is 0 then there will be no such swapping. ;

  If *key[3]* is 1 then the second bit of $byte1$ is swapped with the seventh bit. And if *key[3]* is 0 then there will be no such swapping. ;

  If *key[4]* is 1 then the third bit of $byte1$ is swapped with the sixth bit. And if *key[4]* is 0 then there will be no such swapping. ;

  If *key[7]* is 1 then the fourth bit of $byte1$ is swapped with the fifth bit. And if *key[7]* is 0 then there will be no such swapping. ;

  if The byte1 is the first byte of the "Compressed XML" **then**

    $priviousByteEncryption \leftarrow$ "eight least significant bits of $key[1]$";

  $XOR1 \leftarrow$ "$priviousByteEncryption$" $\oplus$ $byte1$;

  $XOR2 \leftarrow$ "$XOR1$"$\oplus$ "eight least significant bits of $key[0]$";

  $encrypted\_Xml \leftarrow encrypted\_Xml$ concatenates(+) $priviousByteEncryption$;

  $priviousByteEncryption \leftarrow XOR2$;

end

return $encrypted\_Xml$;

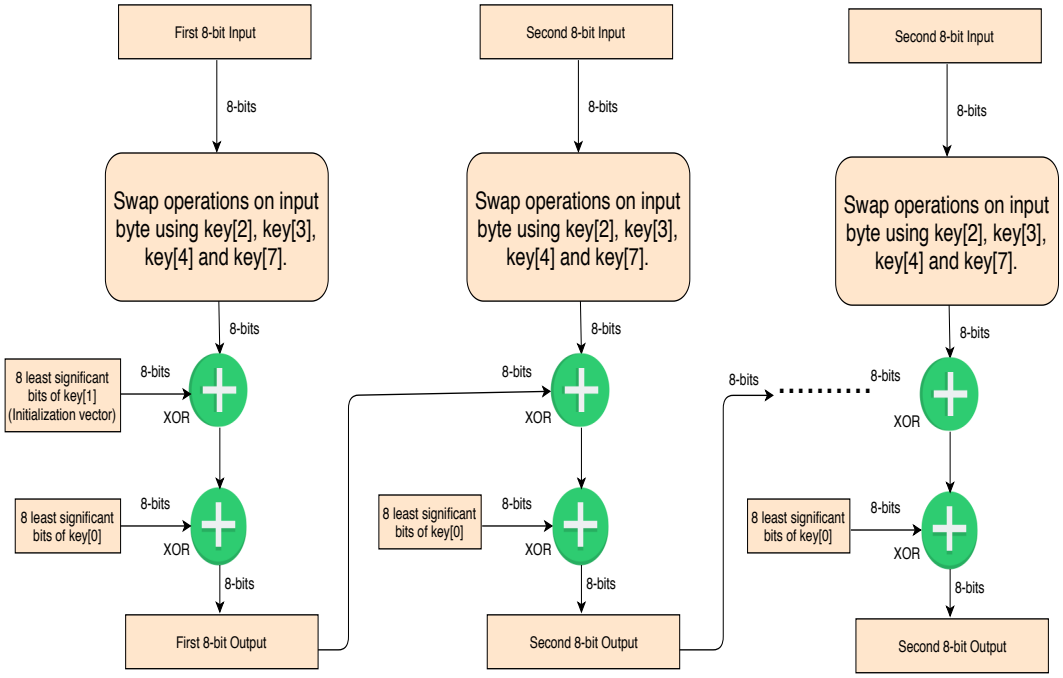


Figure B.2: *Byte Level Encryption Procedure*

byte (i.e first byte). In this case *key[0]* is 12 (i.e 00001100), *key[1]* is 6 (i.e 00000110).

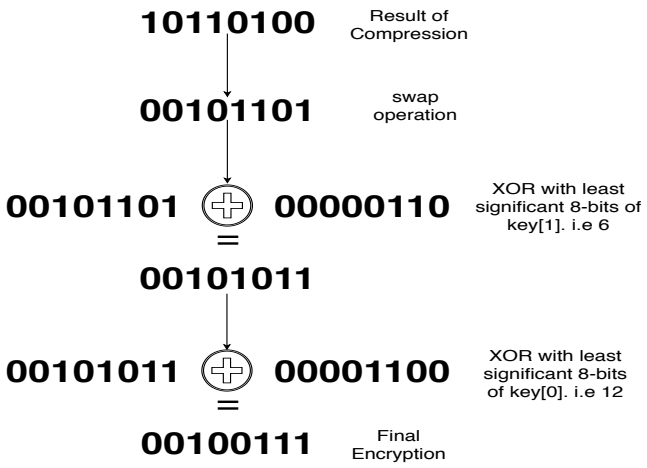


Figure B.3: *Example of Byte Level Encryption (First Byte of Message)*

In this case, CBC mode of encryption is used where the least significant byte of the *key[1]* is used as “Initialization vector (IV)" (nonce) [40]. If there is continuation of the conversation between receiver and sender with the same key then the next “Initialization vector (IV)" will be "Previous Initialization vector"+ key[8]. Therefore in the proposed approach the "Initialization Vector" gets created using the secret key and not known to the outside world. Other implementations may apply other modes of encryption [21].

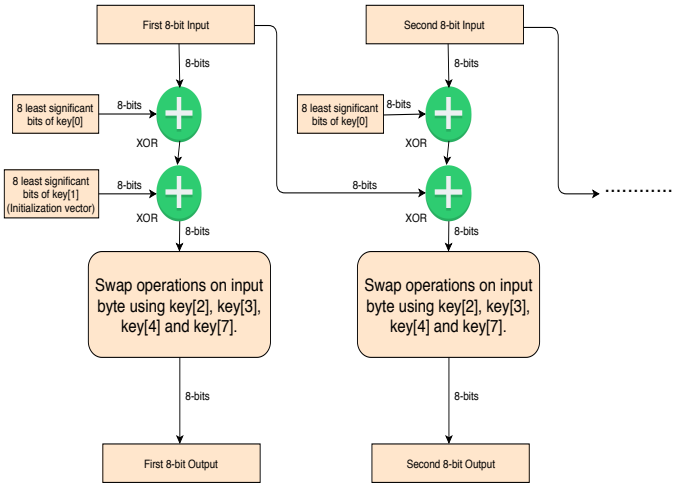


Figure B.4: *Byte Level Decryption*

B.3 The Receiver's Process (The Decryption Process)

Encrypted XML now accessed by the receiver. To read an encrypted XML it is necessary to decrypt it. For decryption, the receiver goes through three steps of decryption as shown in the figure 4.1. All the three steps have been talked about one by one in this section.

B.3.1 Byte level decryption of the encrypted XML

Byte level encrypted XML is received by the receiver. Therefore the first step to decrypt the XML at receiver side is byte level decryption (Figure 4.1). The procedure of byte level decryption of XML is given in the algorithm 8.

The procedure discussed in algorithm 8 is the reverse of the procedure discussed in algorithm 7. The process discussed in the algorithm 8 is depicted in figure B.4. And the example of decrypting the first byte of a message is shown in figure B.5. Result of "Byte Level decryption Algorithm" is *"Compressed XML"*.

Algorithm 8: Byte_Level_Decryption_Algorithm

Input: Encrypted_XML, 10-element key

Result: Compressed_XML

```
1  var compressed_Xml ← null;
2  var byte1 ← null ;
3  var XOR1 ← null;
4  var XOR2 ← null;
5  var priviousByteOfInput ← null;
6  for every byte of the Encrypted_XML do
7      byte1 ← byte_selected_in_current_iteration ;
8      if The byte1 is the first byte in the Encrypted_XML then
9          priviousByteOfInput ← "eight least significant bits of key[1]";
10     else
11         priviousByteOfInput ← byte_selected_in_previous_iteration;
12     XOR1 ← "byte1" ⊕ priviousByteOfInput;
13     XOR2 ← "XOR1"⊕ "eight least significant bits of key[0]";
14     If key[2] is 1 then the first bit of XOR2 is swapped with the eighth bit. And if
         key[2] is 0 then there will be no such swapping. ;
15     If key[3] is 1 then the second bit of XOR2 is swapped with the seventh bit. And if
         key[3] is 0 then there will be no such swapping. ;
16     If key[4] is 1 then the third bit of XOR2 is swapped with the sixth bit. And if
         key[4] is 0 then there will be no such swapping. ;
17     If key[7] is 1 then the fourth bit of XOR2 is swapped with the fifth bit. And if
         key[7] is 0 then there will be no such swapping. ;
18     compressed_Xml ← compressed_Xml concatenates(+) XOR2;
19 end
20 return compressed_Xml;
```

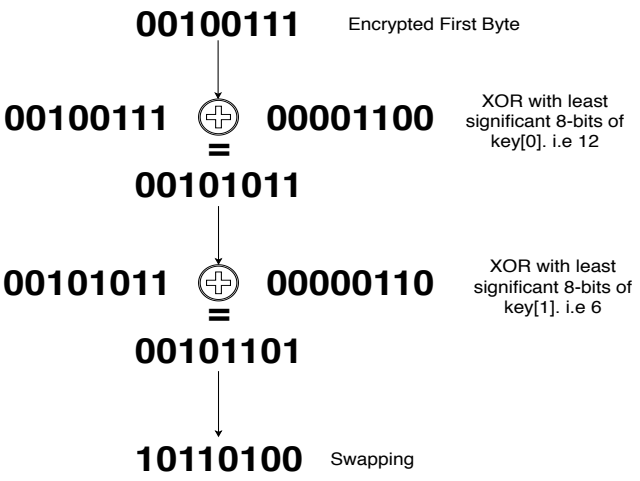


Figure B.5: *Example of Byte Level Decryption (First Byte of Encrypted Message*

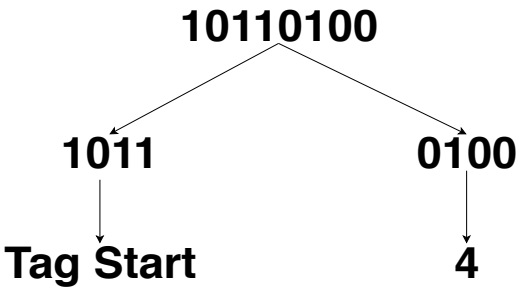


Figure B.6: *Converting a byte in two different symbols*

B.3.2 DeCompression

Algorithm 9 shows the decompression process at the receiver side. Algorithm 9 which is the inverse procedure of the algorithm 6, reads all the bytes of the *"Compressed XML"* one by one. Each byte is then divided into two nibbles. According to the assumptions made in algorithm 6 (i.e 1010 for SPACE, 1011 for Tag start, etc), algorithm 9 then decides which symbol to add to the result of DeCompression. The result of DeCompression (Algorithm 9) at the receiver side is *"Substituted XML"*. Figure B.6 shows the decompression process of a byte.

B.3.3 Substitution Algorithm at Receiver Side

The output of the algorithm 9 is *"Substituted XML"* that the substitution algorithm (i.e algorithm B.7) on the receiver side converts into *"Plain XML"*. While reading the *"Substituted XML"*, the

Algorithm 9: DeCompression_Algorithm

Input: Compressed XML

Result: Substituted XML

```
1  var nibble1 ← null;
2  var nibble2 ← null;
3  var substituted_Xml ← null;
4  for All the bytes in "Compressed XML" do
5      nibble1 ← left 4-bits of current byte;
6      nibble2 ← right 4-bits of current byte;
7      if nibble1 is between 0000 to 1001 then
8          substituted_Xml ← substituted_Xml concatenates(+) 0 to 9 depends upon
             decimal value of nibble1;
9      if nibble1 is 1010 then
10         substituted_Xml ← substituted_Xml concatenates(+) SPACE;
11     if nibble1 is 1011 then
12         substituted_Xml ← substituted_Xml concatenates(+) T;
13     if nibble1 is 1100 then
14         substituted_Xml ← substituted_Xml concatenates(+) A;
15     if nibble1 is 1111 then
16         substituted_Xml ← substituted_Xml concatenates(+) SPACE concatenates(+)
             E;
17     Repeat above If blocks also for nibble2;
18 end
19 return substituted_Xml;
```

algorithm B.7) looks at the various space separated words. The algorithm can differentiate between *"data"* or *"structure"* words by looking at the preceding T, A or E in each word of the *"Substituted XML"*. For *"structure"* words in the *"Substituted XML"*, the algorithm B.7) first searches the corresponding number in the *"Tag Table"*. If the number is found in the *"Tag Table"*, the algorithm B.7 uses the *"Tag Table"* for substitution of such number. Otherwise the algorithm B.7 calls the algorithm 10 with number as input and also calls the algorithm 11 to create a new *"Tag Table"* entry.

"T4", *"A8"*, *"122122104122"*, *"0"* etc. are the various words in the following *"Substituted XML"*.

"T4 A8 850153137820146340 A9 8501531378201463 49 T5 122122102122 E T6 349 E E".

Algorithm 10 accepts one word from *"Substituted XML"* as input. Further the algorithm 10 substitutes every character of a word in substituted_XML using *"Symbol Table"*. The algorithm 11 accepts one word from *"Substituted XML"* as input and creates a Tag Table entry for that word.

Algorithm 10: Symbol Table Based Substitution Algorithm For Receiver

Input: A word in "Substituted XML"

Result: Symbol Table Based Substitution of input word

1 var $STDec \leftarrow null$;
2 Slice the word into sub-strings of key[8] character each;
3 Turn the sub-strings into corresponding integers;
4 Use the *"Symbol Table"* to find the corresponding character of the number in the previous step;
5 Concatenate the characters obtained in the previous step and store the result in variable $STDec$;
6 return $STDec$;

For better understanding of algorithms we assume the following substituted XML (created by substitution using Symbol Table) is taken as the input to the algorithm B.7. substituted_XML = *"T117126126104 A153104104117340 850153137820146340 A153104104117349 850153137820146349 T11615310 122122104122 E T850153137820146 349 E*

Algorithm 11: Find XML Equivalent Of a Word in “Substituted XML" at Receiver Side

Input: A word of *“Substituted XML"*

Result: Tag Table number equivalent to input word

1 var $word \leftarrow Input\_Word$;

2 var x stores the number of entries already present in the Tag Table;

3 var y number of digits required for this Tag Table entry;

4 var *sum* Number equivalent of Given Word in Tag Table;

5 var $diff$;

6 Divide the *word* into *key[8]* equal parts and covert all of them into integers;

7 Add all the integers found in the previous step and store the result of this addition in the variable *sum*;

8 $x \leftarrow$ “the number of entries already present in *Tag Table*" + 1;

9 $y \leftarrow \log_{10}(x+1)$;

10 $diff \leftarrow$ *y-(no. of digits in sum)*;

11 $sum \leftarrow sum \times 10^{diff}$;

12 **if** A record is already present in “Tag Table" with same value of “sum" **then**

13 Keep on increasing the value of *sum* as $sum \leftarrow (sum+1)\%\ 10^{y}$ until *sum* gets a unique value other than *0*;

14 return *sum*;

E". Here the whole *"Substituted XML"* has been substituted by the sender using the *"Symbol Table"*. The following steps show the execution of two algorithms:

1. The word *T117126126104* starts with a *T*, it implies a tag.
2. The preceding *T* is removed from the word. The word *T117126126104* becomes *117126126104*.
3. The number *117126126104* does not have an entry in *"Tag Table"*.
4. Therefore Algorithm B.7 calls the algorithms 10 and 11 with the word *117126126104* as input.
5. The remaining word *117126126104* is sliced into substrings of three characters each because *key[8]* in this case is *3*. Post slicing the receiver gets the following four sub-strings: *"117"*, *"126"*, *"126"* and *"104"*.
6. The corresponding character for each number found after slicing the word is identified in *Symbol Table*. In the number *117126126104* for example, *"117"* belongs to *"r"*, *"126"* belongs to *"o"* and *"104"* belongs to *"t"*.
7. The encrypted number *117126126104* the characters *"r"*, *"o"*, *"o"* and *"t"* are concatenated to make it *"root"*. *STDec* = *"root"* (Algo B.7, step 11).
8. For the word *117126126104*, *sum* = *117* + *126* + *126* + *104* = *473*.
9. *noOfNonVars* = number of new *non-variable* parts introduced in the current message (*7*) + number of entries already present in the *TAT* (*0*) = *7*.
10. $x = \log_{10}(7+1) = 1$.
11. *diff* = *1-3* = *-2*.
12. $sum = 473 \times 10^{-2} = 4$.
13. The record (root, 4)is inserted into the tag table.
14. Algorithm 11 returns *"root"*.
15. The word *0117126126104* is the first word in the series of numbers therefore new value of the variable *"result_of_substitution"* is *"<root"*.

16. New value of the variable *closed* is 0.

17. The word root is pushed onto the stack.

18. Similarly, the variable *"plain_Xml"* will be updated for each number of the series of numbers and finally XML will be found as plaintext.
    "plain_Xml" = "<root attr1='value1'
    attr2='value2'><name>iiti</name><value>2</value>
    </root>".

19. The created *TAT* is the same as that of the sender and is shown in Figure 4.3.

20. If the *"non-variable"* already has an entry in the Tag Table then then use Tag Table to replace number with whole word.

Algorithm: Substitution Algorithm at Receiver Side

Input: Substituted XML
Result: Plain XML

```
1  var plain_Xml <- null, substituted_part <- null, stack \leftarrow null, closed <- 0, sum <- 0;
2  for All the words in Substituted XML do
3      if The word is a ``structure'' part of the Substituted XML then
4          Remove the prefix ``T'' or ``A'' from the word\;
5          if The chosen ``structure'' part has an entry in the Tag Table then
6              substituted_part <- Mapping corresponding to this word in Tag Table;
7          if The choosen ``structure'' part does not have an entry in the Tag Table then
8              substituted_part <- Call Algorithm 11 with word as input;
9              sum <- Call Algorithm 12 with word as input;
10             Insert (``subst\_part'', ``sum'') entry in the Tag Table;
11         if The number is the first word in Output of Substituted XML then
12             plain_Xml <- '<' + substituted_part ; closed <- 0; push substituted_part on stack;
13             continue to the next iteration;
14         if The ``structure'' part is a Tag then
15             if Variable ``closed'' is 0 then
16                 plain_Xml <- plain_Xml + '>' + '<' + substituted_part ;
17             if Variable ``closed'' is 1 then
18                 plain_Xml <- plain_Xml + '<' + substituted_part;
19               closed <- 0;
20             push substituted_part on stack;
21         if The ``structure'' part is an Attribute-name then
22              plain_Xml <- plain_Xml + SPACE + substituted_part + ``='' + ``'';
23     if The word is a ``data'' part of the Substituted XML then
24         substituted_part <- Call Algorithm 11 with word as input;
25         if Value of the variable ``closed'' is 0 then
26             plain_Xml <- plain_Xml$ + $'>'$ + $substituted\_part$ \;
27         if Value of the variable ``closed'' is 1 then
28            plain\_Xml \leftarrow plain\_Xml$ + $substituted\_part$ \;
29         closed <- 1;
30     if The word is ``E'' (i.e ``Tag Closing'') then
31         if The variable ``closed'' is 0 then
32             plain_Xml <- plain_Xml + '>'+'<\'+ pop(stack)+'>';
33         if The variable ``closed'' is 1 then
34             plain_Xml <- plain_Xml + '</'+pop(stack)+'>';
35         closed <- 1;
36     return plain_Xml;
```

Figure B.7: *Algorithm: Substitution Algorithm At Receiver Side*

Bibliography

[1] Abbas, A. M., Bakar, A. A., Ahmad, M. Z., 2014. Fast dynamic clustering soap messages based compression and aggregation model for enhanced performance of web services. Journal of Network and Computer Applications 41, 80–88.

[2] Aggarwal, C. C., Ta, N., Wang, J., Feng, J., Zaki, M., 2007. Xproj: a framework for projected structural clustering of xml documents. In: Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining. pp. 46–55.

[3] Agrawal, R., Kiernan, J., Srikant, R., Xu, Y., 2004. Order preserving encryption for numeric data. In: Proceedings of the 2004 ACM SIGMOD international conference on Management of data. pp. 563–574.

[4] Al-Shammari, A., Liu, C., Naseriparsa, M., Vo, B. Q., Anwar, T., Zhou, R., 2017. A framework for clustering and dynamic maintenance of xml documents. In: International Conference on Advanced Data Mining and Applications. Springer, pp. 399–412.

[5] Al-Shammary, D., Khalil, I., 2010. Soap web services compression using variable and fixed length coding. In: 2010 Ninth IEEE International Symposium on Network Computing and Applications. IEEE, pp. 84–91.

[6] Al-Shammary, D., Khalil, I., 2011. Dynamic fractal clustering technique for soap web messages. In: 2011 IEEE International Conference on Services Computing. IEEE, pp. 96–103.

[7] Al-Shammary, D., Khalil, I., 2012. Redundancy-aware soap messages compression and aggregation for enhanced performance. Journal of network and computer applications 35 (1), 365–381.

[8] Al-Shammary, D., Khalil, I., Tari, Z., 2014. A distributed aggregation and fast fractal clustering approach for soap traffic. Journal of Network and Computer Applications 41, 1–14.

[9] Al-Shammary, D., Khalil, I., Tari, Z., Zomaya, A. Y., 2013. Fractal self-similarity measurements based clustering technique for soap web messages. Journal of Parallel and Distributed Computing 73 (5), 664–676.

[10] Algergawy, A., Mesiti, M., Nayak, R., Saake, G., 2011. Xml data clustering: An overview. ACM Computing Surveys (CSUR) 43 (4), 1–41.

[11] Armando, A., Basin, D., Boichut, Y., Chevalier, Y., Compagna, L., Cuéllar, J., Drielsma, P. H., Héam, P.-C., Kouchnarenko, O., Mantovani, J., et al., 2005. The avispa tool for the automated validation of internet security protocols and applications. In: International conference on computer aided verification. Springer, pp. 281–285.

[12] Barhamgi, M., Bandara, A. K., Yu, Y., Belhajjame, K., Nuseibeh, B., 2016. Protecting privacy in the cloud: Current practices, future directions. Computer 49 (2), 68–72.

[13] Barhamgi, M., Benslimane, D., Amghar, Y., Cuppens-Boulahia, N., Cuppens, F., 2013. Privcomp: a privacy-aware data service composition system. In: Proceedings of the 16th International Conference on Extending Database Technology. pp. 757–760.

[14] Barhamgi, M., Benslimane, D., Medjahed, B., 2010. A query rewriting approach for web service composition. IEEE Transactions on Services Computing 3 (3), 206–222.

[15] Boneh, D., Shacham, H., 2002. Fast variants of rsa. CryptoBytes 5 (1), 1–9.

[16] Bousquet-Mélou, M., Lohrey, M., Maneth, S., Noeth, E., 2015. Xml compression via directed acyclic graphs. Theory of Computing Systems 57 (4), 1322–1371.

[17] Boyer, J., Marcy, G., Datta, P., Hirsch, F., accessed 15 February 2019. Canonical xml version 2.0. https://www.w3.org/TR/xml-c14n2/.

[18] Busatto, G., Lohrey, M., Maneth, S., 2008. Efficient memory representation of xml document trees. Information Systems 33 (4-5), 456–474.

[19] Cleary, J., Witten, I., 1984. Data compression using adaptive coding and partial string matching. IEEE transactions on Communications 32 (4), 396–402.

[20] Coppersmith, D., 1994. The data encryption standard (des) and its strength against attacks. IBM journal of research and development 38 (3), 243–250.

[21] Dworkin, M., 2001. Recommendation for block cipher modes of operation. methods and techniques. Tech. rep., National Inst of Standards and Technology Gaithersburg MD Computer security Div.

[22] Eastlake, D., Hansen, T., 2011. Us secure hash algorithms (sha and sha-based hmac and hkdf). Tech. rep., RFC 6234, May.

[23] Fung, B. C., Trojer, T., Hung, P. C., Xiong, L., Al-Hussaeni, K., Dssouli, R., 2011. Service-oriented architecture for high-dimensional private data mashup. IEEE Transactions on Services Computing 5 (3), 373–386.

[24] Geuer-Pollmann, C., 2002. Xml pool encryption. In: Proceedings of the 2002 ACM Workshop on XML Security. pp. 1–9.

[25] Girardot, M., Sundaresan, N., 2000. Millau: an encoding format for efficient representation and exchange of xml over the web. Computer Networks 33 (1-6), 747–765.

[26] Haroune-Belkacem, N., Semchedine, F., Al-Shammari, A., Aissani, D., 2019. Smca: An efficient soap messages compression and aggregation technique for improving web services performance. Journal of Parallel and Distributed Computing 133, 149–158.

[27] Hashizume, K., Fernandez, E. B., 2009. Symmetric encryption and xml encryption patterns. In: Proceedings of the 16th Conference on Pattern Languages of Programs. pp. 1–8.

[28] Housley, R., 2004. Public key infrastructure (pki). The internet encyclopedia.

[29] Housley, R., Ford, W., Polk, W., Solo, D., 1999. Internet x. 509 public key infrastructure certificate and crl profile. Tech. rep., RFC 2459, January.

[30] Imamura, T., Dillaway, B., Simon, E., Kelvin, Y., Nyström, M., Eastlake, D., Reagle, J., Hirsch, F., Roessler, T., 2013. Xml encryption syntax and processing version 1.1. W3C, Recommendation.

[31] Joan, D., Vincent, R., 2002. The design of rijndael: Aes-the advanced encryption standard. In: Information Security and Cryptography. springer.

[32] Jones, M., Hildebrand, J., 2015. Json web encryption (jwe). Internet Requests for Comments, RFC 7516.

[33] Josefsson, S., et al., 2006. The base16, base32, and base64 data encodings. Tech. rep., RFC 4648, October.

[34] Kaur, G., Fuad, M. M., 2010. An evaluation of protocol buffer. In: Proceedings of the ieee southeastcon 2010 (southeastcon). IEEE, pp. 459–462.

[35] Li, N., Li, T., Venkatasubramanian, S., 2007. t-closeness: Privacy beyond k-anonymity and l-diversity. In: 2007 IEEE 23rd International Conference on Data Engineering. IEEE, pp. 106–115.

[36] Li, W., 2003. Xcomp: An xml compression tool. Ph.D. thesis, University of Waterloo [School of Computer Science].

[37] Liefke, H., Suciu, D., 2000. Xmill: an efficient compressor for xml data. In: Proceedings of the 2000 ACM SIGMOD international conference on Management of data. pp. 153–164.

[38] Lohrey, M., Maneth, S., Mennicke, R., 2013. Xml tree structure compression using repair. Information Systems 38 (8), 1150–1167.

[39] Machanavajjhala, A., Kifer, D., Gehrke, J., Venkitasubramaniam, M., 2007. l-diversity: Privacy beyond k-anonymity. ACM Transactions on Knowledge Discovery from Data (TKDD) 1 (1), 3–es.

[40] Madson, C., Doraswamy, N., 1998. The esp des-cbc cipher algorithm with explicit iv. Tech. rep., RFC 2405, November.

[41] Min, J.-K., Park, M.-J., Chung, C.-W., 2003. Xpress: A queriable compression for xml data. In: Proceedings of the 2003 ACM SIGMOD international conference on Management of data. pp. 122–133.

[42] Mohammed, N., Jiang, X., Chen, R., Fung, B. C., Ohno-Machado, L., 2013. Privacy-preserving heterogeneous health data sharing. Journal of the American Medical Informatics Association 20 (3), 462–469.

[43] Nayak, K., Wang, X. S., Ioannidis, S., Weinsberg, U., Taft, N., Shi, E., 2015. Graphsc: Parallel secure computation made easy. In: 2015 IEEE Symposium on Security and Privacy. IEEE, pp. 377–394.

[44] Nithin, N., Bongale, A. M., 2012. Xbmrsa: A new xml encryption algorithm. In: 2012 World Congress on Information and Communication Technologies. IEEE, pp. 567–571.

[45] Perera, C., Yu, C.-M., Benslimane, D., Camacho, D., Bonnet, C., et al., 2019. Privacy in data service composition. IEEE Transactions on Services Computing 13 (4), 639–652.

[46] Q-Success, 2021. W3techs—world wide web technology surveys.
URL `https://w3techs.com/technologies/overview/content_language`

[47] Ranjan, R., Rana, O., Nepal, S., Yousif, M., James, P., Wen, Z., Barr, S., Watson, P., Jayaraman, P. P., Georgakopoulos, D., et al., 2018. The next grand challenges: Integrating the internet of things and data science. IEEE Cloud Computing 5 (3), 12–26.

[48] Rivest, R. L., Shamir, A., Adleman, L., 1978. A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM 21 (2), 120–126.

[49] Sakr, S., 2009. Xml compression techniques: A survey and comparison. Journal of Computer and System Sciences 75 (5), 303–322.

[50] Seak, S. C., Siong, N. K., 2011. A file-based implementation of xml encryption. In: 2011 Malaysian Conference in Software Engineering. IEEE, pp. 418–422.

[51] Senthilkumar, R., Nandagopal, G., Ronald, D., 2015. Qrfxfreeze: queryable compressor for rfx. The Scientific World Journal 2015.

[52] Seward, J., 1996. bzip2 and libbzip2. avaliable at http://www. bzip. org.

[53] Sion, R., Carbunar, B., 2007. On the computational practicality of private information retrieval. In: Proceedings of the Network and Distributed Systems Security Symposium. Internet Society, pp. 2006–06.

[54] Skibiński, P., Swacha, J., 2007. Combining efficient xml compression with query processing. In: East European Conference on Advances in Databases and Information Systems. Springer, pp. 330–342.

[55] Srivastava, U., Widom, J., Munagala, K., Motwani, R., 2005. Query optimization over web services. Tech. rep., Stanford.

[56] Sweeney, L., 2002. k-anonymity: A model for protecting privacy. International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems 10 (05), 557–570.

[57] Tbahriti, S.-E., Ghedira, C., Medjahed, B., Mrissa, M., 2013. Privacy-enhanced web service composition. IEEE Transactions on Services Computing 7 (2), 210–222.

[58] Tiwary, G. P., Stroulia, E., Srivastava, A., 2021. Compression of XML and JSON API responses. IEEE Access 9, 57426–57439.

[59] Tolani, P. M., Haritsa, J. R., 2002. Xgrind: A query-friendly xml compressor. In: Proceedings 18th International Conference on Data Engineering. IEEE, pp. 225–234.

[60] Wang, S., Agrawal, D., El Abbadi, A., 2014. Towards practical private processing of database queries over public data. Distributed and Parallel Databases 32 (1), 65–89.

[61] Yau, S. S., Yin, Y., 2008. A privacy preserving repository for data integration across data sharing services. IEEE Transactions on Services Computing 1 (3), 130–140.