Access Control Models for XML Databases in the Cloud

Shumukh Alfaqir [1], Saloua Hendaoui [2] Fatimah Alhablani [3] and Wesam Alenzi[4]

Cyber Security Department College of computer science and information Al-Jouf University – Aljouf, Saudi Arabia

Abstract
Security is still a great concern to this day, albeit we have come a long way to mitigate its numerous threats. No-SQL databases are rapidly becoming the new database de-facto, as more and more apps are being developed every day. However, No-SQL databases security could be improved. In this paper, we discuss a way to improve the security of XML-based databases with the use of trust labels to be used as an access control model.

Keywords:
XML Databases, Cloud, Access Control Model, Privacy-Preserving

1. Introduction

With the growth of the internet and the increase in the number of users using an application on the web and mobile, Cloud-Computing has emerged as a solution where computing and processing are no longer conducted locally. Companies and organizations both in governmental and private sectors have immigrated most of their businesses to the cloud, where it becomes more efficient, faster, and maintainable by cloud servers. Hence, the companies and organizations are no longer working on network issues and therefore, they are only focusing on managing their actual businesses.

Access control models are models that are used for managing databases to control the access only to specific users who has certain privileges within a specific organization. For example, in a business company, the database must be only accessible by authorized users such as the business owner, the network administrators, and so on. Another example, in a hospital, where the medical databases can be accessed by authorized users such as patients, doctors, pharmacists, laboratories specialists, and so on. Therefore, the access model either allows or denies users from accessing the data. This is done through assuring three criteria as follows

-Identification: when users claim their identity.
- Authentication: when the claiming user gets authenticated as the real user.
- Authorization: when the user is allowed to access the data.
There are a lot of variations for access control models where it depends on the needs of the organization as well as the methodology used in the implementation. To name a few of the most famous access control models that have been recently studied: attribute-based access control model, role-based access control model, where these access control models are considered fine-grained access control models that are customized with fine, strict, and exact details belonging to the authenticated users who are allowed to gain access to the database. [2]

Extensible Markup Language - XML - databases have become the dominant type of databases being used on the cloud, due to their easy implementation. Yet it is very light with minimum storage space being consumed on the cloud. on the other hand, issues of security and privacy have emerged, where the cloud computing platforms are considered as untrusted media because it is physically existing outside the borders of the workplace of a company. Therefore, privacy-preserving access control to such data being stored on the cloud has become a must to ensure the security and the privacy of the data, as well as to ensure that the data must be only accessible by the authenticated real users i.e., the owner of the data.

There are two types of privacy violations over the cloud:

- **External threat:** where attacks come from outside the network platforms as in common network attacks, such as Denial-of-Service.
- **Internal Threat:** attacks from the operators working on the cloud platforms. To fight against such attacks, data being stored on the cloud are stored in an encrypted form to fight against the external network attacks, while the data being enquired from the XML databases while being encrypted, without the need for the data to be encrypted, this way the data is stored on the cloud is being protected against the internal attacks of the cloud platforms operators. [1]

In this paper, the focus of using Access Control Models that will be used only on XML databases, where the next subsection defines XML databases and mention the featuring characteristics of using XML databases.

1.1 XML Definition and Features

XML stands for extensible Markup Language, the word *Markup* comes for using tags <> as marks, to mark up the stored data within the tag marks. The word *Extensible*

Manuscript received May 5, 2022
Manuscript revised May 20, 2022
https://doi.org/10.22937/IJCSNS.2022.22.5.14

comes from the fact that you can use tags marks in an unlimited manner, unlike when using the HTML where the used tags are pre-defined and limited. The aforementioned features have allowed the XML-Based Databases to be used for storing data and transferring data easily among networked platforms. The most important feature of XML is that it is readable by both machines and humans, this helps in reading, editing, and deleting XML records by programmers, as well as, transferring XML records among machines over the networks. [2]

Lastly, XML is recently used in cloud computing environments due to its ease of implementation and distribution, yet it is a very practical way to model and organize data in a hierarchical way, which helps in adding more info to this hierarchy to add more characteristics such as adding additional features of security and privacy for the sake of sending and receiving data that are stored in the XML records privately and securely through cloud networks. These added security features are going to be the basis for building access control models to assure that the data are being only accessible and retrieved by authorized and authentic users as we shall see in the next section of the literature review.

This remaining of this paper is going to be organized as follows, section 2 lists previous research works related to Access Control Models for XML- based cloud computing platforms. Section 3 introduces the proposed model. Section 4 states the problem to be solved. discussion and analysis, section 5 introduces the proposed methodology. section 6 gives details regarding testing the proposed model, section 7, gives details regarding conducting the performance analysis, section 8 concludes the paper. And finally, section 9 lists the references.

2- LITERATURE REVIEW

In [1], Z. Wu et al. have proposed a privacy protection approach for XML-Based Archive Management Systems in the cloud. Their contribution lies in introducing a trusted body as a middleware between the owner of the data to conduct encryption on the data, after that, when data are sent to be stored in XML databases on the cloud, the data is stored in an encrypted form, with featuring data being appended the data, these featuring data are going be used by the access control. Therefore, data being enquired, are enquired based on the featuring data, so the results of the queries get sent to the owner of the data, without the need to decrypt the data being stored in the cloud. By applying this storing methodology, the data that is being stored in XML databases are guaranteed to be private and secure.
In [2], Norah F. has proposed a new methodology for relabeling the records in the XML databases. This is done by appending trust notations to the indexes of records in the XML database. These trust notations are used later by the access control model to grant access only for authenticated users.

In [3], Samiya F. and Sridevi B have proposed a new privacy-preserving access control model for accessing medical images stored in (EHRS) Electronic Health Records System-based XML databases. Their proposed approach combines both homomorphic encryptions to store the data encrypted, and when sending queries, the access control model applies a multiparty computation for information retrieval privately and securely.

Another work in the same field of EHRS is done in [4], where K. Seol et al. have proposed a new attribute-based access control for XML-Based EHRS. The proposed access control model ensures security and privacy by attaching attribute-data to the stored medical data, so that data are retrieved only to those authenticated users who have these attributes such as doctors, pharmacists, laboratory specialists, etc., this is to protect the medical data from internal attacks that might take place by the could service providers. In addition, XML encryption and XML digital signature are also applied when retrieving data from XML databases, this is to protect the data from external network attacks such as information infringement.

In [6], M. Wang et al. have proposed a dynamic access control model for accessing data stored in XML databases where a scheme called privacy bipartite graph was included to provide a dynamic nature for authentic users to add, edit, and delete the data dynamically over cloud networks. Their proposed scheme aims to achieve protection against external network attacks such as reasoning attacks by using dynamic XML semantic encoding data to be used by the access model on behalf of only authorized users.
In [8], L. Guo et al. have proposed a purpose-based access control that utilizes different structures for different users. To further accomplish applying a security layer on top of the XML database, Shamir’s secret sharing was applied. The experimentations were conducted using C++, and the performance analysis was conducted in terms of storage space and time consumed during requesting queries from the XML database.

Table 1 summarizes the most relevant works being listed in the literature review section in terms of the methodology proposed and evaluation metrics being used.

Paper	Methodology proposed	Evaluation metrics
[1]	Appending data features to the data being used in an XML database, where the access control model uses these data features during sending queries when retrieving data.	Security, Efficiency, and Accuracy.
[2]	Enhancing the labeling syntax in XML databases by adding trust notations into the labeling syntax.	Efficiency, Flexibility, and Scalability
[3]	Proposing an access control model that combines homomorphic encryption and secure multiparty computation.	Security
[4]	Adding attributes to the stored data, to be used by attribute access control model, and using XML Encryption and XML Digital Signature.	Efficiency, Security
[6]	Proposing a dynamic access control model by using privacy Bipartite graph with XML semantic encodings	Security
[7]	Proposing a purpose-based access control that utilizes structure view, as well as Shamir's Secret sharing for protection.	Efficiency in terms of storage space and time consumed during query

Table 1 - Summary of Literature Review

3- PROBLEM STATEMENT

The problem with the current XML-based databases is that there is no practical way to secure access to the database records. Every party can have access to all records within the database. Some may try to achieve a considerable amount of security, but they either employ complex mechanisms or incur high storage and performance overhead.

Several ways can be used to secure such databases, only of which we will be addressing in this project is a trust-based hierarchy access control model. The idea is to assign trust level labels to certain nodes in the hierarchy, depending on which the access to the data in those nodes is restricted. This method is by far the best method that can be implemented in terms of time and storage complexities.
Trust-based access control model has a constant time complexity since the database backend only checks the trust level of the node it is about to access. If the currently logged-in user has sufficient access level, they are granted the right to access the node they are trying to access, otherwise, they are rejected. There is no need to search for any value since the trust level label is stored as a simple XML attribute on the node to be secured.

Storage complexity is constant as well. There is no need to add the trust level label to each child of the secured node, as the level is automatically inherited from parent nodes. If the parent node of any node has a higher trust level label value than its own, the trust level is automatically inherited by that node. For example, if there is a node called **Users** with a trust level of 99, each node of which will have at least a trust level of 99 regardless of what is the current value assigned to it.

In this paper, we are going to propose a trust-based access control model that would be used to grant or deny the access of users to the XML databases. The idea of this proposed scheme would be achieved by proposing a trust labeling approach, which works by adding trust notations (some digits) to the records of the database (the nodes) that reflects the permissions related to the authorized users.
For the performance analysis, it will be conducted in terms of time and storage being consumed.

4- THE PROPOSED METHODOLOGY

Our proposed solution is implemented using the popular programming language C++. the project is implemented using the 2017 standard of the language (C++17) because it offers more compile-time features (thus increasing the overall performance of the final project). The implementation uses the CMake build system as its build scripts generator. The reason we chose this software specifically is that it is cross-platform and can run on any platform. Another reason is that it is the most supported build system for C++ out there and is used by some of the biggest projects built with C++.

The implementation splits the logic of the database into two parts. Each part has distinct responsibilities with separate boundaries. The backend is responsible for managing the database logic. It is responsible for every operation taking place in the database, including adding, deleting, updating, and setting trust levels on the database level. The other part is the database frontend, which is responsible for converting user instructions of different forms into understandable commands to delegate to the backend for execution. The reason we chose to split the project into two parts is that this makes the project much easier to reason about and makes the act of changing the codebase much simpler.

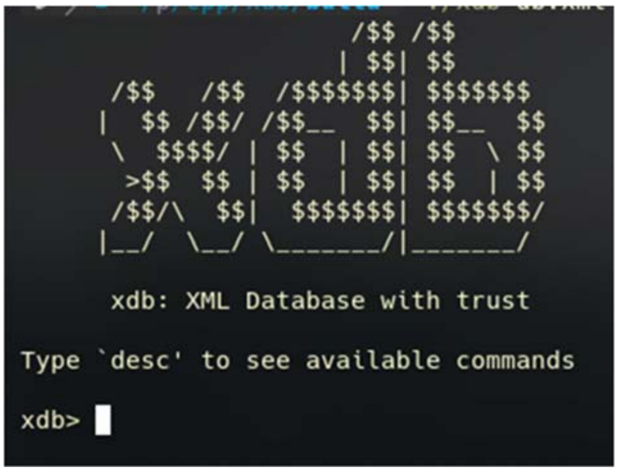
The first part (and by far the most important) is the database backend. The backend has all of the logic that should take place in the database. The database backend is responsible for creating and manipulating the data in the database file. It is also responsible for maintaining security by maintaining the trust access control labels on the data it

is operating on. The backend is also responsible for managing the database file opening, saving, and closing.
To be able to interact with the database backend, the second part is the database frontend comes into place. This part is responsible for taking commands from the user regardless of its form (Command Line Interface- CLI or Graphical User Interface - GUI). The front-end takes the command from the user input source and converts them into a unified form to be passed to the frontend commands. The frontend implements the commands that are used to manipulate the data in the database, as well as querying the data. The current implementation only supports two forms of commands input methods: command-line arguments, and an interactive command-line shell.

Firstly, the commands input method; the command line arguments method works by taking the command line arguments passed to the program when executing. The arguments are taken starting from the third argument (the first is the name of the executable file, and the second is the database file path) to the end of the arguments list. If there were only two arguments passed to the executable file, the second commands input method is used.
While the first commands-input method is simple, it is not sufficient for executing more than one command. The second commands-input method, the Interactive Command-Line shell. The shell runs more than one command in the lifetime of the program, unlike the command line arguments method (which executes only one command). This method works like any other shell by showing a prompt and taking input from the user in a loop.

5- TESTING

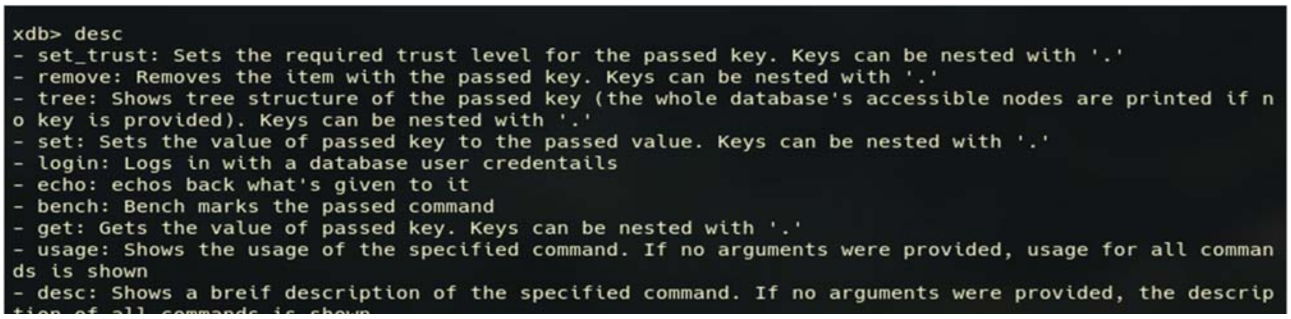
The first thing you see when you run the provided implementation is the ASCII Art logo, shown in Figure 1, and a message saying that you should run “desc” to see available commands.



```
                   /$$ /$$
                  | $$| $$
 /$$   /$$  /$$$$$$$| $$$$$$$
|  $$ /$$/ /$$__  $$| $$__  $$
 \  $$$$/ | $$  | $$| $$  \ $$
  >$$  $$ | $$  | $$| $$  | $$
 /$$/\  $$|  $$$$$$$| $$$$$$$/
|__/  \__/ \_______/|_______/

 xdb: XML Database with trust

Type `desc' to see available commands

xdb> █
```

Fig.1. The greeting screen

The “desc” (which stands for describe) is a built-in command to show descriptions about the commands passed to it. This command has a special case when no arguments are passed to it, which is that the output will be the description of all of the available commands.
Figure 2 shows a list of all available commands in the provided implementation. We shall discuss them one by one in the next sections. We will start with the basic read/write commands, such as “get”, “set”, “remove”, and “tree”. Then we will move to the security commands, namely “login” and “set_trust”. Finally, we will have a tour around the built-in shell commands, which are: “echo”, “usage”, “desc”.



```
xdb> desc
- set_trust: Sets the required trust level for the passed key. Keys can be nested with '.'
- remove: Removes the item with the passed key. Keys can be nested with '.'
- tree: Shows tree structure of the passed key (the whole database's accessible nodes are printed if n
o key is provided). Keys can be nested with '.'
- set: Sets the value of passed key to the passed value. Keys can be nested with '.'
- login: Logs in with a database user credentails
- echo: echos back what's given to it
- bench: Bench marks the passed command
- get: Gets the value of passed key. Keys can be nested with '.'
- usage: Shows the usage of the specified command. If no arguments were provided, usage for all comman
ds is shown
- desc: Shows a breif description of the specified command. If no arguments were provided, the descrip
```

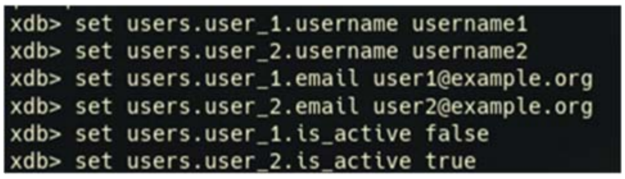
Fig.2.1 “desc” command output with no arguments passed

5.1 First, Read/Write commands:

- “Set” command

The “set” command as shown in Figure 3 is one of the most basic commands found in the provided implementation. It is used to set values to keys passed to it as arguments. The command expects exactly two arguments, the first being the key which to set the value to, and the second is the value to be assigned to the provided key. The command will fail if the user does not have sufficient access to the node identified by the key passed to the command.

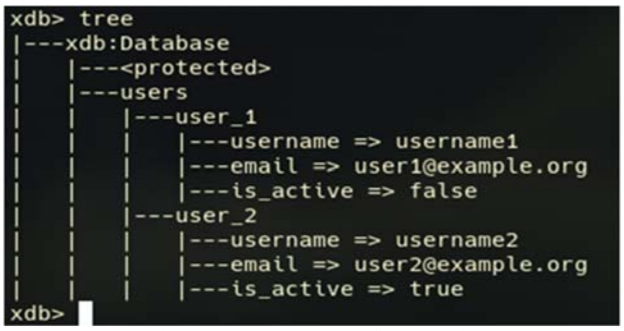
Fig.3. “set” command examples



```
xdb> set users.user_1.username username1
xdb> set users.user_2.username username2
xdb> set users.user_1.email user1@example.org
xdb> set users.user_2.email user2@example.org
xdb> set users.user_1.is_active false
xdb> set users.user_2.is_active true
```

- “Tree” command

The “tree” command as shown in Figure 4 is the easiest way to visualize the database into a readable tree structure. It is the easiest of the two reading commands (the other being “get”). It takes an optional argument, which is the get which to print as a tree. If no argument is passed, the whole database is printed as a tree.



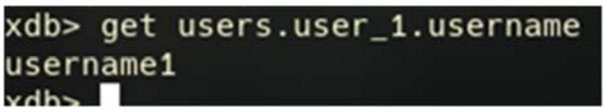
```
xdb> tree
|---xdb:Database
|    |---<protected>
|    |---users
|    |    |---user_1
|    |    |    |---username => username1
|    |    |    |---email => user1@example.org
|    |    |    |---is_active => false
|    |    |---user_2
|    |    |    |---username => username2
|    |    |    |---email => user2@example.org
|    |    |    |---is_active => true
xdb>
```

Fig.4. “tree” command example

We can notice that there is a node marked as “<protected>”. This node is the authentication information node in the database. It is hidden because the current access level is less than the required access level by that node (which is 101 – more than the default max by 1). This field can be shown if we use a user who has access to it, however; we will discuss this in the security commands section.

- “GET” command

The “get” command as shown in Figure 5 is used to get the value of a key passed to it. It accepts exactly one argument, which is the get which to show its value. The command will fail if the user does not have sufficient access to the node identified by the key passed to the command.



```
xdb> get users.user_1.username
username1
xdb>
```

Fig.5. “get” command example

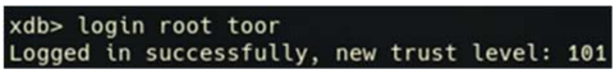
- “Remove” command

The “remove” command is used to delete the data associated with the key passed to it. The command will fail if the user does not have sufficient access to the node identified by the key passed to the command.

1.1 Second, Security commands:

- “Login” command

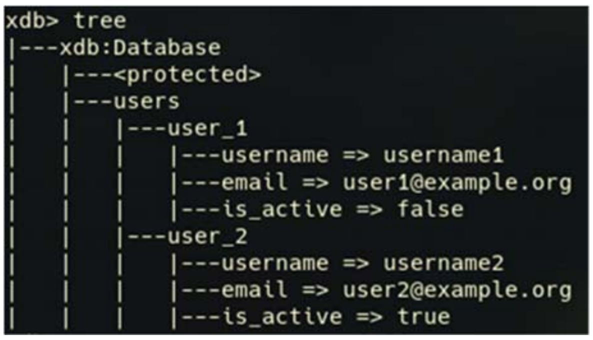
The “login” is used to authenticate a user to the provided database engine. There is one user in the system by default. The default user has the username “root” and the password “toor”. The “root” user has the highest access trust level of **101**. The users’ information resides in the “xdb:Auth” node. The node contains nodes that represent users available on the system. This node is protected by an access label of 101 (meaning only the root user can access it).



```
xdb> login root toor
Logged in successfully, new trust level: 101
```

Fig.6. “login” command example

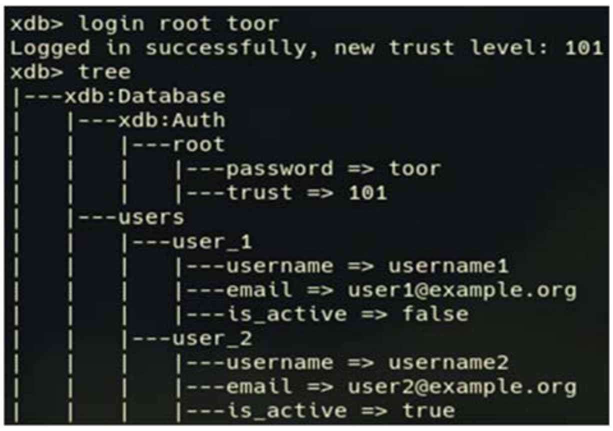
The normal user cannot access the “xdb:Auth” node because it does not have a sufficient trust level. This can be observed in Figure 7. The first node in the database is marked as “<protected>”, meaning it is not possible to access it with the current access level.



```
xdb> tree
|---xdb:Database
|    |---<protected>
|    |---users
|    |    |---user_1
|    |    |    |---username => username1
|    |    |    |---email => user1@example.org
|    |    |    |---is_active => false
|    |    |---user_2
|    |    |    |---username => username2
|    |    |    |---email => user2@example.org
|    |    |    |---is_active => true
```

Fig.7. xdb:Auth node marked “<protected>”

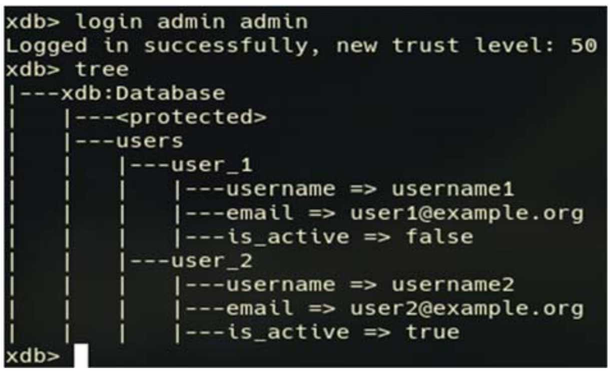
However, the root user can access that node because it has an access trust level of 101, which is the same as the required minimum trust level to access the “xdb:Auth” node. We can see in Figure 8 that after we logged in with the root user, the xdb:Auth node is now revealed when we execute the “tree” command.



```
xdb> login root toor
Logged in successfully, new trust level: 101
xdb> tree
|---xdb:Database
|    |---xdb:Auth
|    |    |---root
|    |    |    |---password => toor
|    |    |    |---trust => 101
|    |---users
|    |    |---user_1
|    |    |    |---username => username1
|    |    |    |---email => user1@example.org
|    |    |    |---is_active => false
|    |    |---user_2
|    |    |    |---username => username2
|    |    |    |---email => user2@example.org
|    |    |    |---is_active => true
```

Fig.8. xdb:Auth node revealed when logging in with the root user

Because the root user can access the authentication node, it can also add other users by just using the "set" command with keys that are inside the "xdb:Auth" node. If logged in with the root user, we can set a child to the xdb:Auth node with the name of the new user we want to create. We can add another child in the created node with the name being "password" and the value is the password of the user. We can set the trust level of the created account by setting the "trust" child value to the value of trust level we want to give to the new user. In Figure 9, we will log in with a newly created user called "admin" with password "admin", and trust level of 50.

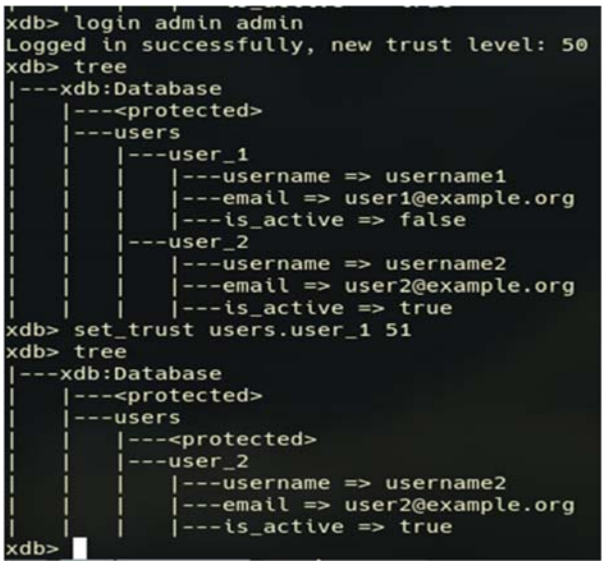


```
xdb> login admin admin
Logged in successfully, new trust level: 50
xdb> tree
|---xdb:Database
|    |---<protected>
|    |---users
|    |    |---user_1
|    |    |    |---username => username1
|    |    |    |---email => user1@example.org
|    |    |    |---is_active => false
|    |    |---user_2
|    |    |    |---username => username2
|    |    |    |---email => user2@example.org
|    |    |    |---is_active => true
xdb> 
```

Fig.9. Logging in with a newly created user, "admin"

- "set_trust" command

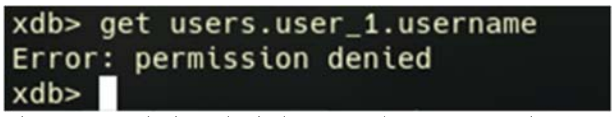
We can set a minimum required trust level on individual nodes in the database using the "set_trust" command. This command accepts exactly two arguments. The first argument is the key on the node, which to set the minimum required trust level on, and the second is the new trust level value. <u>The command will fail if the user does not have sufficient access</u> to the node identified by the key passed to the command.



```
xdb> login admin admin
Logged in successfully, new trust level: 50
xdb> tree
|---xdb:Database
|    |---<protected>
|    |---users
|    |    |---user_1
|    |    |    |---username => username1
|    |    |    |---email => user1@example.org
|    |    |    |---is_active => false
|    |    |---user_2
|    |    |    |---username => username2
|    |    |    |---email => user2@example.org
|    |    |    |---is_active => true
xdb> set_trust users.user_1 51
xdb> tree
|---xdb:Database
|    |---<protected>
|    |---users
|    |    |---<protected>
|    |    |---user_2
|    |    |    |---username => username2
|    |    |    |---email => user2@example.org
|    |    |    |---is_active => true
xdb> 
```

Fig.10. "set_trust" command example

As shown in Figure 10, we log in with the user we created in the previous section. We log in by using the "login" command with the username and password is "admin". We can see that we have been granted an access level of 50 (which is the trust level of the user "admin"). We show the data in the database, and we can see that the data is accessible. Next, we increase the trust level of the node with the key "users.user_1" to 51 (1 above the current access trust level). We try to print the data in the database, but we get that the node "users.user_1" and its children are hidden. In the next command (Figure 11), we see that we cannot access that node by using the "get" command.



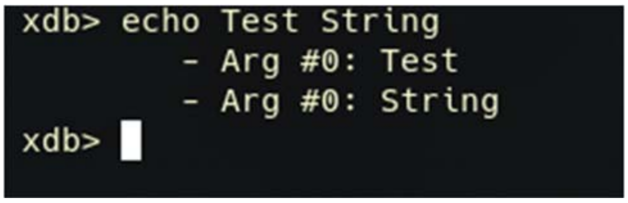
```
xdb> get users.user_1.username
Error: permission denied
xdb> 
```

Fig.11. Permission denied error when we use the "get" command

BUILT-IN COMMANDS

- "Echo" command

The "Echo" command, as shown in Figure 12, is a simple test command to check if the program is working correctly. It accepts zero or more arguments and just prints them back to the user.

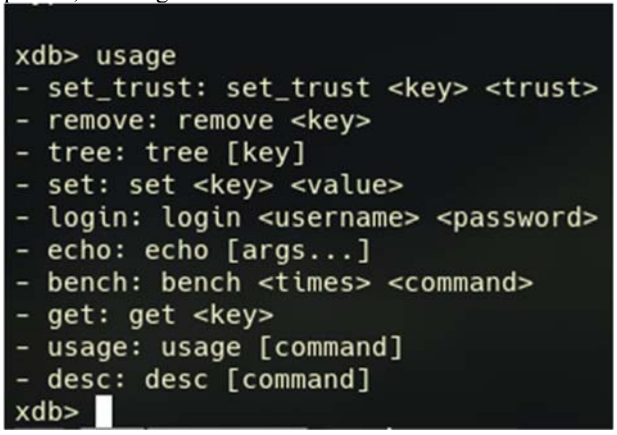


```
xdb> echo Test String
        - Arg #0: Test
        - Arg #0: String
xdb> 
```

Fig.12. "echo" command example

- "Usage" command

"Usage" shows the usage of the command passed to it. It takes either one argument or none. If an argument is passed to the command, the usage of the command identified by the passed argument value is printed. If no arguments were passed, the usage of all of the available commands is shown.



```
xdb> usage
- set_trust: set_trust <key> <trust>
- remove: remove <key>
- tree: tree [key]
- set: set <key> <value>
- login: login <username> <password>
- echo: echo [args...]
- bench: bench <times> <command>
- get: get <key>
- usage: usage [command]
- desc: desc [command]
xdb>
```

Fig.13. "usage" command example

PERFORMANCE ANALYSIS

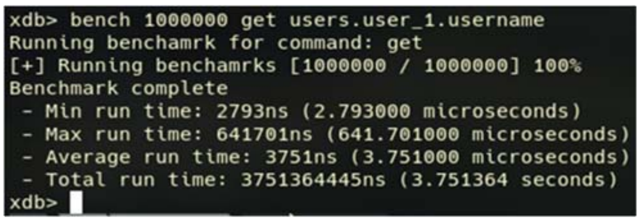
We have added a command called "bench" to the system to benchmark the system executing commands on different system states. We will first benchmark the system with the security being enabled, then we will run the same benchmarks with the security being disabled. Finally, we will test the system after completely disabling the security from within the code.

The specifications of the machine used to do the benchmarking:

- **CPU**: Intel© Core™ i7 10850H (6 cores, 12 threads)
- **RAM**: 16GB DDR4 2933
- **Storage**: Samsung© 970 EVO Plus 2TB PCIe3x4 M.2 SSD
- **GPU:** NVIDIA© RTX 2070 Max-Q 8GB GDDR6

Each benchmark consists of one million iterations of a "get" command.

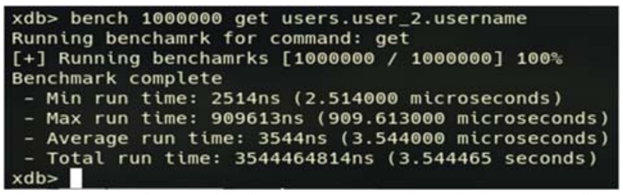
- With security enabled:
  - A protected node:



```
xdb> bench 1000000 get users.user_1.username
Running benchamrk for command: get
[+] Running benchamrks [1000000 / 1000000] 100%
Benchmark complete
 - Min run time: 2793ns (2.793000 microseconds)
 - Max run time: 641701ns (641.701000 microseconds)
 - Average run time: 3751ns (3.751000 microseconds)
 - Total run time: 3751364445ns (3.751364 seconds)
xdb>
```

Fig.14. Security enabled, protected node benchmark

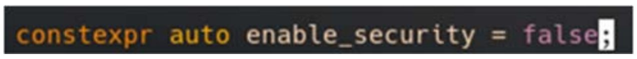
  - A non-protected node:



```
xdb> bench 1000000 get users.user_2.username
Running benchamrk for command: get
[+] Running benchamrks [1000000 / 1000000] 100%
Benchmark complete
 - Min run time: 2514ns (2.514000 microseconds)
 - Max run time: 909613ns (909.613000 microseconds)
 - Average run time: 3544ns (3.544000 microseconds)
 - Total run time: 3544464814ns (3.544465 seconds)
xdb>
```

Fig.15. Security enabled, non-protected node benchmark

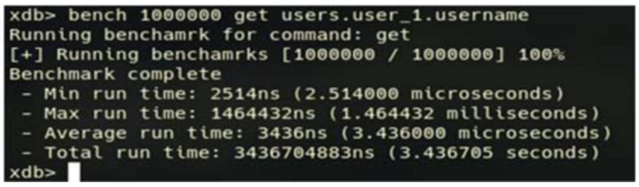
In Figure 16, We disable the security in the code by changing the constant value from true to false in the file "include/xdb/constants.hpp", then recompiling:



```
constexpr auto enable_security = false;
```

Fig.16. Disabling security from the code

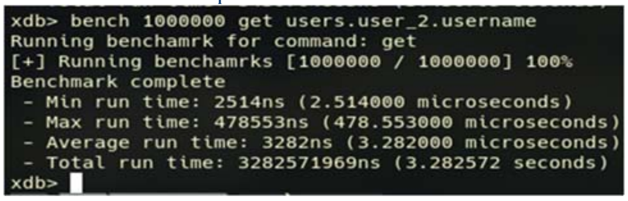
- With security disabled:
  - A protected node:



```
xdb> bench 1000000 get users.user_1.username
Running benchamrk for command: get
[+] Running benchamrks [1000000 / 1000000] 100%
Benchmark complete
 - Min run time: 2514ns (2.514000 microseconds)
 - Max run time: 1464432ns (1.464432 milliseconds)
 - Average run time: 3436ns (3.436000 microseconds)
 - Total run time: 3436704883ns (3.436705 seconds)
xdb>
```

Fig.17. Security disabled, protected node benchmark

  - A non-protected node:



```
xdb> bench 1000000 get users.user_2.username
Running benchamrk for command: get
[+] Running benchamrks [1000000 / 1000000] 100%
Benchmark complete
 - Min run time: 2514ns (2.514000 microseconds)
 - Max run time: 478553ns (478.553000 microseconds)
 - Average run time: 3282ns (3.282000 microseconds)
 - Total run time: 3282571969ns (3.282572 seconds)
xdb>
```

Fig.18. Security disabled, non-protected node benchmark

6.1 OBSERVATIONS

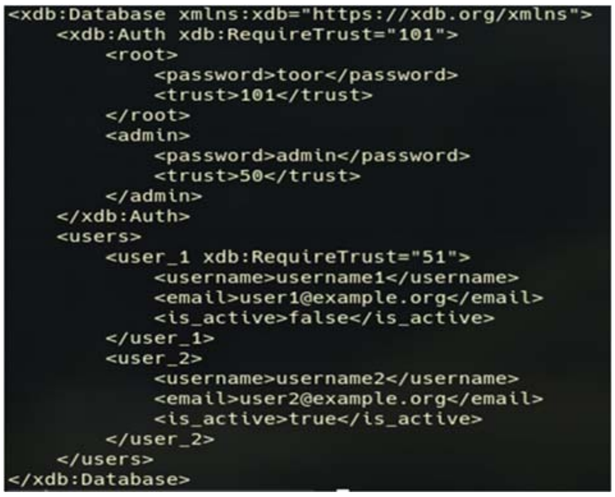
The first benchmark as we can see in figure 14 that the "get" command takes about 3751 nanoseconds (about 3.7 microseconds). The command is executed while the security is being enabled in the codebase, and the current access trust level is 50 (user "admin"), accessing the secured node "users.user_1.username". This (theoretically) should be the slowest scenario in the benchmarks we have covered. The second benchmark as shown in Figure 15 takes on average 3544ns (3.5 microseconds) to execute a get command. The command is executed while security is enabled, and the current access trust level is 50 (user "admin"), accessing the unsecured node "users.user_2.username". This scenario runs 6% faster than the previous one because the node we accessed does not have a security label to be parsed and checked against.

The third benchmark (Figure 17) runs with the security being disabled in the codebase. The test accesses a node with a security label. However, since security is disabled, the label is not checked, thus minimizing overhead incurred by parsing and comparing. This benchmark averages with

3436 nanoseconds, being approximately 9% faster than the first benchmark, and 3.1% faster than the second. The last benchmark (Figure 18) runs with the security being disabled in the codebase. This benchmark should be the least time-consuming benchmark since it does not have any security checks or security labels. This test averages 3282 nanoseconds, being 13.6% faster than the first benchmark. We can conclude that the added security is worth implementing since there is no real performance hit when implemented in this project. The additional security makes the slight performance decrease acceptable for most cases.

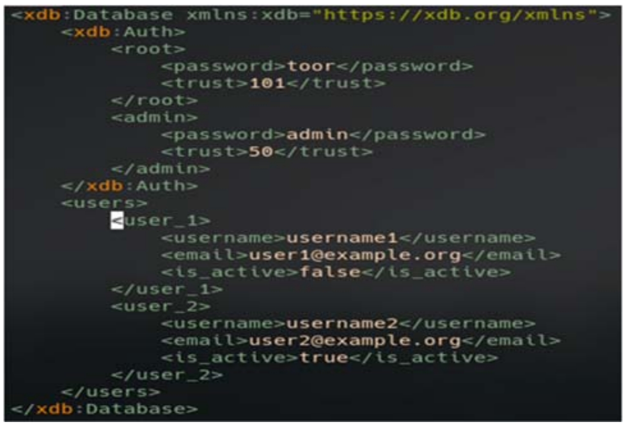
6.2 SPACE ANALYSIS

As we mentioned in the first section, the space overhead of the proposed security implementation is almost negligible. The access trust level labels are only added to tags that are explicitly secured by the user. In the next figures, we show a copy of a database having the security tags on, and another with them being taken away.



```
<xdb:Database xmlns:xdb="https://xdb.org/xmlns">
    <xdb:Auth xdb:RequireTrust="101">
        <root>
            <password>toor</password>
            <trust>101</trust>
        </root>
        <admin>
            <password>admin</password>
            <trust>50</trust>
        </admin>
    </xdb:Auth>
    <users>
        <user_1 xdb:RequireTrust="51">
            <username>username1</username>
            <email>user1@example.org</email>
            <is_active>false</is_active>
        </user_1>
        <user_2>
            <username>username2</username>
            <email>user2@example.org</email>
            <is_active>true</is_active>
        </user_2>
    </users>
</xdb:Database>
```

Fig.19. Security tags present



```
<xdb:Database xmlns:xdb="https://xdb.org/xmlns">
    <xdb:Auth>
        <root>
            <password>toor</password>
            <trust>101</trust>
        </root>
        <admin>
            <password>admin</password>
            <trust>50</trust>
        </admin>
    </xdb:Auth>
    <users>
        <user_1>
            <username>username1</username>
            <email>user1@example.org</email>
            <is_active>false</is_active>
        </user_1>
        <user_2>
            <username>username2</username>
            <email>user2@example.org</email>
            <is_active>true</is_active>
        </user_2>
    </users>
</xdb:Database>
```

Fig.20. Security tags are taken away

Conclusion

As we can see, there are no major space-wasting structures in the database implementation. The security labels take between 20 and 30 additional bytes by each secured node, which is negligible given how cheap storage is nowadays. This makes our security implementation completely feasible for almost every case that requires such a security mechanism.

References

[1] D. SERVOS and S. L. OSBORN, "Current Research and Open Problems in Attribute-Based Access Control," *ACM Computing Surveys,* vol. 49, no. 4, 2017.

[2] Z. Wu, J. Xie, X. Lian and J. Pan, "A privacy protection approach for XML-based archives management in a cloud environment," *The Electronic Library,* vol. 37, no. 6, 2019.

[3] V. Cridlig, R. State and O. Festor, "Secure XML-based Network Management in a Multi-source Context," 2006.

[4] N. Farooqi, "Integrating Trust Notation in XML Database Labelling," *International Journal of Computer and Information Technology,* vol. 06, no. 03, 2017.

[5] S. Firdous and S. B, "EFFICIENT SECURE AND NOVEL ACCESS CONTROL MODEL FOR XML-BASED EHRS," *The International journal of analytical and experimental modal analysis,* vol. XII, no. XII, pp. 549-554, 2021.

[6] K. SEOL, Y.-G. KIM, E. LEE1, Y.-D. SEO and A. D.-K. BAIK, "Privacy-Preserving Attribute-Based Access Control Model for XML-Based Electronic Health Record System," *IEEE Access,* vol. 6, pp. 9114 - 9128, 2018.

[7] M. Wanga, S. Huangb, C. Zheng and H. Lib, "XML Privacy Preserving Model based on Dynamic Context," *International Journal of performability Engineering* , vol. 14, no. 12, pp. 3206-3219, 2018.

[8] L. Guoa, J. Wanga and H. Wub, "Application of Secret Sharing in XML Protection Mechanism," *Procedia Computer Science,* vol. 107, p. 21 – 26, 2017.