INTEGRITY CODED DATABASES:
ENSURING CORRECTNESS AND FRESHNESS OF
OUTSOURCED DATABASES

by

Ujwal Karki

A thesis
submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Science
Boise State University

August 2017
Thesis Title: Integrity Coded Databases: Ensuring Correctness and Freshness of Outsourced Databases

Date of Final Oral Examination: 16 June 2017

The following individuals read and discussed the thesis submitted by student Ujwal Karki, and they evaluated the presentation and response to questions during the final oral examination. They found that the student passed the final oral examination.

Jyh-haw Yeh, Ph.D. Chair, Supervisory Committee
Dianxiang Xu, Ph.D. Member, Supervisory Committee
Gaby Dagher, Ph.D. Member, Supervisory Committee

The final reading approval of the thesis was granted by Jyh-haw Yeh, Ph.D., Chair of the Supervisory Committee. The thesis was approved by the Graduate College.
This thesis work is dedicated to all my family members, well wishers and supporters.
Certainly, this journey until now was not by me alone. There has been continuous guidance and support from many individuals, who have helped me reach this far. First and foremost, I would like to express my sincere gratitude to my very helpful advisor Dr. Jyh-haw Yeh. He always provided me the motivation, correct guidance, important and timely suggestions which has made this work a successful one. I would also like to thank my committee member Dr. Dianxiang Xu from whom I have learned a lot in the issues related to computer security, which is closely related to my thesis work, and also the various aspects of software engineering from the classes I have taken with him. I am also very thankful to my other committee member Dr. Gaby Dagher, for providing me with invaluable feedback, ideas and suggestions during my regular meetings with him. I am also very grateful to him for allowing me to be the part of his advanced cryptography class, where I got to learn a lot of topics related to my thesis work.

The financial support from the Department of Computer Science by providing me a full scholarship is one of the reasons I was able to pursue my graduate studies at Boise State University. I feel very humbled for having that opportunity. I also feel lucky enough to be surrounded by such wonderful friends and colleagues, which made me feel this new place as friendly and as close as my home.

My special thanks go to my mother who has always been my sole inspiration in life’s many lessons.
ABSTRACT

In recent years, cloud storage has become an inexpensive and convenient option for individuals and businesses to store and retrieve information. The cloud releases the data owner from the financial burden of hiring professionals to create, update and maintain local databases. The advancements in the field of networking and the growing need for computing resources for various applications have made cloud computing more demanding. Its positive aspects make the cloud an attractive option for data storage, but this service comes with a cost that it requires the data owner to relinquish control of their information to the cloud service provider. So, there remains the possibility for malicious insider attacks on the data that may involve addition, omission, or manipulation of data. This paper presents a novel Integrity Coded Database (ICDB) approach for ensuring data correctness and freshness in the cloud. Various options for verifying the integrity of queried data in different granularities are provided, such as the coarse-grained integrity protection for the entire returned dataset or a more fine-grained integrity protection down to each tuple or even each attribute. ICDB allows data owners to insert integrity codes into a database, outsource the database to the cloud, run queries against the cloud database server, and verify that the queried information from the cloud is both correct and fresh. An ICDB prototype has been developed in order to benchmark several ICDB schemes to evaluate their performance.
# TABLE OF CONTENTS

ABSTRACT ................................................................................................. vi

LIST OF TABLES ....................................................................................... xi

LIST OF FIGURES ..................................................................................... xvi

LIST OF ABBREVIATIONS ......................................................................... xxii

LIST OF SYMBOLS .................................................................................... xxii

1 Introduction ............................................................................................ 1
  1.1 Background .......................................................................................... 1
    1.1.1 Thesis Statement ............................................................................ 2
    1.1.2 Outline .......................................................................................... 5

2 Previous work ........................................................................................ 6
  2.0.1 Integrity Coded Databases (ICDB)—Protecting Integrity for Outsourced Databases ................................................................. 6
  2.0.2 Securing Storage in Public Cloud Infrastructure ............................. 7
  2.0.3 Freshness Guarantee for an Outsourced Database ....................... 8
  2.0.4 Authentication of Outsourced Databases using Signature Aggregation and Chaining ................................................................. 9
2.0.5 Integrity Protection using Authenticated Data Structure-based Techniques ........................................... 10

3 ICDB Models .............................................................. 12
   3.1 Basic ICDB Model .................................................... 12
   3.2 Dual Mode Verification (DMV) Model ......................... 15

4 ICDB Construction ..................................................... 20
   4.1 Definition of Variables ............................................ 20
   4.2 Integrity Codes .................................................... 21
      4.2.1 Integrity Code Generating Algorithms ................... 23
      4.2.2 Serial Number Construction .............................. 26
   4.3 Granularity Schemes .............................................. 27
      4.3.1 One Code per Field (OCF) ............................... 28
      4.3.2 One Code per Tuple (OCT) ............................. 37
   4.4 Aggregate Integrity Code Generation and Verification .......... 44

5 Experimental Results and Analysis ............................... 47
   5.1 Hardware and Software Used ................................... 47
   5.2 Integrity Protection .............................................. 48
      5.2.1 Forgery Attack .......................................... 48
      5.2.2 Substitution Attack .................................... 49
      5.2.3 Old-Data Attack ....................................... 50
      5.2.4 Tuple Insertion Attack ................................ 51
      5.2.5 Tuple Deletion Attack ................................ 51
   5.3 Memory Penalty ................................................ 52
5.4 Performance Penalty ................................................................. 56
  5.4.1 Experimental Results for the Basic ICDB Model ............... 56
  5.4.2 Experimental Results for the Dual Mode Verification (DMV) Model ................................................................. 85

6 Conclusion and Future Work ...................................................... 117

REFERENCES ............................................................................. 121

A Employees Database Schema ....................................................... 124

B Hardware Specifications .............................................................. 129

C Development Environment: ......................................................... 132

D ICDB Commands ....................................................................... 134
  D.1 Initial Setup ........................................................................ 135
  D.2 Running the ICDB tool ........................................................ 135
  D.3 Commands Available .......................................................... 136
    D.3.1 Convert DB Command .................................................. 137
    D.3.2 Convert Query Command ............................................. 137
    D.3.3 Execute Query Command ............................................. 137

E Query Conversion: ................................................................. 138
  E.1 MySQL Select Queries: ....................................................... 139
    E.1.1 ICDB Select Queries(OCF): ...................................... 139
    E.1.2 ICDB Select Queries(OCT): ....................................... 140
  E.2 MySQL DELETE Queries ...................................................... 141
E.2.1 ICDB Delete Verification Queries (OCF): .................. 141
E.2.2 ICDB Delete Verification Queries (OCT): .................. 142

E.3 MySQL Functional Query: ................................. 143
E.3.1 ICDB Functional Query (OCF): ......................... 143
E.3.2 ICDB Functional Query (OCT): ......................... 143
LIST OF TABLES

3.1 An example of an ICDB table with tuple-level integrity protection . . . . 14

4.1 An example table for OCF granularity ........................................... 28

4.2 An example table for OCT granularity ........................................... 38

5.1 Database sizes: DB is the original Employees SQL database; MAC-OCT and MAC-OCF are the Employees ICDB databases using 128-bit HMAC or CMAC integrity code implementing the OCT scheme and the OCF scheme, respectively. Sizes are displayed in Megabytes. ....................... 53

5.2 Database sizes for the original Employees SQL database, and the converted ICDB RSA-OCT and RSA-OCF databases respectively. Sizes are displayed in Megabytes ........................................ 55

5.3 Using HMAC-SHA, query process time raw data in milliseconds for different number of tuples returned (in thousands) by the SELECT * query over the Employees.salaries table. ......................... 59

5.4 Using CMAC-AES, query process time raw data in milliseconds for different number of tuples returned (in thousands) by the SELECT * query over the Employees.salaries table. ............................. 60

5.5 Using RSA, query process time raw data in milliseconds for different number of tuples returned (in thousands) by the SELECT * query over the Employees.salaries table. ......................... 61
5.6 A table with raw data for the Basic ICDB model showing the process rate, i.e., how much user data (size in MB) can be processed in a SELECT * query using three different algorithms in OCT or in OCF. The data in the DB row is the process rate for a standard SQL database.

5.7 Using HMAC-SHA, query process time raw data in milliseconds for different number of tuples inserted (in thousands) by the INSERT query into the Employees.salaries table.

5.8 Using CMAC-AES, query process time raw data in milliseconds for different number of tuples inserted (in thousands) by the INSERT query into the Employees.salaries table.

5.9 Using RSA, query process time raw data in milliseconds for different number of tuples inserted (in thousands) by the INSERT query into the Employees.salaries table.

5.10 Using HMAC-SHA, query process time raw data in milliseconds for different number of tuples deleted (in thousands) by the DELETE query from the Employees.salaries table.

5.11 Using CMAC-AES, query process time raw data in milliseconds for different number of tuples deleted (in thousands) by the DELETE query from the Employees.salaries table.

5.12 Using RSA, query process time raw data in milliseconds for different number of tuples deleted (in thousands) by the DELETE query from the Employees.salaries table.

5.13 HMAC-SHA raw data for a Query joining the Employees.employees and the Employees.salaries tables, showing the time required (in milliseconds) for the number of tuples selected (in thousands).
5.14 CMAC-AES raw data for a Query joining the Employees.employees and the Employees.salaries tables, showing the time required (in milliseconds) for the number of tuples selected (in thousands). .......... 78
5.15 RSA raw data for a Query joining the Employees.employees and the Employees.salaries tables, showing the time required (in milliseconds) for the number of tuples selected (in thousands). ....................... 79
5.16 HMAC-SHA plotted with time required (in milliseconds) against number of tuples (in thousands) returned using an ICDB SELECT Query corresponding to the Functional Query ......................... 82
5.17 CMAC-AES plotted with time required (in milliseconds) against number of tuples (in thousands) returned using an ICDB SELECT Query corresponding to the Functional Query ......................... 83
5.18 RSA plotted with time required (in milliseconds) against number of tuples (in thousands) returned using an ICDB SELECT Query corresponding to the Functional Query ......................... 84
5.19 Using HMAC-SHA, query process time raw data in AV mode in milliseconds for different number of tuples returned (in thousands) by the SELECT * query from the Employees.salaries table. ............... 89
5.20 Using CMAC-AES, query process time raw data in AV mode in milliseconds for different number of tuples returned (in thousands) by the SELECT * query from the Employees.salaries table. ............... 90
5.21 Using RSA, query process time raw data in AV mode in milliseconds for different number of tuples returned (in thousands) by the SELECT * query from the Employees.salaries table. ............... 91
5.22 A table with process rate raw data for the AV mode in DMV model showing the speed the user data (size in MB) can be processed by three different algorithms in OCT and in OCF. The data in the row labeled DB is the process rate for a standard SQL database. ........................ 93

5.23 Using HMAC-SHA, query process time raw data in AV mode in milliseconds for different number of tuples deleted (in thousands) by the DELETE * query from the Employees.salaries table. ................. 97

5.24 Using CMAC-AES, query process time raw data in AV mode in milliseconds for different number of tuples deleted (in thousands) by the DELETE * query from the Employees.salaries table. ................. 98

5.25 Using RSA, query process time raw data in AV mode in milliseconds for different number of tuples deleted (in thousands) by the DELETE * query from the Employees.salaries table. .................. 99

5.26 HMAC-SHA plotted query processing time for a query joining the Employees.employees and the Employees.salaries tables, showing the time required (in milliseconds) for the number of tuples selected (in thousands) in aggregate verification mode. ......................... 103

5.27 CMAC-AES plotted query processing time for a query joining the Employees.employees and the Employees.salaries tables, showing the time required (in milliseconds) for the number of tuples selected (in thousands) in aggregate verification mode. ......................... 104

5.28 RSA plotted query processing time for a query joining the Employees.employees and the Employees.salaries tables, showing the time required (in milliseconds) for the number of tuples selected (in thousands) in aggregate verification mode. ......................... 105
5.29 HMAC-SHA plotted query processing time against the number of tuples returned using a SELECT Query corresponding to the Functional Query ................................................................. 108

5.30 CMAC-AES plotted query processing time against the number of tuples returned using a SELECT Query corresponding to the Functional Query ................................................................. 109

5.31 RSA plotted query processing time against the number of tuples returned using a SELECT Query corresponding to the Functional Query ................................................................. 110

5.32 Memory penalty rates for all experiments in both the ICDB basic and the DMV models ................................................................. 111

5.33 Average performance penalty rates for all experiments in both ICDB basic and DMV models ................................................................. 111

5.34 Ranking Scale ................................................................. 116

5.35 Ranking of different ICDB schemes in either Basic or DMV model ........ 116
LIST OF FIGURES

2.1 Merkle Hash Tree ............................................................... 11

3.1 A diagram illustrating the interactions between ICDB client and cloud
DB Service Provider .............................................................. 13

3.2 An Architecture for Dual Mode Verification (DMV) model (built on
the top of the basic ICDB model) .............................................. 19

5.1 Database sizes: DB is the original Employees SQL database; MAC-OCT
and MAC-OCF are the Employees ICDB databases using 128-bit HMAC
or CMAC integrity code implementing the OCT scheme and the OCF
scheme, respectively. Sizes are displayed in megabytes. ...................... 53

5.2 Database sizes for the original Employees SQL database, and the con-
verted ICDB RSA-OCT and RSA-OCF databases respectively. Sizes
are displayed in megabytes ....................................................... 55

5.3 Using HMAC-SHA, plotted query process time in milliseconds for dif-
ferent number of tuples returned (in thousands) by the SELECT *
query over the Employees.salaries table. ................................. 59

5.4 Using CMAC-AES, plotted query process time in milliseconds for dif-
ferent number of tuples returned (in thousands) by the SELECT *
query over the Employees.salaries table. ................................. 60
5.5 Using RSA, plotted query process time in milliseconds for different number of tuples returned (in thousands) by the SELECT * query over the Employees.salaries table. ................................. 61

5.6 A chart plotted for Basic ICDB model showing the process rate, i.e., how much user data (size in MB) can be processed in a SELECT * query using three different algorithms in OCT or in OCF. The leftmost bar marked as DB is the process rate for a standard SQL database. .... 63

5.7 Using HMAC-SHA, plotted query process time in milliseconds for different number of tuples inserted (in thousands) by the INSERT query into the Employees.salaries table. ................................. 66

5.8 Using CMAC-AES, plotted query process time in milliseconds for different number of tuples inserted (in thousands) by the INSERT query into the Employees.salaries table. ................................. 67

5.9 Using RSA, plotted query process time in milliseconds for different number of tuples inserted (in thousands) by the INSERT query into the Employees.salaries table. ................................. 68

5.10 Using HMAC-SHA, plotted query process time in milliseconds for different number of tuples deleted (in thousands) by the DELETE query from the Employees.salaries table. ................................. 71

5.11 Using CMAC-AES, plotted query process time in milliseconds for different number of tuples deleted (in thousands) by the DELETE query from the Employees.salaries table. ................................. 72

5.12 Using RSA, plotted query process time in milliseconds for different number of tuples deleted (in thousands) by the DELETE query from the Employees.salaries table. ................................. 73
5.13 HMAC-SHA plotted for a Query joining the Employees.employees and the Employees.salaries tables, showing the time required (in milliseconds) for the number of tuples selected (in thousands). 77
5.14 CMAC-AES plotted for a Query joining the Employees.employees and the Employees.salaries tables, showing the time required (in milliseconds) for the number of tuples selected (in thousands). 78
5.15 RSA plotted for a Query joining the Employees.employees and the Employees.salaries tables, showing the time required (in milliseconds) for the number of tuples selected (in thousands). 79
5.16 HMAC-SHA plotted with time required (in milliseconds) against number of tuples (in thousands) returned using an ICDB SELECT Query corresponding to the Functional Query 82
5.17 CMAC-AES plotted with time required (in milliseconds) against number of tuples (in thousands) returned using an ICDB SELECT Query corresponding to the Functional Query 83
5.18 RSA plotted with time required (in milliseconds) against number of tuples (in thousands) returned using an ICDB SELECT Query corresponding to the Functional Query 84
5.19 Using HMAC-SHA, plotted query process time in AV mode in milliseconds for different number of tuples returned (in thousands) by the SELECT * query from the Employees.salaries table 89
5.20 Using CMAC-AES, plotted query process time in AV mode in milliseconds for different number of tuples returned (in thousands) by the SELECT * query from the Employees.salaries table 90
5.21 Using RSA, plotted query process time in AV mode in milliseconds for different number of tuples returned (in thousands) by the SELECT * query from the Employees.salaries table. .......................... 91

5.22 A chart plotted the process rates for the AV mode in DMV model showing the speed the user data (size in MB) can be processed by three different algorithms in OCT and in OCF. The leftmost bar labeled DB is the process rate for a standard SQL database. ......................... 93

5.23 Using HMAC-SHA, plotted query process time in AV mode in milliseconds for different number of tuples deleted (in thousands) by the DELETE * query from the Employees.salaries table. ................. 97

5.24 Using CMAC-AES, plotted query process time in AV mode in milliseconds for different number of tuples deleted (in thousands) by the DELETE * query from the Employees.salaries table. ................. 98

5.25 Using RSA, plotted query process time in AV mode in milliseconds for different number of tuples deleted (in thousands) by the DELETE * query from the Employees.salaries table. .......................... 99

5.26 HMAC-SHA plotted query processing time for a query joining the Employees.employees and the Employees.salaries tables, showing the time required (in milliseconds) for the number of tuples selected (in thousands) in aggregate verification mode. ......................... 103

5.27 CMAC-AES plotted query processing time for a query joining the Employees.employees and the Employees.salaries tables, showing the time required (in milliseconds) for the number of tuples selected (in thousands) in aggregate verification mode. ......................... 104
5.28 RSA plotted query processing time for a query joining the Employees.employees and the Employees.salaries tables, showing the time required (in milliseconds) for the number of tuples selected (in thousands) in aggregate verification mode. ............................... 105

5.29 HMAC-SHA plotted query processing time against the number of tuples returned using a SELECT Query corresponding to the Functional Query ................................................................. 108

5.30 CMAC-AES plotted query processing time against the number of tuples returned using a SELECT Query corresponding to the Functional Query ................................................................. 109

5.31 RSA plotted query processing time against the number of tuples returned using a SELECT Query corresponding to the Functional Query 110
LIST OF ABBREVIATIONS

ICDB – Integrity Coded Databases

IC – Integrity Code

AIC – Aggregate Integrity Code

SQL – Structured Query Language

CA – Cloud Application

CDS – Cloud Database Server

DMV – Dual Mode Verification

AV – Aggregate Verification

DV – Detailed Verification

OCF – One Code per Field

OCT – One Code per Tuple

MAC – Message Authentication Code

CMAC – Cipher-based Message Authentication Code

HMAC – key-based Hash Message Authentication Code

ICRL – Integrity Code Revocation List
LIST OF SYMBOLS

+     concatenation/ summation
*     multiplication
\texttt{mod} modulo operation
$^{-1}$ multiplicative inverse operation
⊕     exclusive or (XOR)
Π     product
CHAPTER 1

INTRODUCTION

1.1 Background

Cloud services provide tremendous benefits, since businesses do not need to maintain an IT department that hires an IT staff and purchases-installs expensive hardware and software. Among many cloud services, Database-as-a-Service (DaaS) has grown significantly in the cloud market in recent years. Examples include Amazon Relational Database Service [13], Microsoft Azure SQL Database [12], Google Cloud SQL [11] and many others. However, the convenience of outsourcing database management to the cloud comes with potential privacy and security risks [17]. Different kinds of attacks can be carried out on the database [10]. Insiders who have access to private user data (e.g., DBAs and system administrators) could steal, modify, or even destroy sensitive information. Most notably, data owners subscribed to the cloud service must trust the service providers and assume the cloud system/database administrators will not attempt to manipulate their data. Despite the existence of such insider security threats, current Cloud Database Server (CDS) do not implement any data integrity protection mechanism to assure customers. Although they have implemented state of the art technologies to defend against external attacks, there remains the real risk of attacks from insiders who have privileges to access the stored data. Thus, this insider security threat is one of the major concerns in developing/growing cloud technologies.
and it needs to be addressed.

While protecting data privacy in the cloud is an important topic, protecting data integrity is equally important, but often is a more challenging issue [1] since a company will actually lose the physical control of their data if the company wishes to outsource their data to the cloud. Without physical control of their data, the company cannot prevent (but may be able to detect) unauthorized data tampering from malicious cloud insiders.

1.1.1 Thesis Statement

Since unauthorized data modification in outsourced databases cannot be prevented, this paper presents an Integrity Coded Database (ICDB) approach, as first seen in [18], which makes it possible for data owners to detect an insider’s modifications.

Database Integrity

Data integrity protection is to prevent/detect unauthorized data modifications. Once a database is outsourced, the data owner expects that data item retrieved from the cloud database should be the original data without unauthorized modification. If there is no protection mechanism in place, the database service provider (e.g., malicious DBA or compromised cloud software) can provide fabricated data without detection.

By using an ICDB, the data owner is capable of verifying the integrity of their queried data. The key idea of ICDB is to insert some Integrity Codes (IC) into the database and these ICs are stored alongside with the data they are protecting. The ICs are generated by applying a cryptographic function with the data (to be protected) and the data owner’s secret key as input. A unique serial number is
assigned to each IC to guarantee the freshness of the data (defined below). When
an ICDB is queried, each data item will be fetched along with its corresponding IC
and serial number. Use of the secret key in the construction of ICs assures that only
the database owner is able to verify the integrity of queried data. To verify data, the
data owner re-computes the IC from the data (with serial number and secret key)
and then comparing it to the IC returned from the query. If either the data or the
IC is forged, the data owner will be able to detect these changes. In order to fully
guarantee data integrity, the returned data should be

- **Correct**: Returned data should be original, and not forged.

- **Fresh**: Returned data should be current and not include previously removed
data.

- **Complete**: All data items satisfying query conditions should be returned.

By upholding all three points above, it is possible to achieve maximal integrity by
detecting the following unauthorized actions:

- **Data Manipulation**: The alteration of data in the database or in the returned
  values.

- **Data Omission**: Deletion of data in the database or omission of information
  in the returned values.

- **Data Addition**: Insertion of data in the database or addition of information
  in the returned values.

- **Stale Data**: Returning old data or data previously removed from the database.
Due to the complexity and performance issues incurred while enforcing data completeness as shown in [2], [3] and [4], our proposed ICDB approach in this study will only concentrate on ensuring correctness and freshness. This means that the ICDB models as formulated in this thesis will be only able to fully detect data manipulation, data addition, and stale data on the fetched query results.

Database Evaluation

Although ICDB is able to detect data alterations, it comes with a cost. An ICDB will incur memory penalties for storing integrity codes in addition to the original data, and performance penalties to retrieve extra information (integrity codes) for verifying the integrity of returned data. To test and benchmark the trade-off between security and performance penalty, we have implemented an ICDB prototype [16]. This thesis will focus on answering the following research questions:

- **Question 1: Integrity protection.** How effective is the introduction of Integrity Codes to ensure the data correctness and freshness?

- **Question 2: Memory Penalty.** How much additional memory is required to store integrity codes?

- **Question 3: Performance Penalty.** How much additional time is required for data to be verified after it is queried and/or how much extra information (i.e., integrity codes) is required to be retrieved from the cloud in every query?

Each of these evaluation metrics will be benchmarked against a pre-populated database. The database will be converted to an ICDB first, and then it will be tested by several experiments designed to evaluate its performance. In this thesis, three cryptographic
algorithms were chosen to generate ICs. They are RSA, CMAC AES, and HMAC SHA. In addition, we have studied the ICDB in two different integrity protection granularities, i.e., One integrity Code per Tuple (OCT) or One integrity Code per Field (OCF), which will be described in section 4.2.

1.1.2 Outline

This thesis is organized into six chapters describing the various aspects of ICDB. We also share the experimental results and offer suggestions for various real cloud schemes.

In section 1.1, we discuss concerns about outsourced databases and provide a thesis statement to address the problem described.

In Chapter 2, we give a literature review on various existing data integrity protection schemes of outsourced databases.

In Chapter 3, we propose two different models of ICDB and how to evaluate their performance.

In Chapter 4, we present the construction of Integrity Unit (integrity code and serial number) and also describe the two different integrity granularity schemes.

In Chapter 5, we present and analyze the experimental results for different combinations of algorithms and granularities for both the ICDB models.

In Chapter 6, we conclude the research and also outline future work that can be done.
CHAPTER 2

PREVIOUS WORK

In this chapter we discuss some of the research that is directly related to the integrity protection of outsourced databases. The ongoing research has made use of different approaches such as signature, hash functions, and encryption. We will review how these various research approaches are related to this research work.

2.0.1 Integrity Coded Databases (ICDB)–Protecting Integrity for Outsourced Databases

Nanjundarao’s work [18] (2015) defined the processes for which ICDBs can be created and queried. Her thesis outlined and provided preliminary results for an ICDB model using RSA to generate one integrity code per data field.

My thesis is a continuation of Nanjundarao’s work. My research provides additional ICDB configurations and enhancements to improve performance. In addition to using RSA signatures, this thesis details performance results for using AES and SHA as the underlying integrity-code-generating function, along with performance differences for different integrity granularity protection, i.e., generating integrity codes per field, or per tuple. We also introduce an innovative technique ”aggregate integrity code verification” in the ICDB model to significantly enhance its performance.
2.0.2 Securing Storage in Public Cloud Infrastructure

Haxhijaha et al. (2014) in their research [25] focused on verifying the integrity of the outsourced data by the client themselves, rather than depending upon some third party auditor. Their paper attempts to point out advantages and security concerns of cloud computing and focuses on avoiding third party auditors. For this, hash values of files are computed at the customer’s side to avoid the need for third party auditors. The hash values are then stored at a secure local hash repository. The client can request the data file from the cloud at any time and regenerate the hash for the file. The regenerated hash value can be matched against the precomputed and stored hash values for verification. Unlike ICDB, the technique proposed in their paper is targeted for the file system and not for databases. The MD5 hash algorithm is used as cryptographic algorithm. The MD5 is reported with various attacks [36]. The hash values are stored at the client machine, which incurs extra storage cost to the client. In their study, unfortunately no experimental results regarding performance overhead are provided.

In 2004, researchers [19] outlined an implementation for ensuring data completeness in relational databases. Although not explicitly stated, the model offers correctness and completeness guarantees by providing a signature per tuple for correctness, along with a separate table signature for completeness. Tuple signatures, referred to as Record Integrity Codes (RICs), are generated using keyed hashes. In addition to RICs, the paper offers an incremental signature scheme using XOR MACs [20]. With this they are able to provide a table-level signature that requires computation based on the number of updates on the data (and not the size of the table). This table signature is able to detect additions and deletions of tuples in the database, thereby
ensuring completeness for each table.

Much like RICs, integrity codes in ICDB offer correctness by generating a keyed signature. The major difference is that integrity codes in ICDB, in addition to signatures, also have a serial component attached to them to maintain freshness. Unlike RICs, integrity codes in ICDB can also be generated per field.

2.0.3 Freshness Guarantee for an Outsourced Database

A 2008 paper [3] focused on providing freshness guarantees for outsourced databases provided two approaches for freshness, using timestamps or using a probabilistic method. The first approach with timestamps sets an expiration time for each signature, and the signature must be updated when it is expired. This approach works under an assumption that the local time is always correct. The second approach defines operations, known as audits, which occasionally perform fake updates. Audits can insert new fake tuples and delete old ones from the outsourced database, checking that the new fake tuples always show up in query results. If any deleted fake tuple show up in the query result, the old data attack can be detected.

Another research work [14] tried a similar approach by inserting fake data. They require a Trusted Third Party (TTP) for performing verification, where the data owner hands the check sum to the TTP. The search request contains two parts: the real search request and the verification one. For the real one the user can receive the expected results but for the verification one a check value sum' generated by the service provider should be provided. The user can then compare the sum and sum' to check for freshness and completeness.

ICDB’s freshness guarantee differs from both the timestamp and probabilistic methods. The timestamp method requires constant updates to expired timestamps,
while the probabilistic method is not completely secure, and both methods require periodical operations (timestamp updates and audits) which contribute noticeable system overhead. ICDBs use serial numbers, which can be cross-checked with a compact list of invalid serial numbers without extra operations. This significantly reduces system overhead to guarantee freshness, since the serial number only needs to be updated when the data is updated.

2.0.4 Authentication of Outsourced Databases using Signature Aggregation and Chaining

A paper detailing another authenticated database system [4] (2006), provides a similar approach to our ICDB implementation and addresses the integrity of query replies in the outsourced databases. The approach, called Digital Signature Aggregation and Chaining (DSAC), is used along with digital signatures at the granularity of individual tuples. Each individual signature is constructed by a hash value of the tuple combined with the hash value of its preceding tuple in each searchable attribute, signed with a secret key. This construction helps to achieve completeness by a secure linking of tuple-level signatures to form a so-called signature chain. The verification object includes sets of matching tuples for the query, sets of immediate predecessor and successor nodes of the first and last nodes respectively along the search dimension, a unified signature rather than individual signatures, and hashes of immediate predecessor tuples along all other searchable dimensions.

Like ICDBs, they provided results for signing each tuple in the database for correctness. The key difference is that their approach provides completeness with signature chaining, but no freshness guarantees are provided. Also, the paper is solely focused on the completeness problem for range queries. This requires the tuples to
be sorted in ascending order prior to the signature computation. This adds extra computational overhead to both the database server(cloud) and the client.

2.0.5 Integrity Protection using Authenticated Data Structure-based Techniques

Niaz and Saake in their work [5] focused on Merkle Hash Tree–based data integrity techniques. Merkle Hash Tree is a Signature Scheme based on a binary tree of hashes. Each leaf node holds the hash of a data block. To verify the integrity of any data block, a signer transmits the hashes of only those nodes which are involved in the authentication path of the data block under consideration. With those hashes, the client can compute the hash of the root node and then match it against the stored root hash. For example, for the merkle hash tree in Figure 2.1, if the receiver needs to verify the integrity of data block 2 then only $H(1)$, $H(3,4)$ and $H(5,8)$ need to be transferred to the receiver. With these hashes, the receiver can compute the root hash $H(1,8)$ and then compare it against the stored root hash. Two models have been implemented: Single Authentication Table (SAT), which is one table with all of the authentication data and Level Based Authentication Table (LBAT), where authentication data for each level of an MHT is stored in an individual table.

Though the integrity is protected, the structure of the relational database is changed. In addition, it requires the server side to trace the involved nodes in the authentication path and thus adding performance overhead. The ICDB model is more transparent to the database server without the need to change the database structure or to trace the authentication path.
Figure 2.1: Merkle Hash Tree
CHAPTER 3

ICDB MODELS

In this thesis, we propose two ICDB models: (1) the Basic ICDB model and (2) Dual Mode Verification (DMV) model. The DMV model is built on top of the basic model and verifies query results in aggregation.

3.1 Basic ICDB Model

There are two components in the basic ICDB model: the ICDB client and the Cloud Database Server (CDS) as shown in figure 3.1. The Cloud Service Provider uses a Cloud Database Server for data storage and various database-related operations. The ICDB client and the Cloud Service Provider are connected via the Internet. The ICDB client (on behalf of the data owner) seeks to outsource their data to the CDS, but would like to keep the capability of detecting unauthorized changes to their data. Hereafter, we will use the term "ICDB client" instead of data owner, knowing that the ICDB client is a software client conducting ICDB operations on behalf of the data owner.

The ICDB client has five different modules responsible for central management, DB conversion, query conversion, verification and user interface display respectively. The ICDB Manager is responsible for managing the other four modules by providing input and receiving output for each module, forwarding data between modules,
Figure 3.1: A diagram illustrating the interactions between ICDB client and cloud DB Service Provider connecting with the Cloud Service Provider via the Internet, and displaying results on the user interface. The ICDB database instance is created and outsourced to the cloud as shown in Steps 1 and 2 in figure 3.1. The ICDB manager, upon receiving a SQL query request from the user, will forward the SQL query to the ICDB query conversion module to convert the SQL query to an ICDB query which will then be forwarded to the CDS as shown in Steps 3 and 4. The CDS returns the query result
along with serial numbers and Integrity Codes to the ICDB manager and the ICDB manager forwards the query result to the verification module in Step 5. Finally, the results of integrity verification are presented to the user in Step 6.

The model in figure 3.1 is designed with the goal that employing the proposed ICDB technology will be transparent to the cloud service provider. The ICDB client can convert an existing database to an ICDB instance without the need to redesign existing database systems (e.g., MySQL, PostgreSQL). The CDS does not have to treat any ICDB instance differently from a standard database.

The ICDB client is responsible for the following actions:

1. Converting a database to an ICDB instance by inserting Integrity Codes (ICs) into the database before outsourcing it to the service provider:

An Integrity Code (IC) is a cryptographic code used for the integrity verification for a data item (either an attribute or a tuple). The code is constructed by applying a cryptographic function with inputs such as the data itself, the secret key of the data owner, and a unique ‘Serial Number’. Each IC is generated to protect either an attribute data or the entire tuple in a table based on the level of protection granularity. For example, in the Table 3.1 below, the column ‘IC’ represents the Integrity Code of the entire tuple along with a unique serial number associated with the IC. In the case of attribute level integrity protection, each attribute will have an IC column and a serial number column associated with it.

<table>
<thead>
<tr>
<th>dept_no</th>
<th>dept_name</th>
<th>IC (d001,Marketing,1135980685)</th>
<th>serial</th>
</tr>
</thead>
<tbody>
<tr>
<td>d001</td>
<td>Marketing</td>
<td>IC(d001,Marketing,1135980685)</td>
<td>1135980685</td>
</tr>
</tbody>
</table>

Table 3.1: An example of an ICDB table with tuple-level integrity protection
2. Upon receiving a SQL query, converting it to an ICDB query:

   In addition to the requested data, the ICDB query should also retrieve corresponding ICs and serials related to the data so that the ICDB client is able to verify the integrity of requested data. In the example of Table 3.1, a SQL query to retrieve department number is written: 
   ```
   SELECT dept_name FROM departments WHERE dept_no='d001';
   ```
   The converted ICDB query for the given SQL query would be: 
   ```
   SELECT dept_no, dept_name, IC, serial number FROM departments WHERE dept_no='d001';
   ```
   See below for why the ICDB query needs to retrieve extra data for the integrity verification.

3. Verifying the integrity of queried data to ensure correctness and freshness:

   After the data is fetched along with the corresponding IC and serial number, verification is done by the ICDB client recomputing the IC to compare with the IC fetched from the ICDB query. Regeneration of the IC requires the dept_no, the dept_name, the IC and the serial number.

3.2 Dual Mode Verification (DMV) Model

Comparing with the data to be protected, integrity code size could be relatively larger. Fetching these larger size integrity codes along with queried data is a notable overhead in the network. To reduce this network overhead, we propose an innovative Dual Mode Verification (DMV) model. In addition to the ICDB client running in the data owner’s machine, this model requires an ICDB cloud application to be running in the cloud service provider. This is easily done through Software-as-a-Service (SaaS) [31] [32] in cloud computing. The DMV model provides an option to verify the fetched data in:
1. **Aggregate Verification (AV)** mode: verify the integrity of all fetched data as a whole, and/or

2. **Detailed Verification (DV)** mode: verify the integrity of each fetched tuple or attribute in a finer granularity.

The DMV model shown in figure 3.2 is designed to reduce network overhead if the ICDB client chooses to perform data integrity verification using the AV mode only. If the AV mode verification fails, it means some data is corrupted within the fetched dataset. In this case, the ICDB client can either discard the entire dataset or can choose to perform the DV mode verification to figure out which data items (attributes or tuples) in the fetched dataset are actually corrupted. This means integrity codes will be fetched only in the DV mode if the aggregate verification fails.

The DMV architecture consists of three components: the **ICDB Client**, the **Cloud Database Server (CDS)** and the **ICDB Cloud Application (CA)**. The DMV model has an extra component, the ICDB cloud application, for aggregate verification in addition to those components in the basic ICDB model. The ICDB cloud application is responsible for generating an **Aggregate Integrity Code (AIC)** for the entire dataset fetched in the AV mode. AIC is a single integrity code that is generated by aggregating all the ICs of all fetched data. Depending on the encryption algorithm used, we will need to use a different approach to generate the AIC. The algorithms and techniques used for generating ICDB integrity codes and AIC will be described in detail in sections 4.2 and 4.4. In aggregate verification mode, the ICDB client only needs to fetch one integrity code (AIC) along with the query result, rather than an integrity code for each data item in the query result. Current cloud service providers have 64-bit maximum size limit [15] for a BigInteger variable in the cloud...
database server. This restricts the computation of AIC in the cloud database server and thus the DMV model requires a separate cloud software application for AIC generation. We call this software the ICDB cloud application or the AIC generator.

Because the DMV model is built on top of the basic ICDB model, the module for converting the database to an ICDB instance and outsourcing it remains the same as that shown in steps 1 and 2 in figure 3.2. Similar to the Basic model, the ICDB manager in the DMV model is responsible for forwarding an SQL query to the ICDB query conversion module based on the level of protection granularity as depicted in Step 3. The difference is that the DMV model converts an SQL query to two ICDB queries Q1 and Q2 in step 4. Q1 is sent to the cloud database server to fetch the SQL query result (data along with associated serial numbers), and Q2 is sent to the ICDB cloud application to delegate the ICDB cloud application to fetch all the ICs corresponding to the data retrieved by Q1. The ICDB cloud application, on receiving Q2, forwards it to the CDS to fetch all the ICs as described in Steps 5 and 6. The ICDB cloud application then generates and sends the AIC to the ICDB client as shown in Step 7. In this step, the AIC is generated from ICs (not from data). For aggregate verification, the ICDB client has to regenerate the aggregate integrity code from data (not from ICs), using the ICDB Q1 result that includes data and the serials fetched from the CDS in Step 8. The details of AIC generation and regeneration by the cloud application and by the ICDB client respectively will be described in detail in section 4.4. The AIC regenerated by the ICDB client is then matched with that returned by the ICDB cloud application for aggregate verification. If aggregate verification fails, the ICDB client provides an option for DV. If aggregate verification succeeds, the ICDB manager presents verified data to the user through the User Interface Display. Since DV is optional, if the user chooses to not perform detailed verification, the
entire dataset will be discarded and a message of a failed aggregate verification will notify the user as shown in Step 10. On the other hand, if the user chooses to perform detailed verification, all the ICs for all the data items (fetched earlier by Q1) have to be fetched using ICDB Q2 as shown in Steps 11 and 12. Upon receiving the ICs from the CDS, the ICDB manager will forward the results of both ICDB Q1 and Q2 to the detailed verification module as in Step 13. Finally, the result of detailed verification will present each individual corrupted data item to the user as shown in Step 14. DV can detect either particular tuples or attribute values have been altered, depending on the level of protection granularity. Since the integrity code construction in the DMV model is same as that of the basic ICDB model, the ICDB instances in the DMV model are no different than those in the basic ICDB model.
Figure 3.2: An Architecture for Dual Mode Verification (DMV) model (built on the top of the basic ICDB model)
CHAPTER 4

ICDB CONSTRUCTION

4.1 Definition of Variables

In this section, we will define some variables.

- \(d\): A data item to be protected. In this thesis, it could be a field entry (an attribute value) or an entire tuple. In ICDB, each data item \(d\) will have an Integrity Code (IC) to ensure its integrity.

- \(D = \{d_1, d_2, \ldots, d_r\}\): A dataset (a set of data item \(d_i\)’s) returned by an SQL query, where \(r\) is a non-negative integer.

- \(ICSet = \{IC_1, IC_2, \ldots, IC_r\}\): The set of corresponding integrity codes to the dataset \(D\). Each \(IC_i\) is the integrity code for each data item \(d_i\).

- \(s\): Each integrity code IC has a unique serial number \(s\). The construction of ICs ensures that \(s\) is an unforgeable identity of that IC.

- \(S = \{s_1, s_2, \ldots, s_r\}\): The set of corresponding serial numbers to the \(ICSet\) and thus to the dataset \(D\). That is, each \(s_i \in S\) is the corresponding serial number to each \(IC_i \in ICSet\) and thus to each \(d_i \in D\).

- \(m\): A collection of information related to \(d\), including \(d\) itself and a unique serial number \(s\), that are used to construct the integrity code for the data item \(d\).
• $M = \{m_1, m_2, \ldots, m_r\}$: A set of $m_i$'s, returned by an ICDB query (converted from an SQL query) so that the ICDB client has enough information to regenerate the ICs.

### 4.2 Integrity Codes

In the ICDB model, we protect data integrity by assuring their correctness and freshness. To accomplish this, our approach is to insert an integrity code along with a serial number into the database for each data item to be protected. By checking the integrity code to see whether it matches with the data to be protected, we are able to ensure the data's correctness since the integrity code is constructed using a secret key known only by the data owner. Nobody, except the data owner, is able to modify the data and generate a matching integrity code for it. By checking the serial number, we are able to ensure the data's freshness since the serial numbers for all removed data will be marked as invalid (not fresh) in the data owner’s record. The approach we use to track valid serial numbers is similar to that of the X.509 standard on maintaining the validity of public key certificates.

Given a data item $d$ and a secret key $k$ (known only to the data owner), the Integrity Code $IC(d)$ of data $d$ is constructed using an IC generating function $G(m, k)$:

$$IC(d) = G(m, k)$$  \hfill (4.1)

where $m$ is a collection of information related to $d$ as described in Section 4.1. In each level of integrity protection granularity, $m$ contains a different collection of information. In OCF (One Code Per Field) granularity, $m$ is formed by concatenating the following information:
\[ m = T.A(e) + D + T.K(e) + A + T + s = d + D + T.K(e) + A + T + s \] (4.2)

where \( T \) is a table name, \( A \) is an attribute (field) name, \( s \) is a unique serial number assigned to the code, + means concatenation, \( T.A(e) = d \) is the attribute \( A \)'s value of an entity \( e \) in table \( T \), a delimiter \( D \) for prohibiting the replacement attack as described in Section 5.2.2, \( T.K(e) \) is the primary key value of the same entity \( e \) (i.e., the primary key in the same tuple as \( T.A(e) \) is located). In OCT (One Code per Tuple) granularity, \( m \) is organized as:

\[ m = T.A_1(e) + D + T.A_2(e) + \ldots + T.A_n(e) + T + s = d + T + s \] (4.3)

where \( T \) is a table name, \( s \) is the unique serial number, + means concatenation, and \( T.A_1(e), \ldots, T.A_n(e) \) are the field values in a tuple (or in other words, the attribute \( A_1 \)'s, \( A_2 \)'s, \( \ldots \), \( A_n \)'s values for an entity \( e \), respectively) and \( D \) is a delimiter in between each field value for prohibiting the replacement attack as described in Section 5.2.2. In OCT the data item to be protected is the entire tuple, i.e., \( d = T.A_1(e) + T.A_2(e) + \ldots + T.A_n(e) \).

Sections 4.3.1 and 4.3.2 will describe in detail why \( m \) should contain a different collection of information in OCF and OCT. Equation 4.1 above shows an integrity code's construction, where the function \( G \) can be one of three different cryptographic algorithms which will be described later. On top of the integrity code, let’s define an integrity unit for a data item \( d \) as a pair of \( < IC(d), s > \), i.e., an integrity unit is an integrity code and the corresponding serial number (in clear).

The following sections describe the three different cryptographic algorithms used to generate the integrity code in this thesis, and the management of serial numbers.
4.2.1 Integrity Code Generating Algorithms

To maintain correctness, an IC must be cryptographically strong against forgery. With this in mind, we chose to use three different cryptographic algorithms to generate the integrity code: RSA, CMAC AES, and HMAC SHA [24], [7], [4], [34] [35]. In the literature, both CMAC (Cipher-based Message Authentication Code) and HMAC (Hash-based Message Authentication Code) algorithms have been designed with the objective that anyone without the secret key is unable to produce a valid (matching) MAC. A matching MAC to a data item means that anyone with the knowledge of the secret key can verify the validity of the MAC which in turn proves the integrity of the data item. In this project, we chose to use CMAC AES and HMAC SHA provided by the BouncyCastle Java library [22] to generate integrity codes in our experiments. Compared to RSA, these MAC algorithms have the advantages of smaller-size integrity codes and more efficient code generation. Despite RSA’s large memory and performance footprint, we have included RSA in our experiments because its homomorphic property [23] is useful in aggregate verification and also as a comparison to other notable publications involving the use of RSA [24], [7] and [4].

RSA

RSA has been used widely in many public-key cryptosystems. The public key is used for the encryption (or signature verification) while the private key is used to decrypt the encrypted data (or generate a signature). If \( y \) is an RSA private key and \((N, x)\) is the public key, for a message \( m \) (or a collection of information related to a data item \( d \) as described in Equations 4.2 or 4.3), the signature \( \text{Sig}(m) \) can be used as the integrity code to protect the data item \( d \):
\[ \text{Sig}(m) = m^y \mod N \quad (4.4) \]

The signature \( \text{Sig}(m) \) can be verified by checking:

\[ m^x \equiv \text{Sig}(m)^x \mod N \quad (4.5) \]

The homomorphic property allows the operations to be made on the ciphers without the need of decryption [30]. RSA has the multiplicative homomorphic property which allows multiplication operations to be made on the integrity code (signature). The result of these operations is equal to the integrity code of the product of the messages (or data) as shown in the the equations below:

\[ \text{Sig}(m_1) = m_1^y \mod N \quad (4.6) \]

\[ \text{Sig}(m_2) = m_2^y \mod N \quad (4.7) \]

\[ \text{Sig}(m_1) \ast \text{Sig}(m_2) = (m_1 \ast m_2)^y \mod N = \text{Sig}(m_1 \ast m_2) \quad (4.8) \]

This RSA homomorphic property [23] allows the ICDB cloud application in our ICDB model to generate a single Aggregate Integrity Code (AIC) from many individual ICs (which will be described in section 4.4), which can reduce the integrity verification process’s computational complexity in the ICDB client. Furthermore, checking the integrity of a queried dataset as a whole in the AV mode, the ICDB client only requires one AIC, rather than requiring all individual ICs for all data items in the dataset, thus greatly reducing the network load.

There are other public key algorithms such as Elgamal and Paillier [8] [9], which also have the homomorphic property. We chose RSA partially because the research
work in [33] have found that RSA takes less time for encryption and decryption compared to Elgamal and Paillier, though RSA’s key generation may take a longer time. We use RSA in this research only for integrity code generation/verification and not for the purpose of protecting confidentiality. The time to generate keys is not an issue since the data owner (ICDB client) will not share his/her RSA keys (both public and private keys) with anyone: neither the cloud server nor the ICDB cloud application. Thus, the keys can be generated offline. The data owner uses the private key to generate integrity codes (signatures) and uses the public key to verify the integrity codes along with data returned from clouds. The only value shared with the cloud server and the ICDB cloud application is the RSA modulus $N$, which will be used in modular multiplication of multiple ICs to generate one Aggregate Integrity Code (AIC). Because the public key is not shared with anyone, the known attacks on unpadded RSA are not possible. The use of unique serial number means that all data items (even with the same value) have a unique integrity code.

**Cipher-based Message Authentication Code (CMAC)**

One type of MAC suited for generating integrity code is a CMAC, using AES-128 [6] as its backing cipher. CMAC is a technique for constructing a Message Authentication Code from a block cipher much like CBC-MAC [34]. It provides assurance of both the authentication and integrity. CMAC is a variation of CBC-MAC and fixes the security vulnerabilities such as the variable-length attack in CBC-MAC. An attacker who knows the correct message-tag in CBC-MAC pairs for two messages $(m, t)$ and $(m', t')$ can generate a third message $m''$ whose CBC-MAC will also be $t'$. These security issues are fixed in CMAC and hence we chose it as one of our integrity code generating algorithms.
Keyed-Hash Message Authentication Code (HMAC)

The second MAC algorithm we chose to use is HMAC, using SHA-128 as its digest. HMAC is a message authentication code that uses a cryptographic hash and a secret key. So HMAC can simultaneously verify both integrity and authentication. Hash functions are used in various prior works [25] for integrity protection. Hash algorithms such as SHA-1 and MD5 have been attacked in various ways [36]. One way is with length extension attacks where the attacker uses $\text{Hash}(m_1)$ and length of $m_1$ to generate $\text{Hash}(m_1\|m_2)$. Since HMAC doesn’t use the construction $\text{Hash}(key\|message)$, HMAC is not subject to length extension attacks [35]. The cryptographic strength of HMAC depends on the properties of underlying hash function. The construction of HMAC is described as:

$$\text{HMAC}(k, m) = \text{H}((k' \oplus \text{opad}) + \text{H}((k' \oplus \text{ipad}) + m)) \tag{4.9}$$

Where, $H$ is a cryptographic hash function, $k$ is the secret key, $m$ is the message, $k'$ is a secret key derived from the original key $k$, $+ \text{ denotes concatenation, } \oplus \text{ denotes exclusive or (XOR), } \text{opad} \text{ is the outer padding and } \text{ipad} \text{ is the inner padding}$

4.2.2 Serial Number Construction

In addition to the integrity code, a unique serial number is generated and stored along with the IC to ensure the freshness of the protected data. The data owner must keep an Integrity Code Revocation List (ICRL) locally, which contains those serial numbers that are revoked/invalid. When the data owner removes or updates a piece of data, the serial number corresponding to the data (and thus the integrity code) should be listed or marked as revoked/invalid in the ICRL file. Thus, if a
query were to return a data entry with a verifiable integrity code but with a serial number marked as revoked, the returned data is not fresh even though it is correct (not forged). The data owner will be able to detect invalid serial numbers for the data returned from the cloud by simply looking up the local ICRL file.

In case the list of revoked serial numbers grows, ICRL size can be reduced by applying the same technique used in the X.509 standard (this standard is used in keeping the size of the public key certificate revocation list small). Using this technique, ICDB maintains first valid serial number $S_f$. All the serial numbers less than $S_f$ are invalid. Here are the steps to reduce the size of the ICRL:

- Select a new first valid serial number $S'_f$
- All valid integrity codes in the ICDB whose serial number is less than $S'_f$ need to be renewed.
- Fetch the corresponding data and ICs for those serials to be renewed. Regenerate ICs with new serials (greater than $S'_f$).
- The updated ICRL now contains only those revoked serial numbers greater than or equal to $S'_f$, listed in an ascending order. Those revoked serial numbers less than $S'_f$ will be no longer kept in the list, and thus the ICRL size is reduced.

### 4.3 Granularity Schemes

We have studied and experimentally bench-marked two different granularity levels of integrity protection, henceforth referred to as a Granularity Scheme. These two schemes are One Code per Field (OCF) and One Code per Tuple (OCT). Both
schemes specify how data will be grouped together to construct the integrity code so that the correctness and freshness of the data can be guaranteed.

4.3.1 One Code per Field (OCF)

The OCF scheme specifies that for every field (attribute) data, there must exist a corresponding Integrity Code (IC). In other words, for every field containing user data in a table, there must exist a field to store the corresponding ICs. Furthermore, another field is also required to store the corresponding serial numbers. Table 4.1 below illustrates this ICDB OCF scheme:

<table>
<thead>
<tr>
<th>dept_no</th>
<th>dept_name</th>
<th>dept_no_ic</th>
<th>dept_no_serial</th>
<th>dept_name_ic</th>
<th>dept_name_serial</th>
</tr>
</thead>
<tbody>
<tr>
<td>d001</td>
<td>Marketing</td>
<td>IC(d001)</td>
<td>1135980686</td>
<td>IC(Marketing)</td>
<td>1135980687</td>
</tr>
<tr>
<td>d002</td>
<td>Finance</td>
<td>IC(d002)</td>
<td>1135980682</td>
<td>IC(Finance)</td>
<td>1135980683</td>
</tr>
</tbody>
</table>

Here, the OCF integrity code $IC_{OCF}(d)$ for a user data $d$ is defined by the following IC generating function:

$$IC_{OCF}(d) = G(m, k) = G(T.A(e) + D + T.K(e) + A + T + s, k)$$ (4.10)

where the symbol + means concatenation and $m$ is a collection of information related to the user data $d$, including the data $d = T.A(e)$ itself (a field $A$’s value of an entity $e$ in table $T$), a delimiter $D$ for prohibiting the replacement attack as described in 5.2.2, primary key value $T.K(e)$ of the same entity $e$ (i.e., the key value in the same tuple where $T.A(e)$ is located), the name of the field $A$, the name of the table $T$, a unique serial number $s$ assigned to the code. $G(m, k)$ is the IC generating function
using the data owner’s secret key $k$. We use different information related to $d$ in the collection $m$ for preventing/detecting various attacks described in section 5.2. Each of the entity values is tied with their corresponding Primary Key value. This releases us from the burden of generating ICs for all the combinations of attributes in the table. For example, a Database University may contain a Table student with attributes FirstName, LastName, SSN, StudentNumber, Sex, Birthdate, Class, Degree and Address. The query fetching the values for attributes FirstName and Address would require the IC to be generated by combining the same two attributes; this is also true for all other combinations of attributes. The use of the primary key value $T.K(e)$ along with each of the attribute values/data $T.A(e)$ for generating the Integrity Code helps in tying each of the attributes with their respective Primary Keys - which in turn ties them with each other. This helps us get rid of multiple ICs to be generated based on the different combination of attributes. In addition, the use of attribute name $A$ along with the data $d$, detects a substitution attack in the particular table since different data items with the same values cannot be related to the same Primary Key value and to the same attribute name. However, the attack may not be detected if another table in the same database contains the same attribute name with same data value and primary key value. To ensure the detection of a substitution attack in such cases, we use the table name $T$ in the collection $m$, since a database cannot have multiple tables with the same name.

This scheme allows for fine-grained integrity protection down-to each individual field. If a field’s data does not match its integrity code, the data owner knows that this field entry is invalid. In order for the ICDB client (on behalf on the data owner) to verify the integrity of the returned data, the query to be issued by the ICDB client to the cloud database server needs to request information in addition to the data
itself such including the corresponding integrity codes, key values, serial numbers, etc. Thus, the ICDB client needs to convert an original SQL query to an ICDB query to retrieve enough information for integrity verification.

**OCF Query Conversion in the ICDB Basic Model**

In the ICDB Basic model, there are only two entities: the CDS and the ICDB client.

Once the ICs are generated and inserted into the database, the ICDB client is responsible for converting any SQL query [26] to an ICDB query. The query conversion can be explained as the Algorithm A below:

**Algorithm A: OCF-Basic Query Conversion.**

Input (an SQL query):

```
SELECT A_1, A_2, \ldots A_r, FROM T_1, T_2, \ldots T_m WHERE C_1 and/or \ldots C_n;
```

Output (an OCF-Basic query):

```
SELECT A_1, A_2, \ldots A_r, K_1, K_2, \ldots K_m, B_1, B_2, \ldots B_k, 
   IC(A_1), IC(A_2), \ldots IC(A_r), S(A_1), S(A_2), \ldots S(A_r),
   IC(k_1), IC(k_2), \ldots IC(k_m), S(k_1), S(k_2), \ldots S(k_m),
   IC(B_1), IC(B_2), \ldots IC(B_k), S(B_1), S(B_2), \ldots S(B_k)
FROM T_1, T_2, \ldots T_m
WHERE C_1 and/or \ldots C_n;
```
where $A_1 \ldots A_r$ are attribute names, $K_1 \ldots K_m$ are key attribute names for tables $T_1 \ldots T_m$ respectively, $B_1 \ldots B_k$ are attribute names that appear in the conditions $C_1 \ldots C_n$ within the WHERE clause. In order to verify the integrity of these attribute values listed above, we also need to retrieve the associated Integrity Unit (IC and serial number), one for each of these attributes $A_1 \ldots A_r$, $K_1 \ldots K_m$, $B_1 \ldots B_k$ respectively.

We use example SQL queries Query 1 and Query 2 over an 'Employees' database to demonstrate how to apply the above query conversion algorithm, where Query 1 is a selection query from a single table and Query 2 is a query with a join operation from multiple tables. The 'Employees' database chosen is a MySQL sample database publicly available online: Employees (v1.0.6) [21]. It has a total of six tables and consists of 4 million records. These six tables are departments, dept_emp, dept_manager, employees, salaries and titles, of which the table salaries is the largest. The database schema is provided in Appendix A.

**Query 1 - Retrieve the salary of employees whose employee number is 1001:**

The original SQL query for Query 1 is

```
SELECT salary FROM salaries WHERE emp_no = 1001;
```

Applying the query conversion Algorithm A, the converted OCF-Basic query is:

```
SELECT salary, emp_no, from_date
    salary_IC, salary_serial, emp_no_IC, emp_no_serial,
    from_date_IC, from_date_serial
```

FROM salaries
WHERE emp_no = 1001;

The resulting ICDB query, in addition to 'salary' attribute, also selects 'emp_no' and 'from_date' attributes along with their integrity unit (IC and serial). This is because these additional attributes are the primary key for the 'salaries' table, and is required for the integrity verification process. All the example queries discussed in the following sections will follow the same pattern of fetching additional related information to the data based on the query conversion Algorithm A described earlier.

In addition to a selection query from a single table, Algorithm A also describes how to convert a join query from multiple tables. We use the Query 2 below to demonstrate the conversion:

**Query 2 - For each employee, retrieve the department name in which they are working and the start and end dates of their time working there.**

The original SQL join query for Query 2 is:

```
SELECT departments.dept_name, dept_emp.from_date, dept_emp.to_date
FROM departments, dept_emp
WHERE departments.dept_no = dept_emp.dept_no AND
    departments.dept_no = 'd002';
```

The converted OCF-Basic query for Query 2 is:

```
SELECT departments.dept_name, dept_emp.from_date, dept_emp.to_date, 
    departments.dept_no, dept_emp.emp_no, 
    IC(departments.dept_name), S(departments.dept_name),
```

In addition to selection and join queries, the ICDB was also designed to handle SQL functional queries. For any aggregate operation (sum, min, max, average, count), all the data used to determine the result of an aggregate function has to be fetched back to the ICDB client so that the ICDB client is able to perform the aggregate operation over these fetched data on behalf of the data owner. ICDB was designed in this way because in the ICDB models the cloud database server is not trustworthy to directly perform all the SQL queries, including the aggregate queries. Thus, the ICDB client will need to convert the original SQL aggregate query to an OCF query so that all the required data can be fetched to calculate the aggregate value. Consider an example SQL functional Query 3 below.

**Query 3 - calculates the sum of all the salaries in the 'salaries' table:**

The original SQL aggregate functional query for Query 3 is:

```
Select sum (salary) from salaries;
```

The converted OCF-Basic query for Query 3 is:
SELECT salary, emp_no, from_date
    salary_IC, salary_serial, emp_no_IC, emp_no_serial,
    from_date_IC, from_date_serial
FROM   salaries;

After all the fetched data are verified, the ICDB client can then perform the aggregate (sum) operation, i.e., all the salary data are summed. For other SQL functional queries, the query conversion and aggregate value generation processes are exactly the same as we described in this example.

**OCF Query Conversion in the ICDB DMV Model (OCF-DMV)**

In the ICDB Dual Mode Verification (DMV) model, there are three entities: the cloud database server, the ICDB cloud application, and the ICDB Client.

DMV model makes use of two different cloud services i.e, the cloud database server and the ICDB cloud application. So, each SQL query needs to be converted to separate ICDB queries for CDS and CA so that both of them can accomplish their respective tasks. The ICDB cloud application requires only the IC’s to generate an Aggregate Integrity Code (AIC). The ICDB client needs to fetch from the cloud database server all the data along with the serials to regenerate IC’s. The ICDB client also needs to fetch the AIC from the cloud application to verify the regenerated IC’s and thus verify the user data.

The DMV model has two modes (aggregate verification or detailed verification). Different queries need to be executed in different entities in different modes.
OCF Query Conversion in Aggregate Verification Mode: In an aggregate verification mode, the ICDB client has to issue a query to the cloud database server to fetch data along with the corresponding serial numbers (but not the IC’s). The ICDB client simultaneously has to issue another query to the ICDB cloud application to request the AIC of the corresponding IC’s.

The query to be issued to the cloud database server would be similar to the Algorithm A by excluding the Integrity Codes, whereas the query to be issued to the ICDB cloud application would be again similar to the Algorithm A but only retrieving the IC’s. Algorithm B below describes this query conversion.

**Algorithm B: OCF-DMV Query Conversion.**

Input (an SQL query):

```
SELECT A_1, A_2, \ldots A_r FROM T_1, T_2, \ldots T_m WHERE C_1 and/or \ldots C_n;
```

Output (an OCF-DMV query $Q_1$ to the cloud database server and an OCF-DMV query $Q_2$ to the ICDB cloud application):
OCF-DMV query \(Q_1\): to be issued to the cloud database server.

```
SELECT \(A_1, A_2, \ldots A_r, K_1, K_2, \ldots K_m, B_1, B_2, \ldots B_k\),
       \(S(A_1), S(A_2), \ldots S(A_r),\)
       \(S(k_1), S(k_2), \ldots S(k_m),\)
       \(S(B_1), S(B_2), \ldots S(B_k)\)
FROM  \(T_1, T_2, \ldots T_m\)
WHERE \(C_1\ and/or \ldots C_n;\)
```

OCF-DMV query \(Q_2\): to be issued to the ICDB cloud application.

```
SELECT \(IC(A_1), IC(A_2), \ldots IC(A_r), IC(k_1), IC(k_2), \ldots IC(k_m),\)
       \(IC(B_1), IC(B_2) \ldots IC(B_k)\)
FROM  \(T_1, T_2, \ldots T_m\)
WHERE \(C_1\ and/or \ldots C_n;\)
```

Upon receiving OCF-DMV query \(Q_1\), the cloud database server will return the requested data and the corresponding serials to the ICDB client. However, when the ICDB cloud application receives OCF-DMV query \(Q_2\), it will forward the query to the database server to retrieve all the ICs so that it can generate the AIC. Finally, the ICDB cloud application will forward the single AIC to the ICDB client. After receiving the AIC and all the data and serials (from \(Q_1\)), the ICDB client is able to check the data integrity by performing an aggregate verification. The AIC generation in the ICDB cloud application and the aggregate verification process in the ICDB client will be described later in Section 4.4.
OCF Query Conversion in Detailed Verification Mode: When aggregate verification fails, it means there is something wrong in the returned dataset as a whole. The ICDB client can either discard the entire dataset since it is incorrect, or the ICDB client can choose the option to verify the data in detail to identify which particular data’s integrity has been compromised. In case the ICDB decides not to take the detailed verification option, there is no need to download all the IC’s from the cloud and thus reduce the network overhead. If the ICDB client chooses to perform detailed verification, then it has to issue the query $Q_2$ again to the cloud database server to fetch all the individual IC’s, which is in fact the same query issued to the cloud application in the aggregate verification mode.

4.3.2 One Code per Tuple (OCT)

When bench marking the performance of the OCF schemes, results have indicated that the average ratio between an Integrity Code size and its corresponding data field size is high (see experimental results). The storage usage is inefficient since each data field usually holds only a small amount of data, but it needs to have a much larger size of storage to store the corresponding IC. For example, a tuple with an attribute First_Name can contain the data, George, which is only 6 characters (48 bits) compared to an Integrity Code (IC) which are usually recommended to have a minimum of 128 bits (using MAC algorithms) or at least 1028 bits (using the RSA algorithm) to be safe from forgery.

To make the storage usage more efficient, instead of one code per field (OCF), we can generate only one code per tuple (OCT). A tuple usually contains multiple fields, and therefore in many cases, the data size (entire tuple) is more than the 128-bit integrity code size (if using MAC algorithms).
To generate an integrity code for an entire tuple, all the data in the tuple can be concatenated before passing it to the IC-generating function, as shown in the example table 4.2.

**Table 4.2:** An example table for OCT granularity

<table>
<thead>
<tr>
<th>dept_no</th>
<th>dept_name</th>
<th>IC</th>
<th>serial</th>
</tr>
</thead>
<tbody>
<tr>
<td>d001</td>
<td>Marketing</td>
<td>IC(d001,Marketing,1135980685)</td>
<td>1135980685</td>
</tr>
<tr>
<td>d002</td>
<td>Finance</td>
<td>IC(d002,Finance,1135980689)</td>
<td>1135980689</td>
</tr>
</tbody>
</table>

IC unit is an ordered pair consisting of an IC and serial number, where the IC in OCT schemes is defined by the following IC-generating function:

\[
IC_{OCT}(d) = G(m, k) = G(T.A_1(e) + D + T.A_2(e) + \ldots + T.A_n(e) + T + s, k) = G(d + T + s, k)
\]  

(4.11)

where the symbol + means concatenation. \(T.A_1(e), \ldots, T.A_n(e)\) are the field values in a tuple (or in other words, attributes values for an entity \(e\)). \(D\) is a delimitier in between each field values for prohibiting the replacement attack as described in 5.2.2, and in OCT the data item to be protected is \(d = T.A_1(e) + T.A_2(e) + \ldots + T.A_n(e)\) (i.e., the entire tuple). The information collection \(m = d + T + s\) represents the data item \(d\) along with the table name \(T\) and a unique serial number \(s\) assigned to the code.

The purpose of including additional information related to the data item (tuple) \(d\) in the collection \(m\) is to ensure the uniqueness of \(m\) and also to prevent/detect various attacks described in section 5.2. Although each tuple in a table \(T\) is unique within the same table due to the Primary Key constraint in the SQL standard, another table \(T'\) in the same database could have the exactly same tuple values, i.e., a tuple in table \(T'\) with different attributes but with the same values, compared to the tuple in table \(T\). So, including table name \(T\) in the IC construction helps in detecting a substitution.
attack in this scenario. $G(m, k)$ is the integrity code generating function on $m$ using data owners secret key $k$.

By contrast with OCT it is not possible to determine whether a particular field entry is invalid since there is only one integrity code for the entire tuple. If the integrity code verification fails, it means some field entry in the tuple is invalid. It has no way to know which field entry in the tuple causes the verification failure. Thus OCT schemes relinquish fine-grained detection in exchange for a smaller memory footprint, and fewer ICs that need to be generated.

**OCT Query Conversion in the ICDB Basic Model**

In the ICDB Basic model, with only two entities, an SQL query needs to be converted to an ICDB OCT query to request enough information for the ICDB client to verify. Algorithm C below describes the query conversion.

**Algorithm C: OCT-Basic Query Conversion.**

Input (an SQL query):

```
SELECT A_1, A_2, \ldots A_r FROM T_1, T_2, \ldots T_m WHERE C_1 and/or \ldots C_n;
```

Output (an OCT-Basic query):

```
SELECT A_1, \ldots A_t, IC_1, \ldots IC_m, S_1, \ldots S_m (Actually is SELECT *)
FROM T_1, T_2, \ldots T_m
WHERE C_1 and/or...C_n;
```

where $T_1 \ldots T_m$ are tables from where the data are to fetched, each $IC_i$ and $S_i$, $\forall i = 1, 2, \ldots m$, are the IC and serial in each table $T_i$, $C_1 \ldots C_n$ are conditions within
the WHERE clause. $A_1, \ldots, A_r$ in the input query are some attributes from all tables $T_1, \ldots T_m$, whereas $A_1, \ldots, A_t$ in the output query are all attributes for all tables $T_1, \ldots, T_m$. To regenerate IC's, it is necessary to select all attributes of the entire tuple. An example of OCT-Basic query conversion is provided using the same Query 1 used in the OCF query conversion example.

The original SQL query for Query 1 is:

```
SELECT salary FROM salaries WHERE emp_no = 1001;
```

Applying the query conversion Algorithm C, the converted OCT-Basic query is:

```
SELECT emp_no, salary, from_date, to_date, salaries.IC, salaries.serial
FROM salaries
WHERE emp_no = 1001;
```

The converted ICDB query for OCT in this example includes additional attributes such as `emp_no`, `from_date`, `to_date` in addition to the requested attribute `salary`. This is because, to regenerate the IC in the integrity verification process by the ICDB client, all attributes of the table (the entire tuple) are required to be retrieved.

As in OCF schemes, an SQL join query from multiple tables can be converted in OCT scheme as well. An ICDB join query would require all the attributes from all the tables that are joined together. In addition to a selection query from a single table, Algorithm C also describes how to convert a join query from multiple tables.

We use the same Query 2 mentioned in OCF query conversion to demonstrate the conversion: The original SQL join query for Query 2 is:
SELECT departments.dept_name, dept_emp.from_date, dept_emp.to_date
FROM departments, dept_emp
WHERE departments.dept_no = dept_emp.dept_no AND
    departments.dept_no = 'd002';

The converted OCT-Basic query for Query 2 is:

SELECT *
FROM departments, dept_emp
WHERE departments.dept_no=dept_emp.dept_no AND
    departments.dept_no='d002';

Just as with selection and join queries, functional queries are similar to what
has been explained in the OCF query conversion. However, OCT needs to fetch all
attribute values in a tuple even for a query only requesting an aggregate value over a
single attribute. We use the example Query 3 used earlier in OCF query conversion to
demonstrate the OCT query conversion for a functional SQL query with an aggregate
operation.

The original SQL aggregate functional query for Query 3 is:

Select sum (salary) from salaries;

The converted OCT-Basic query for Query 3 is:
Upon receiving all data from the cloud, the ICDB client will need to verify the integrity correctness for all received data, including the salary data. If all data pass the integrity check, the ICDB client can perform the aggregate operation over the received salary data and then present the aggregate result to the user.

**OCT Query Conversion in the ICDB DMV Model (OCT-DMV)**

In the DMV model, the communication is among three entities, which are the ICDB client, the ICDB cloud application and the cloud database server.

**OCT Query Conversion in Aggregate Verification Mode:** Similar to OCF schemes, for the aggregate verification mode in OCT schemes, the ICDB client has to issue a query $Q_1$ to the cloud database server to fetch all those tuples that match the query conditions along with the corresponding serial numbers (but not the IC’s). At the same time, the ICDB client also has to issue another query $Q_2$ to the ICDB cloud application to request an AIC, which is an aggregate integrity code for all the IC’s corresponding to those tuples retrieved in query $Q_1$.

The query $Q_1$ to be issued to the cloud database server would be similar to the one in Algorithm C by excluding the integrity codes, whereas the query $Q_2$ to be issued to the ICDB cloud application would also be similar to the one in Algorithm C but including only the integrity codes. This query conversion is described by the Algorithm D below.
### Algorithm D: OCT-DMV Query Conversion

**Input (an SQL query):**

```
SELECT A_1, A_2, \ldots, A_r \text{ FROM } T_1, T_2, \ldots, T_m \text{ WHERE } C_1 \text{ and/or } \ldots C_n;
```

**Output (an OCT-DMV query to the cloud database server and an OCT-DMV query to the ICDB cloud application):**

**OCT-DMV query \(Q_1\):** to be issued to the cloud database server

```
SELECT A_1, \ldots, A_t, S_1, \ldots, S_m 
FROM T_1, T_2, \ldots, T_m 
WHERE C_1 \text{ and/or } \ldots C_n;
```

**OCT-DMV query \(Q_2\):** to be issued to the ICDB cloud application

```
SELECT IC_1, \ldots, IC_m 
FROM T_1, T_2, \ldots, T_m 
WHERE C_1 \text{ and/or } \ldots C_n;
```

As in the OCF schemes, upon receiving OCT-DMV query \(Q_1\), the cloud database server will return the requested data and the corresponding serials to the ICDB client. When the ICDB cloud application receives OCT-DMV query \(Q_2\), it will forward the query to the database server to retrieve all the ICs so that it can generate the AIC. Finally, the ICDB cloud application will forward the AIC to the ICDB client. After receiving the AIC and all the data and serials (from \(Q_1\)), the ICDB client is able to
check the data integrity by performing an aggregate verification. The AIC generation in the ICDB cloud application and the aggregate verification process in the ICDB client will be described later in Section 4.4.

**OCT Query Conversion in Detailed Verification Mode:** When aggregate verification fails, it means there is something wrong in the returned dataset (a set of tuples) as a whole. Similar to the OCF-DMV scheme, the OCT-DMV scheme also provides the data owner with an option to either discard the dataset and stop, or verify the data further in the detail mode to identify which particular tuple has been compromised. To verify in the detail mode, the ICDB client has to issue the same query $Q_2$ to the cloud database server again to fetch all the IC’s so that it can verify each individual tuple.

### 4.4 Aggregate Integrity Code Generation and Verification

We use different approaches to construct the Aggregate Integrity Code (AIC) depending on which cryptographic algorithm is used in generating ICs. In the previous examples, the query $Q_2$ fetches all the ICs to the ICDB cloud application from the cloud database server. If ICs are generated using the RSA algorithm, its homomorphic property allows multiplication to be performed on ciphers (or ICs here) without decrypting it. Making use of this property, the ICDB cloud application can modularly multiply all the fetched ICs together to generate a single AIC. According to the variable definitions in Section 4.1, let’s assume query $Q_1$ returns a set of information collections $M = \{m_1, m_2, \ldots, m_r\}$, which includes a dataset $D = \{d_1, d_2, \ldots, d_r\}$ and their serial numbers $S = \{s_1, s_2, \ldots, s_r\}$, to the ICDB client. We also assume
query $Q_2$ fetches the corresponding $ICSet = \{IC_1, IC_2, \ldots, IC_r\}$ to the ICDB cloud application. Equation 4.12 below describes the AIC generation for RSA algorithm.

$$AIC = \prod_{i=1}^{r} IC_i \mod N$$  \hspace{1cm} (4.12)

If ICs are generated by MAC algorithms (CMAC AES or HMAC SHA), the ICDB cloud application can generate a single AIC by concatenating all the ICs fetched from the cloud database server and then applying a hashing algorithm (SHA-256) to the concatenated ICs. The produced digest or result is the AIC. Equation 4.13 describes the AIC generation for MAC algorithms.

$$AIC = H(IC_1 + IC_2 + \ldots + IC_r)$$  \hspace{1cm} (4.13)

where $H()$ is a hashing function implementing the SHA-256 hash algorithm.

After the AIC is generated and returned to the ICDB client, along with all the data returned by the query $Q_2$, the ICDB client can then perform the AV mode verification. In the AV mode verification, if RSA is the IC generating function, the ICDB client has to compute the aggregate data by modularly multiplying all $m_i$, for all data items $d_i$ in the return dataset. The AIC can be regenerated by applying the RSA algorithm to the aggregate data. Equation 4.14 below describes this AIC regeneration by the ICDB client.

$$AIC = Sig(\prod_{i=1}^{r} m_i \mod N) = (\prod_{i=1}^{r} m_i)^y \mod N$$  \hspace{1cm} (4.14)

According to Equation 4.4, each $IC_i = Sig(m_i) = (m_i)^y \mod N$, where $y$ is the RSA private key and $N$ is the RSA public modulus. The AIC constructed in Equation 4.12
by the ICDB cloud application should match with the regenerated AIC in Equation 4.14 by the ICDB client. The following derivation shows this.

**Original AIC**

\[
\text{Original AIC} = \prod_{i=1}^{r} IC_i \mod N
\]

\[
= IC_1 \times IC_2 \times \ldots \times IC_r \mod N
\]

\[
= m_1^y \times m_2^y \times \ldots \times m_r^y \mod N
\]

\[
= (m_1 \times m_2 \times \ldots m_r)^y \mod N
\]

\[
= (\prod_{i=1}^{r} m_i)^y \mod N
\]

\[
= \text{Regenerated AIC}
\]

IF the IC generation function is one of the MAC algorithms (CMAC AES or HMAC SHA), the ICDB client has to regenerate all the integrity codes for all the returned data \( m_i \) and then regenerate the AIC from all the regenerated ICs. Equation 4.15 shows this AIC regeneration by the ICDB client.

\[AIC = H(MAC(m_1) + MAC(m_2) + \ldots + MAC(m_r)) \quad (4.15)\]

where \( H() \) is an SHA-256 hash function and \( MAC() \) is either the CMAC AES or the HMAC SHA algorithm. The original AIC based on Equation 4.13 generated by the ICDB cloud application should match the AIC based on Equation 4.15 regenerated by the ICDB client since each \( IC_i = MAC(m_i) \) and thus

**Original AIC**

\[
\text{Original AIC} = H(IC_1 + IC_2 + \ldots + IC_r)
\]

\[
= H(MAC(m_1) + MAC(m_2) + \ldots + MAC(m_r))
\]

\[
= \text{Regenerated AIC}
\]

Finally if the regenerated AIC does not match with the returned AIC from the ICDB cloud application, an integrity violation message will be sent to the user.
CHAPTER 5

EXPERIMENTAL RESULTS AND ANALYSIS

5.1 Hardware and Software Used

All of our ICDB prototype implementation, testing, and benchmarking were performed at Boise State University’s Onyx server (See Appendix B for specifications). MySQL (MariaDB) was used as the underlying database management system, using InnoDB as its database engine. For database conversion (initial IC code generation and insertion), query conversion, and query verification, we have implemented all the modules with JAVA SE 1.8 and made them available in [16].

To test against a database with real data, we have chosen a sample MySQL database publicly available online: Employees (v1.0.6) [21] (See Appendix A). The database has size of 196.4 MB with close to 4 million tuples, which includes six tables employees, departments, dept_manager, dept_emp, titles, salaries. Before each experiment, the databases were returned to their original state for fair comparison.

We tested all possible combinations of RSA, CMAC-AES, and HMAC-SHA with different granularities of protection levels in OCF and OCT schemes, for a total of 6 different ICDB implementations in each of the ICDB Basic model and in the ICDB DMV model. RSA uses a 1024-bit (fixed size) output for all integrity codes. AES outputs a 128-bit IC using a CMAC, SHA outputs a 128-bit IC using an HMAC, and
both are given a 128-bit key as input.

As mentioned in the introduction, the metrics to evaluate the proposed ICDB approach are: integrity protection, memory penalty, and performance penalty. The following sections show the results in evaluating each metric over a standard database and its converted ICDB counterpart. All results in this section are based on the Employees database provided by MySQL [21].

5.2 Integrity Protection

If an ICDB cannot verify the integrity of the protected data, then there is no advantage of using it over a standard database. The highest priority of this thesis is to test whether the ICDB is able to detect various kinds of malicious attacks. The following attacks have been tested with our ICDB prototype which also answers the Question 1 stated in section 1.1.1.

5.2.1 Forgery Attack

A forgery attack is an attack that mutates or alters fields in a database. This could involve either the manipulation of the data itself, or its integrity code. Since there is no feasible way within a reasonable amount of time to generate an IC without the private key of the data owner, the ICDB should be able to detect this kind of attack.

For testing, we modified some attribute values and a few IC codes. Our ICDB prototype was able to detect such modifications.
5.2.2 Substitution Attack

A substitution attack modifies fields by replacing them with other existing fields within the database. This could include copying and replacing a field somewhere else, moving data around, or swapping data from the same row/column. In OCT schemes, the IC is cryptographically generated by concatenating the entire tuple with the table name and thus is unique. It is not possible to substitute among tuples without being detected. It is also not possible to substitute some fields within a tuple without being detected since the IC is generated based on the original tuple as a whole. In OCF schemes, since primary key (row identity), attribute name (column identity) and the table name concatenated together with the field data will make the collection of information unique and thus make the integrity code unique, it is not possible to swap field’s value or an IC from other places (e.g., different rows and columns) without being detected.

To test the substitution attack, we swapped the field values and ICs of attributes from_date and to_date in salaries table. Our ICDB prototype was able to detect this attack due to the uniqueness of the attribute name within a table. Also, swapping the values of attributes from_date and to_date of two tables salaries and dept_manager was detected due to the uniqueness maintained by the table name within a database, though the four attributes have the same data type.

Replacement Attack:

A replacement attack in OCT schemes moves the value of one field (or more) and concatenates that value with the value of a neighboring field. A replacement attack in OCF schemes moves part of the Primary Key value to another attribute.
In a replacement attack on an OCT scheme, the overall data within a tuple will not be altered but the position of some data will be changed. For example, certain characters from one attribute value may be moved and attached to some other attribute value.

To test for a replacement attack on an OCT granularity scheme, a prefix substring from attribute last_name was moved to the end of first_name in employees table, keeping the overall tuple data unaltered. The presence of the inter-field delimiter affects the calculation of the IC, making the IC of the changed row different from the IC on the original data.

To test the attack for an OCF granularity scheme, a number from the value in the Primary Key emp_no was moved to the end of value in that row’s salary attribute. The modified salary value, its IC and serial were then copied to the corresponding attributes in a different tuple whose Primary Key value is equal to the value of the now-modified Primary Key. For an example replacement attack, an emp_no with value '11001' and salary with value '11677' was modified by moving the first number '1' from '11001' to the end of '11677' creating a new salary value '116771'. This new salary value along with its corresponding IC and serial were copied to a tuple with primary key '1001'. In this case as well, the use of a delimiter in between the Attribute value and the Primary Key aided in detecting the replacement of the numbers.

5.2.3 Old-Data Attack

An old-data attack means the cloud returns data which has been previously updated or deleted. Because each IC contains a serial number and the data owner has a private list of revoked serial numbers in their ICRL, old-data attacks can be detected if the data owner keeps the ICRL up-to-date.
To test for an old-data attack, several tuples in the salaries table were deleted by the ICDB client. These deleted rows were then re-inserted (with the same data value, serial and IC) into the ICDB database. In a normal process, adding previously deleted data back into the database, a new serial number should be used. In this test, we used the old serial number to re-insert the deleted data for the purpose of testing old-data attacks. The reinserted old data was successfully detected by scanning the ICRL file.

5.2.4 Tuple Insertion Attack

A tuple insertion attack introduces new rows in the database. This could be a completely new fake tuple, a tuple containing data copies from other rows, or a duplicate of another tuple.

To test a tuple insertion attack, new rows were added to the departments table with random ICs or with ICs copied from other tuples. These modifications were also detected by our prototype.

5.2.5 Tuple Deletion Attack

A tuple deletion attack removes existing tuples from the database. Because we do not guarantee completeness, it is not possible to know whether a targeted data item is missing from a query result being fetched.
5.3 Memory Penalty

A memory penalty is an obvious trade-off for obtaining integrity protection in an ICDB since the database needs extra fields to store ICs and serial numbers. The following charts illustrate the database size increase for all six ICDB combinations with different cryptographic algorithms and different protection granularities as discussed earlier. This section answers the Question 2 stated in section 1.1.1.

To interpret the experimental result easily, we define a memory penalty rate as

\[
\text{memory penalty rate} = \frac{\text{size of the ICDB database}}{\text{size of the SQL database}} \tag{5.1}
\]

where the rate indicates the database size increase rate for an ICDB database compared to the original SQL database.

Figure 5.1 and the raw data in Table 5.1 display the size increase between the Employees DB [21] and its ICDB counterparts using MAC algorithms (CMAC and HMAC) in OCT and in OCF. Since both HMAC-SHA and CMAC-AES generate a 128-bit integrity code, the ICDB sizes are the same for both algorithms.

A quick comparison between granularity schemes reveals that the memory penalty rate for the MAC-OCT is about 1.53, while the memory penalty rate for the MAC-OCF is 3.15. This means MAC-OCF is larger in size than that of MAC-OCT by a factor of 2.05.
Figure 5.1: Database sizes: DB is the original Employees SQL database; MAC-OCT and MAC-OCF are the Employees ICDB databases using 128-bit HMAC or CMAC integrity code implementing the OCT scheme and the OCF scheme, respectively. Sizes are displayed in megabytes.

Table 5.1: Database sizes: DB is the original Employees SQL database; MAC-OCT and MAC-OCF are the Employees ICDB databases using 128-bit HMAC or CMAC integrity code implementing the OCT scheme and the OCF scheme, respectively. Sizes are displayed in Megabytes.

<table>
<thead>
<tr>
<th></th>
<th>Size (MB)</th>
<th>Memory Penalty Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB</td>
<td>196.44</td>
<td></td>
</tr>
<tr>
<td>MAC-OCT</td>
<td>300.61</td>
<td>1.53</td>
</tr>
<tr>
<td>MAC-OCF</td>
<td>619</td>
<td>3.15</td>
</tr>
</tbody>
</table>
Figure 5.2 and Table 5.2 show the size increase again, but this time using 1024-bit RSA integrity code. The memory penalty rate defined in Equation 5.1 for the RSA-OCT is about 3.88 compared to the original database, while RSA-OCF database size is increased by a memory penalty rate of 13.07. Thus, RSA-OCF resulted in a database with 3.37 times larger size than that of RSA-OCT. By the experimental results above, it is clear that OCT is a better choice over OCF with respect to memory cost.

Though different databases may have a different memory penalty rate, the experimental result based on the example Employees database still gives us an approximate idea of how much memory penalty will be if the ICDB technique is used to protect data integrity.
Table 5.2: Database sizes for the original Employees SQL database, and the converted ICDB RSA-OCT and RSA-OCF databases respectively. Sizes are displayed in Megabytes.

<table>
<thead>
<tr>
<th></th>
<th>Size (MB)</th>
<th>Memory Penalty Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB</td>
<td>196.44</td>
<td></td>
</tr>
<tr>
<td>RSA-OCT</td>
<td>762.06</td>
<td>3.88</td>
</tr>
<tr>
<td>RSA-OCF</td>
<td>2567.78</td>
<td>13.07</td>
</tr>
</tbody>
</table>
5.4 Performance Penalty

To evaluate the performance penalty, we focused on testing the performance of query processing in ICDB and in the standard SQL database. This thesis has evaluated different type of queries such as SELECT, INSERT, DELETE, UPDATE, JOIN and a Functional Query. In a standard SQL database, these type of queries are well-defined. However, to perform these queries in ICDB, the process of each query needs to be changed accordingly. Though we have described how to change the queries in ICDB in previous chapters, we will describe the query process in steps again when we present its performance penalty in later sections. This section explains the Question 3 stated in section 1.1.1.

To present the performance penalty for queries, we define a performance penalty rate as

\[
\text{performance penalty rate} = \frac{\text{time required to conduct the ICDB query}}{\text{time required to conduct the SQL query}} \tag{5.2}
\]

We will present the performance penalty rate for each type of query in later sections based on our experimental results. Note that different ICDB queries may have different query process steps.

5.4.1 Experimental Results for the Basic ICDB Model

In this thesis, We have proposed two ICDB models, the Basic model and the Dual Mode Verification (DMV) model. This section will show and analyze our experimental results in the Basic model.
SELECT

In a standard SQL database, a SELECT query only contains one step - the execution of the query in the database server, i.e., to fetch data. However, an ICDB SELECT query has three steps as described below.

ICDB SELECT query process steps in the Basic model:

1. Query conversion: The ICDB client converts an SQL SELECT query to an ICDB SELECT query as described in Algorithms A and C in Chapter 4, and then issues the query to the cloud database server.

2. Query execution (Data fetching): The cloud database server executes the ICDB SELECT query over the ICDB database, and then returns the query result.

3. Query result verification: The ICDB client verifies the returned query result.

This section presents the performance penalty of the ICDB SELECT query, using a SELECT * query on the salaries table in Employees database under the basic ICDB model described in section 3.1. The experimental results are shown in three figures and three tables, one for each algorithm (HMAC-SHA, CMAC-AES, and RSA). Each figure shows three data points a) Query process time (only query execution) for a standard database, b) Query process time for an ICDB OCT counterpart, and c) Query process time for ICDB OCF counterpart. The X-axis in the figure shows the number of tuples fetched in thousands and the Y-axis shows the time in milliseconds. Although query conversion is a part of every query process in the ICDB, it is not shown in the figures due to its negligible average time of about 25ms.
Figures 5.3, 5.4 and 5.5 (and Tables 5.3, 5.4 and 5.5) show that each data point roughly scales linearly with the amount of data queried for all three algorithms. The figures and tables also show that query result verification takes significantly more time compared to the data fetching (i.e., query execution). This result is expected since the verification process must regenerate all the ICs for the returned data. Generating an IC is time-consuming since it is a cryptographic operation. The operation is especially expensive if the RSA algorithm is used. Verification time in an OCF scheme is greater than that in an OCT scheme because there are more ICs to regenerate (for verification) in OCF than in OCT.

The Performance Penalty Rate for an ICDB SELECT query using CMAC-AES is the least of all three algorithms with 11.07 and 25.24 on average in OCT and OCF respectively. The rate is slightly higher for HMAC-SHA with 13.29 and 31.39 on average in OCT and OCF respectively. While for RSA, the penalty rate is greatest with 104.81 and 389.21 on average in OCT and OCF respectively.

The overall query process time (query conversion time, data fetching time and verification time) in OCF is a little more than 2 times, on average, than the time used in OCT with MAC algorithms. While using the RSA, the overall query processing time in OCF takes roughly 4 times more than the time used in OCT.
Figure 5.3: Using HMAC-SHA, plotted query process time in milliseconds for different number of tuples returned (in thousands) by the SELECT * query over the Employees.salaries table.

Table 5.3: Using HMAC-SHA, query process time raw data in milliseconds for different number of tuples returned (in thousands) by the SELECT * query over the Employees.salaries table.

<table>
<thead>
<tr>
<th>No. of Tuples</th>
<th>Scheme</th>
<th>Fetch</th>
<th>Verification</th>
<th>Total Query Process</th>
<th>Performance Penalty Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>50k</td>
<td>DB</td>
<td>34.8</td>
<td>0</td>
<td>34.8</td>
<td>10.99425287</td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>130.4</td>
<td>252.2</td>
<td>382.6</td>
<td>15.55813953</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>201.4</td>
<td>688.6</td>
<td>890</td>
<td>25.57471264</td>
</tr>
<tr>
<td>100k</td>
<td>DB</td>
<td>60.2</td>
<td>0</td>
<td>60.2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>354.2</td>
<td>582.4</td>
<td>936.6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>427.6</td>
<td>1492.8</td>
<td>1920.4</td>
<td></td>
</tr>
<tr>
<td>150k</td>
<td>DB</td>
<td>79.2</td>
<td>0</td>
<td>79.2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>442.6</td>
<td>691</td>
<td>1133.6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>556.8</td>
<td>2053.4</td>
<td>2610.2</td>
<td></td>
</tr>
<tr>
<td>200k</td>
<td>DB</td>
<td>107.8</td>
<td>0</td>
<td>107.8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>530.6</td>
<td>766.2</td>
<td>1296.8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>811.8</td>
<td>2621</td>
<td>3432.8</td>
<td></td>
</tr>
</tbody>
</table>
Figure 5.4: Using CMAC-AES, plotted query process time in milliseconds for different number of tuples returned (in thousands) by the SELECT * query over the Employees.salaries table.

Table 5.4: Using CMAC-AES, query process time raw data in milliseconds for different number of tuples returned (in thousands) by the SELECT * query over the Employees.salaries table.

<table>
<thead>
<tr>
<th>No. of Tuples</th>
<th>Scheme</th>
<th>Time (ms)</th>
<th>Total Query Process</th>
<th>Performance Penalty Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Fetch</td>
<td>Verification</td>
<td></td>
</tr>
<tr>
<td>50k</td>
<td>DB</td>
<td>34.8</td>
<td>0</td>
<td>34.8</td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>141</td>
<td>168.2</td>
<td>309.2</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>224.4</td>
<td>407.4</td>
<td>631.8</td>
</tr>
<tr>
<td>100k</td>
<td>DB</td>
<td>60.2</td>
<td>0</td>
<td>60.2</td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>287.2</td>
<td>428.6</td>
<td>715.8</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>414.2</td>
<td>1074.6</td>
<td>1488.8</td>
</tr>
<tr>
<td>150k</td>
<td>DB</td>
<td>79.2</td>
<td>0</td>
<td>79.2</td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>399.8</td>
<td>477</td>
<td>876.8</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>600.2</td>
<td>1513.8</td>
<td>2114</td>
</tr>
<tr>
<td>200k</td>
<td>DB</td>
<td>107.8</td>
<td>0</td>
<td>107.8</td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>540.8</td>
<td>680.4</td>
<td>1221.2</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>749.8</td>
<td>2136</td>
<td>2885.8</td>
</tr>
</tbody>
</table>
Figure 5.5: Using RSA, plotted query process time in milliseconds for different number of tuples returned (in thousands) by the SELECT * query over the Employees.salaries table.

Table 5.5: Using RSA, query process time raw data in milliseconds for different number of tuples returned (in thousands) by the SELECT * query over the Employees.salaries table.

<table>
<thead>
<tr>
<th>No. of Tuples</th>
<th>Scheme</th>
<th>Time (ms)</th>
<th>Fetch</th>
<th>Verification</th>
<th>Total Query Process</th>
<th>Performance Penalty Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>50k</td>
<td>DB</td>
<td>34.8</td>
<td>0</td>
<td>34.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>153</td>
<td>2804.2</td>
<td>2957.2</td>
<td></td>
<td>84.97701149</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>222.6</td>
<td>10712.8</td>
<td>10935.4</td>
<td></td>
<td>314.2356322</td>
</tr>
<tr>
<td>100k</td>
<td>DB</td>
<td>60.2</td>
<td>0</td>
<td>60.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>303.8</td>
<td>5759</td>
<td>6062.8</td>
<td></td>
<td>100.7109635</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>475.6</td>
<td>21409</td>
<td>21884.6</td>
<td></td>
<td>363.5315615</td>
</tr>
<tr>
<td>150k</td>
<td>DB</td>
<td>79.2</td>
<td>0</td>
<td>79.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>418</td>
<td>8371.6</td>
<td>8789.6</td>
<td></td>
<td>110.979798</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>646.4</td>
<td>32274.2</td>
<td>32920.6</td>
<td></td>
<td>415.6641414</td>
</tr>
<tr>
<td>200k</td>
<td>DB</td>
<td>107.8</td>
<td>0</td>
<td>107.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>571.2</td>
<td>11175.4</td>
<td>11746.6</td>
<td></td>
<td>108.9666048</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>1224.8</td>
<td>42820.8</td>
<td>44045.6</td>
<td></td>
<td>408.5862709</td>
</tr>
</tbody>
</table>
We can interpret the results in Figures 5.3, 5.4 and 5.5 (and Tables 5.3 5.4, and 5.5) in a different perspective. Rather than the performance penalty rate, a process rate is defined as

\[
\text{Process Rate} = \frac{\text{Total fetched user data size in MB}}{\text{Total processing time}}
\]  

(5.3)

where the total fetched user data does not include the fetched ICs and serials, and the total processing time includes the query conversion time, query execution time and the query result verification time. In short, the Process Rate means how many MB of user data can be processed (in ICDB) per second.

Figure 5.6 and the corresponding Table 5.6 show that the Process Rate for HMAC-SHA is 3.93 MB/sec in OCT and 1.68 MB/sec in OCF. CMAC-AES has similar but slightly higher rates than HMAC-SHA, with 4.78 MB/sec in OCT and 2.15 MB/sec in OCF. RSA, by comparison, processes at a much slower rate of 0.501 MB/sec in OCT and 0.134 MB/sec in OCF. Thus in the basic ICDB model, CMAC-AES in OCT scheme has the highest (best) Process Rate among the six combinations.
Figure 5.6: A chart plotted for Basic ICDB model showing the process rate, i.e., how much user data (size in MB) can be processed in a SELECT * query using three different algorithms in OCT or in OCF. The leftmost bar marked as DB is the process rate for a standard SQL database.

Table 5.6: A table with raw data for the Basic ICDB model showing the process rate, i.e., how much user data (size in MB) can be processed in a SELECT * query using three different algorithms in OCT or in OCF. The data in the DB row is the process rate for a standard SQL database.

<table>
<thead>
<tr>
<th></th>
<th>HMAC-SHA</th>
<th>CMAC-AES</th>
<th>RSA</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCT</td>
<td>3.93</td>
<td>4.78</td>
<td>0.501</td>
</tr>
<tr>
<td>OCF</td>
<td>1.68</td>
<td>2.15</td>
<td>0.134</td>
</tr>
</tbody>
</table>
INSERT

To perform an INSERT query in an ICDB:

<table>
<thead>
<tr>
<th>ICDB INSERT query process steps in the Basic model:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Query conversion: Convert an SQL INSERT query to an ICDB INSERT query. This step requires generating an IC for each data item to be inserted with a new and unique serial number. The ICDB client then issues this converted ICDB INSERT query to the cloud database server.</td>
</tr>
<tr>
<td>2. Query execution: The database server inserts all the data items, as well as the ICs and serials into the ICDB database.</td>
</tr>
</tbody>
</table>

We evaluate the performance using an INSERT query on the Employees.salaries table. The results are shown in three figures (and three tables), one for each algorithm (HMAC-SHA, CMAC-AES, and RSA). Each figure shows two data points: a) the query conversion time: to generate the ICs and serials for all the data to be inserted and b) the query execution time: to insert all the data and their ICs and serials into the database.

Figures 5.7, 5.8 and 5.9 show that each data point roughly scales linearly with the amount of data inserted. The only difference is the scaling factor for each data point. Note that in the SELECT query the query conversion time is minor and can be ignored. However, the query conversion process in the INSERT query contributes a major part of the overall performance penalty because query conversion needs to generate ICs (an expensive operation) for those data to be inserted.

The average Performance Penalty Rate for an INSERT query using CMAC-AES is the lowest among all three algorithms with 2.68 and 4.94 in OCT and OCF.
respectively. The penalty rate is slightly higher for HMAC-SHA with 2.69 and 5.14 on average in OCT and OCF respectively. While for RSA, the penalty rate is the highest with 6.62 and 21.01 on average in OCT and OCF respectively.

Also, note in Figures 5.7, 5.8 and 5.9 (and Tables 5.7, 5.8 and 5.9) that conversion and execution time for HMAC-SHA and CMAC-AES are roughly equal. While for RSA, conversion time on average is 2.45 times and 3.27 times greater than execution time in OCT and OCF respectively.
Figure 5.7: Using HMAC-SHA, plotted query process time in milliseconds for different number of tuples inserted (in thousands) by the INSERT query into the Employees.salaries table.

Table 5.7: Using HMAC-SHA, query process time raw data in milliseconds for different number of tuples inserted (in thousands) by the INSERT query into the Employees.salaries table.

<table>
<thead>
<tr>
<th>No. of Tuples</th>
<th>Scheme</th>
<th>Conversion</th>
<th>Execution</th>
<th>Total Query Process</th>
<th>Performance Penalty Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>50k</td>
<td>DB</td>
<td>0</td>
<td>851.2</td>
<td>851.2</td>
<td>2.313909974</td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>813.8</td>
<td>1155.8</td>
<td>1969.6</td>
<td>4.411419173</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>1543.4</td>
<td>2211.6</td>
<td>3755</td>
<td>4.411419173</td>
</tr>
<tr>
<td>100k</td>
<td>DB</td>
<td>0</td>
<td>1579.8</td>
<td>1579.8</td>
<td>2.773009242</td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>2061.2</td>
<td>2319.6</td>
<td>4380.8</td>
<td>2.773009242</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>3584.2</td>
<td>4755.6</td>
<td>8339.8</td>
<td>5.279022061</td>
</tr>
<tr>
<td>150k</td>
<td>DB</td>
<td>0</td>
<td>2305.6</td>
<td>2305.6</td>
<td>2.867539903</td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>3032</td>
<td>3579.4</td>
<td>6611.4</td>
<td>2.867539903</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>5757.6</td>
<td>6231.8</td>
<td>11989.4</td>
<td>5.200121443</td>
</tr>
<tr>
<td>200k</td>
<td>DB</td>
<td>0</td>
<td>2966.2</td>
<td>2966.2</td>
<td>2.792461736</td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>3652.2</td>
<td>4630.8</td>
<td>8283</td>
<td>2.792461736</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>8080.6</td>
<td>8746.2</td>
<td>16826.8</td>
<td>5.672847414</td>
</tr>
</tbody>
</table>
**Figure 5.8:** Using CMAC-AES, plotted query process time in milliseconds for different number of tuples inserted (in thousands) by the INSERT query into the Employees.salaries table.

**Table 5.8:** Using CMAC-AES, query process time raw data in milliseconds for different number of tuples inserted (in thousands) by the INSERT query into the Employees.salaries table.

<table>
<thead>
<tr>
<th>No. of Tuples</th>
<th>Scheme</th>
<th>Conversion</th>
<th>Execution</th>
<th>Total Query Process</th>
<th>Performance Penalty Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>50k</td>
<td>DB</td>
<td>0</td>
<td>851.2</td>
<td>851.2</td>
<td>2.338580827</td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>832</td>
<td>1158.6</td>
<td>1990.6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>1389.8</td>
<td>2267.2</td>
<td>3657</td>
<td>4.296287594</td>
</tr>
<tr>
<td>100k</td>
<td>DB</td>
<td>0</td>
<td>1579.8</td>
<td>1579.8</td>
<td>2.976959109</td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>2080.2</td>
<td>2622.8</td>
<td>4703</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>3464.6</td>
<td>5595.2</td>
<td>9059.8</td>
<td>5.734776554</td>
</tr>
<tr>
<td>150k</td>
<td>DB</td>
<td>0</td>
<td>2305.6</td>
<td>2305.6</td>
<td>2.730034004</td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>2910.6</td>
<td>3397.6</td>
<td>6308.2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>5091.4</td>
<td>7262</td>
<td>12353.4</td>
<td>5.357997918</td>
</tr>
<tr>
<td>200k</td>
<td>DB</td>
<td>0</td>
<td>2966.2</td>
<td>2966.2</td>
<td>2.657474209</td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>3566.6</td>
<td>4316</td>
<td>7882.6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>7097.8</td>
<td>8825</td>
<td>15922.8</td>
<td>5.368080372</td>
</tr>
</tbody>
</table>
Figure 5.9: Using RSA, plotted query process time in milliseconds for different number of tuples inserted (in thousands) by the INSERT query into the Employees.salaries table.

Table 5.9: Using RSA, query process time raw data in milliseconds for different number of tuples inserted (in thousands) by the INSERT query into the Employees.salaries table.

<table>
<thead>
<tr>
<th>No. of Tuples</th>
<th>Scheme</th>
<th>Conversion (ms)</th>
<th>Execution (ms)</th>
<th>Total Query Process (ms)</th>
<th>Performance Penalty Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>50k</td>
<td>DB</td>
<td>0</td>
<td>851.2</td>
<td>851.2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>3393.4</td>
<td>1513.4</td>
<td>4906.8</td>
<td>5.764567669</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>12355.6</td>
<td>3521.8</td>
<td>15877.4</td>
<td>18.65296053</td>
</tr>
<tr>
<td>100k</td>
<td>DB</td>
<td>0</td>
<td>1579.8</td>
<td>1579.8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>7392.8</td>
<td>3917.6</td>
<td>11310.4</td>
<td>7.159387264</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>24606</td>
<td>7858</td>
<td>32464</td>
<td>20.54943664</td>
</tr>
<tr>
<td>150k</td>
<td>DB</td>
<td>0</td>
<td>2305.6</td>
<td>2305.6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>10597</td>
<td>4086.4</td>
<td>14683.4</td>
<td>6.368580847</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>37686.2</td>
<td>11234.8</td>
<td>48921</td>
<td>21.21833796</td>
</tr>
<tr>
<td>200k</td>
<td>DB</td>
<td>0</td>
<td>2966.2</td>
<td>2966.2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>15200.4</td>
<td>4922.4</td>
<td>20122.8</td>
<td>6.784033443</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>48782.4</td>
<td>15863</td>
<td>64645.4</td>
<td>21.79401254</td>
</tr>
</tbody>
</table>
DELETE

An ICDB DELETE query needs to fetch the data to be deleted back to the ICDB client to ensure these data are valid and their serial numbers are recorded in the local ICRL file. Thus, an ICDB DELETE query contains the following steps:

<table>
<thead>
<tr>
<th>ICDB DELETE query process steps in the Basic model:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Data fetch: The ICDB client needs to issue a SELECT query to fetch the data to be deleted from the database server.</td>
</tr>
<tr>
<td>2. Verification: The ICDB client verifies the fetched data.</td>
</tr>
<tr>
<td>3. Query execution: If all fetched data is verified, the ICDB client will issue the DELETE query and the cloud database server will execute the DELETE.</td>
</tr>
<tr>
<td>4. ICRL update: Revoke the serial numbers in the ICRL after the DELETE operation is done.</td>
</tr>
</tbody>
</table>

We also analyzed the benchmarks of the time it takes to delete the data and their corresponding integrity codes, using a DELETE query on the Employees.salaries table. The DELETE operation must verify the data to be deleted, before executing the query. Each figure shows three data points a) Query process time (only query execution) for a standard database, b) Query process time of the corresponding SELECT query plus Query execution time for an ICDB OCT counterpart, and c) Query process time of the corresponding SELECT query plus Query execution time for an ICDB OCF counterpart. The X-axis in the figure shows the number of tuples deleted in thousands and the Y-axis shows the time in milliseconds. The time required
to update the ICRL file is insignificant compared to the required time for other steps of DELETE query process.

Figures 5.10, 5.11 and 5.12 (and Tables 5.10, 5.11 and 5.12) show that each data point roughly scales linearly with the amount of data deleted. The only difference is the scaling factor for each data point. Also, not that DELETE query execution takes much larger time than other SQL queries (such as SELECT, INSERT) for the standard database itself. This is because a DELETE query on a table requires checking each referential integrity constraint in which the table is a parent. Also, for a large number of tuples being deleted, MySQL needs to maintain a large transaction log which is also time consuming.

For HMAC-SHA and CMAC-AES, DELETE query execution time takes most of the total query process time, but it is not the case for RSA schemes. For RSA-OCT the verification time is almost equal to the DELETE query execution time, whereas for RSA-OCF the verification time takes the majority of the total query process time. This is because the verification process requires regenerating ICs for all the data to be deleted, in which RSA will take more time to regenerate ICs than the MAC algorithm does.

The average performance penalty rate for DELETE query using CMAC-AES is the lowest among all three algorithms with 1.53 in OCT and 1.25 in OCF. The average penalty rate is slightly higher for HMAC-SHA with 1.62 in OCT and 1.25 in OCF. While for RSA, the average penalty rate is the highest with 2.39 in OCT and 5.75 in OCF.
**Figure 5.10:** Using HMAC-SHA, plotted query process time in milliseconds for different number of tuples deleted (in thousands) by the DELETE query from the Employees.salaries table.

**Table 5.10:** Using HMAC-SHA, query process time raw data in milliseconds for different number of tuples deleted (in thousands) by the DELETE query from the Employees.salaries table.

<table>
<thead>
<tr>
<th>No. of Tuples</th>
<th>Scheme</th>
<th>Fetch</th>
<th>Verification</th>
<th>ICRL Revoke</th>
<th>Execution</th>
<th>Total</th>
<th>Performance Penalty Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>50k</td>
<td>DB</td>
<td></td>
<td></td>
<td></td>
<td>2573.4</td>
<td>2573.4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>135.2</td>
<td>247</td>
<td>36.6</td>
<td>2766</td>
<td>3184.8</td>
<td>1.237584519</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>189.8</td>
<td>677.6</td>
<td>140.8</td>
<td>3293.4</td>
<td>4301.6</td>
<td>1.671562913</td>
</tr>
<tr>
<td>100k</td>
<td>DB</td>
<td></td>
<td></td>
<td></td>
<td>4805</td>
<td>4805</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>301.8</td>
<td>515.8</td>
<td>58.4</td>
<td>5292.8</td>
<td>6168.8</td>
<td>1.283829344</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>414.2</td>
<td>1332.4</td>
<td>308.8</td>
<td>5862.8</td>
<td>7918.2</td>
<td>1.647908429</td>
</tr>
<tr>
<td>150k</td>
<td>DB</td>
<td></td>
<td></td>
<td></td>
<td>7124</td>
<td>7124</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>406.8</td>
<td>691.6</td>
<td>79.6</td>
<td>7421.6</td>
<td>8599.6</td>
<td>1.207130825</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>585.6</td>
<td>1813.6</td>
<td>407.4</td>
<td>8516.8</td>
<td>11323.4</td>
<td>1.589472207</td>
</tr>
<tr>
<td>200k</td>
<td>DB</td>
<td></td>
<td></td>
<td></td>
<td>9457.8</td>
<td>9457.8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>525.6</td>
<td>814</td>
<td>104.8</td>
<td>10001.6</td>
<td>11446</td>
<td>1.210218021</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>739.4</td>
<td>2531</td>
<td>515.8</td>
<td>11056.2</td>
<td>14842.4</td>
<td>1.569329019</td>
</tr>
</tbody>
</table>
Figure 5.11: Using CMAC-AES, plotted query process time in milliseconds for different number of tuples deleted (in thousands) by the DELETE query from the Employees.salaries table.

Table 5.11: Using CMAC-AES, query process time raw data in milliseconds for different number of tuples deleted (in thousands) by the DELETE query from the Employees.salaries table.
**Figure 5.12:** Using RSA, plotted query process time in milliseconds for different number of tuples deleted (in thousands) by the DELETE query from the `Employees.salaries` table.

**Table 5.12:** Using RSA, query process time raw data in milliseconds for different number of tuples deleted (in thousands) by the DELETE query from the `Employees.salaries` table.

<table>
<thead>
<tr>
<th>No. of Tuples</th>
<th>Scheme</th>
<th>Fetch</th>
<th>Verification</th>
<th>ICRL Revoke</th>
<th>Execution</th>
<th>Total</th>
<th>Performance Penalty Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>50k</td>
<td>DB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>130.8</td>
<td>2573.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>234.4</td>
<td>3378.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100k</td>
<td>DB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>296.6</td>
<td>5111.8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>450</td>
<td>5640.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>150k</td>
<td>DB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>407.6</td>
<td>8136.8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>640.8</td>
<td>8218.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>200k</td>
<td>DB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>547.4</td>
<td>10794</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>882.4</td>
<td>11136.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
UPDATE

The UPDATE operation mirrors both DELETE and INSERT as it is functionally equivalent to first DELETE a data item (or a set of data items) and then INSERT a new data item (or a new set of data items). The performance penalty for an ICDB UPDATE query can be analyzed based on the performance penalty presented earlier in both ICDB DELETE and ICDB INSERT.
JOIN

To perform a JOIN query in ICDB requires the following steps:

ICDB JOIN query process steps in the Basic model:

1. Query Conversion: The ICDB client converts an SQL JOIN query to an ICDB JOIN query (See Algorithms A and C in Chapter 4) and then issues the query to the database server.

2. Query Execution (Data fetch): The database server executes the query to fetch and return data to the ICDB client.

3. Verification: The ICDB client verifies the fetched data.

In our experiment for the JOIN query, attributes from Employees.employees and Employees.salaries tables were joined together. The JOIN query used for the experiment was:

```
SELECT * FROM employees, salaries WHERE employees.emp_no = salaries.emp_no;
```

Figure 5.13, 5.14, 5.15 and Tables 5.13, 5.14, 5.15 show the time to process an ICDB JOIN query in the Basic model. The verification process in the JOIN query contributes more to the overall performance penalty than the query execution (data fetching) in both OCT and OCF because verification needs to regenerate ICs. Again, the verification time in OCF is greater than that in OCT because there are more ICs to regenerate in OCF than in OCT.

The average Performance Penalty Rate for the ICDB JOIN query using CMAC-AES is the lowest among all three algorithms with 12.18 in OCT and 31.73 in OCF. The average penalty rate is slightly higher for HMAC-SHA with 13.56 in OCT and
36.36 in OCF. For RSA, the average penalty rate is the highest with 89.05 in OCT and 407.41 in OCF.
Figure 5.13: HMAC-SHA plotted for a Query joining the Employees.employees and the Employees.salaries tables, showing the time required (in milliseconds) for the number of tuples selected (in thousands).

Table 5.13: HMAC-SHA raw data for a Query joining the Employees.employees and the Employees.salaries tables, showing the time required (in milliseconds) for the number of tuples selected (in thousands).

<table>
<thead>
<tr>
<th>No. of Tuples</th>
<th>Scheme</th>
<th>Fetch (ms)</th>
<th>Verification (ms)</th>
<th>Total Query Process (ms)</th>
<th>Performance Penalty Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>50k</td>
<td>DB</td>
<td>65</td>
<td>65</td>
<td>879.68</td>
<td>13.53353846</td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>221.68</td>
<td>658</td>
<td>879.68</td>
<td>13.54324324</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>342.38</td>
<td>1688.4</td>
<td>2030.78</td>
<td>31.24276923</td>
</tr>
<tr>
<td>100k</td>
<td>DB</td>
<td>124</td>
<td>124</td>
<td>2406.14</td>
<td>31.24276923</td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>602.14</td>
<td>1078</td>
<td>2406.14</td>
<td>31.24276923</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>726.92</td>
<td>3754</td>
<td>4480.92</td>
<td>36.13645161</td>
</tr>
<tr>
<td>150k</td>
<td>DB</td>
<td>177</td>
<td>177</td>
<td>3051.42</td>
<td>37.1839548</td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>752.42</td>
<td>1653.8</td>
<td>2406.22</td>
<td>37.1839548</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>946.56</td>
<td>5635</td>
<td>6581.56</td>
<td>40.87048889</td>
</tr>
<tr>
<td>200k</td>
<td>DB</td>
<td>225</td>
<td>225</td>
<td>3051.42</td>
<td>37.1839548</td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>902.02</td>
<td>2152</td>
<td>3051.42</td>
<td>37.1839548</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>1380.06</td>
<td>7815.8</td>
<td>9195.86</td>
<td>40.87048889</td>
</tr>
</tbody>
</table>
Figure 5.14: CMAC-AES plotted for a Query joining the Employees.employees and the Employees.salaries tables, showing the time required (in milliseconds) for the number of tuples selected (in thousands).

Table 5.14: CMAC-AES raw data for a Query joining the Employees.employees and the Employees.salaries tables, showing the time required (in milliseconds) for the number of tuples selected (in thousands).

<table>
<thead>
<tr>
<th>No. of Tuples</th>
<th>Scheme</th>
<th>Fetch (ms)</th>
<th>Verification (ms)</th>
<th>Total Query Process (ms)</th>
<th>Performance Penalty Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>50k</td>
<td>DB</td>
<td>65</td>
<td>65</td>
<td>738.5</td>
<td>11.36153846</td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>239.7</td>
<td>498.8</td>
<td>738.5</td>
<td>11.36153846</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>381.48</td>
<td>1666.2</td>
<td>2047.68</td>
<td>31.50276923</td>
</tr>
<tr>
<td>100k</td>
<td>DB</td>
<td>124</td>
<td>124</td>
<td>1405.04</td>
<td>11.33096774</td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>488.24</td>
<td>916.8</td>
<td>1405.04</td>
<td>11.33096774</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>704.14</td>
<td>3019.4</td>
<td>3723.54</td>
<td>30.02854839</td>
</tr>
<tr>
<td>150k</td>
<td>DB</td>
<td>177</td>
<td>177</td>
<td>2330.86</td>
<td>13.16870056</td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>679.66</td>
<td>1651.2</td>
<td>2330.86</td>
<td>13.16870056</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>1020.34</td>
<td>5318.6</td>
<td>6338.94</td>
<td>35.81322034</td>
</tr>
<tr>
<td>200k</td>
<td>DB</td>
<td>225</td>
<td>225</td>
<td>2893.36</td>
<td>12.85937778</td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>919.36</td>
<td>1974</td>
<td>2893.36</td>
<td>12.85937778</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>1274.66</td>
<td>7382</td>
<td>8656.66</td>
<td>38.47404444</td>
</tr>
</tbody>
</table>
Figure 5.15: RSA plotted for a Query joining the Employees.employees and the Employees.salaries tables, showing the time required (in milliseconds) for the number of tuples selected (in thousands).

Table 5.15: RSA raw data for a Query joining the Employees.employees and the Employees.salaries tables, showing the time required (in milliseconds) for the number of tuples selected (in thousands).

<table>
<thead>
<tr>
<th>No. of Tuples</th>
<th>Scheme</th>
<th>Fetch</th>
<th>Verification</th>
<th>Total Query Process</th>
<th>Performance Penalty Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>50k</td>
<td>DB</td>
<td>65</td>
<td></td>
<td>65</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>260.1</td>
<td>5199.8</td>
<td>5459.9</td>
<td>83.99846154</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>378.42</td>
<td>25458.4</td>
<td>25836.82</td>
<td>397.4895385</td>
</tr>
<tr>
<td>100k</td>
<td>DB</td>
<td>124</td>
<td></td>
<td>124</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>516.46</td>
<td>10113</td>
<td>10629.46</td>
<td>85.72145161</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>808.52</td>
<td>47549.4</td>
<td>48357.92</td>
<td>389.9832258</td>
</tr>
<tr>
<td>150k</td>
<td>DB</td>
<td>177</td>
<td></td>
<td>177</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>710.6</td>
<td>15725.6</td>
<td>16436.2</td>
<td>92.85988701</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>1098.88</td>
<td>71085.6</td>
<td>72184.48</td>
<td>407.8219209</td>
</tr>
<tr>
<td>200k</td>
<td>DB</td>
<td>225</td>
<td></td>
<td>225</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>971.04</td>
<td>20099.8</td>
<td>21070.84</td>
<td>93.64817778</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>2082.16</td>
<td>95642</td>
<td>97724.16</td>
<td>434.3296</td>
</tr>
</tbody>
</table>
**Functional Query**

To perform a functional query, all the data items used to evaluate the aggregate value in the functional query must be fetched and verified. Thus, an ICDB functional query will need to perform the following steps:

<table>
<thead>
<tr>
<th>ICDB Function query process steps in the Basic model:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Data fetch: The ICDB client will need to issue a SELECT query to fetch the required data for evaluating the aggregate operation.</td>
</tr>
<tr>
<td>2. Verification: The ICDB verifies the fetched data.</td>
</tr>
<tr>
<td>3. Query execution in local: The ICDB client now is able to Compute the aggregate value based on the fetched data. The aggregate function could be <strong>SUM</strong>, <strong>MIN</strong>, <strong>MAX</strong>, <strong>AVG</strong>, or <strong>COUNT</strong>.</td>
</tr>
</tbody>
</table>

Figures 5.16, 5.17, 5.18 and Tables 5.16, 5.17, 5.18 show the experimental results for an ICDB Functional query. The Functional query used for this experiment includes four aggregate operations (SUM, MIN, MAX, AVG) in a single Functional query. The data shows that the process time for a functional query is similar to the corresponding SELECT query process time. The only additional time required is the time for the ICDB client to locally compute the aggregate values over the verified data from the SELECT query, which is negligible. It is also noted that for OCF, the functional query take less time than the SELECT query in our previous experiment. This is because the corresponding SELECT query for the Functional query must fetch only the `Employees.salaries` and its related attributes for verification. For OCT, all the attributes in the tuple must be fetched.
The average performance penalty rate for Functional query using CMAC-AES is the lowest among all three algorithms with 30.09 in OCT and 47.82 in OCF. The average penalty rate is slightly higher for HMAC-SHA with 36.05 in OCT and 53.46 in OCF. While for RSA, the average penalty rate is the highest with 197.38 in OCT and 534.33 in OCF.
Figure 5.16: HMAC-SHA plotted with time required (in milliseconds) against number of tuples (in thousands) returned using an ICDB SELECT Query corresponding to the Functional Query

Table 5.16: HMAC-SHA plotted with time required (in milliseconds) against number of tuples (in thousands) returned using an ICDB SELECT Query corresponding to the Functional Query

<table>
<thead>
<tr>
<th>No. of Tuples</th>
<th>Scheme</th>
<th>Time (ms)</th>
<th>Total Query Process</th>
<th>Performance Penalty Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Fetch</td>
<td>Verification</td>
<td></td>
</tr>
<tr>
<td>50k</td>
<td>DB</td>
<td>17.4</td>
<td>17.4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>130.4</td>
<td>435.2</td>
<td>565.6</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>140.98</td>
<td>608.8</td>
<td>749.78</td>
</tr>
<tr>
<td>100k</td>
<td>DB</td>
<td>30.1</td>
<td>30.1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>354.2</td>
<td>809.6</td>
<td>1163.8</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>299.32</td>
<td>1357</td>
<td>1656.32</td>
</tr>
<tr>
<td>150k</td>
<td>DB</td>
<td>39.6</td>
<td>39.6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>442.6</td>
<td>1025.6</td>
<td>1468.2</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>389.76</td>
<td>1824.6</td>
<td>2214.36</td>
</tr>
<tr>
<td>200k</td>
<td>DB</td>
<td>53.9</td>
<td>53.9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>530.6</td>
<td>1407.2</td>
<td>1937.8</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>568.26</td>
<td>2655.6</td>
<td>3223.86</td>
</tr>
</tbody>
</table>
Figure 5.17: CMAC-AES plotted with time required (in milliseconds) against number of tuples (in thousands) returned using an ICDB SELECT Query corresponding to the Functional Query

Table 5.17: CMAC-AES plotted with time required (in milliseconds) against number of tuples (in thousands) returned using an ICDB SELECT Query corresponding to the Functional Query
**Figure 5.18:** RSA plotted with time required (in milliseconds) against number of tuples (in thousands) returned using an ICDB SELECT Query corresponding to the Functional Query

**Table 5.18:** RSA plotted with time required (in milliseconds) against number of tuples (in thousands) returned using an ICDB SELECT Query corresponding to the Functional Query

<table>
<thead>
<tr>
<th>No. of Tuples</th>
<th>Scheme</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fetch</td>
<td>Verification</td>
</tr>
<tr>
<td>50k</td>
<td>DB</td>
<td>17.4</td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>153</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>155.82</td>
</tr>
<tr>
<td>100k</td>
<td>DB</td>
<td>30.1</td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>303.8</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>332.92</td>
</tr>
<tr>
<td>150k</td>
<td>DB</td>
<td>39.6</td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>418</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>452.48</td>
</tr>
<tr>
<td>200k</td>
<td>DB</td>
<td>53.9</td>
</tr>
</tbody>
</table>
5.4.2 Experimental Results for the Dual Mode Verification (DMV) Model

In this section, we will present and analyze our experimental results for the ICDB DMV model as described in Section 3.2. In order to minimize network variability for our experiments, all three entities in the DMV model: cloud database server, ICDB cloud application and ICDB client, were run on the same machine (see Appendix B for machine specifications). However, in the real world, these three entities should be hosted by different machines in different sites. The purpose of having the DMV model, rather than the Basic model, is to reduce the network loads (downloading/uploading ICs between the ICDB client and the cloud database server).

DMV model includes two different verification modes: aggregate verification mode and an optional detailed verification mode. For an ICDB query process, DV mode does not involve the ICDB cloud application, whereas AV mode involves all three entities. The figures in this section include only the experimental results for AV mode because all the processes (and thus the performance) for the DV mode would be the same as those in the basic ICDB model presented in the previous section 5.4.1.

**SELECT**

In a standard SQL database, a SELECT query only contains one step - the execution of the query in the database server, i.e., to fetch data. However, in the AV mode of the DMV model, an ICDB SELECT query has the following steps as described below.

<table>
<thead>
<tr>
<th>ICDB SELECT query process steps in the AV mode of DMV model:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Query conversion: ICDB client converts an SQL SELECT query to two</td>
</tr>
</tbody>
</table>
ICDB SELECT queries $Q_1$ and $Q_2$ as described in Algorithms B and D in Chapter 4, and then issues the queries $Q_1$ and $Q_2$ to the cloud database server and the ICDB cloud application, respectively.

2. Query $Q_2$ processing has two major steps.

   (a) IC fetch: The ICDB cloud application forwards query $Q_2$ to the cloud database server to retrieve all the corresponding ICs.

   (b) AIC generation: The cloud application generates an Aggregate Integrity Code (AIC) and then returns it to the ICDB client.

3. Query $Q_1$ processing has two major steps.

   (a) Data fetch: Query $Q_1$ fetches data and serial numbers (but not the corresponding ICs) from the ICDB database and returns them to the ICDB client.

   (b) Verification: Upon receiving data and serial numbers through $Q_1$ and an AIC through $Q_2$, the ICDB client verifies the data integrity by regenerating an AIC from the fetched $Q_1$ result and then match it with the AIC returned by $Q_2$.

This section presents the performance penalty of the ICDB SELECT query, using a SELECT * query on the Employees.salaries table in the aggregate verification mode of the DMV model described in section 3.2. The experimental results are shown in three figures and three tables, one for each algorithm (HMAC-SHA, CMAC-AES, and RSA). Each figure shows three data points a) Query process time (only query execution) for a standard database, b) Query process time for the ICDB OCT counterpart, and c) Query process time for the ICDB OCF counterpart. Although
query conversion is a part of the query process in the ICDB, it is not shown in the figures due to its negligible duration.

The ICDB client sends the ICDB query $Q_1$ to the cloud database server to fetch the data and their serials and sends the ICDB query $Q_2$ to the cloud application requesting the AIC. Since the requests are asynchronous, the query process time with the longest of the two queries could be considered as the best case. However, we benchmarked and reported the total processing time as the summation of both $Q_1$’s and $Q_2$’s processing time. In Figures 5.21, 5.20 and 5.19, the reported time for IC Fetch and AIC generation are the processing time for $Q_2$, while the reported time for Data Fetch and verification are the processing time for $Q_1$.

Observing the experimental results from Figures 5.21, 5.20 and 5.19 (and Tables 5.21, 5.20 and 5.19), we list some notable results below.

1. AIC generation time (by cloud application) is relatively less than the verification time (by ICDB client) for both CMAC-AES and HMAC-SHA. This is due to the fact that the ICDB client has to regenerate all the ICs for the data fetched in the verification step. While the cloud application just has to fetch all ICs from the cloud database server to generate the AIC.

2. In the best case scenario when both $Q_1$ and $Q_2$ are processed simultaneously, one need only consider the longer time taken by $Q_1$ or $Q_2$ while analyzing the performance penalty. In this scenario, RSA performance is quite similar to CMAC-AES and HMAC-SHA in both OCT and OCF. Note that RSA performance is much worse compared to the CMAC-AES and HMAC-AES in the Basic model.
3. In the worst case scenario when considering the summation of the time taken by both $Q_1$ and $Q_2$, CMAC-AES trivially outperforms both RSA and HMAC-SHA.

4. The average performance penalty rate for the SELECT query using CMAC-AES is the lowest among the three algorithms with 18.2 in OCT and 43.97 in OCF. The average penalty rate is slightly higher for HMAC-SHA with 18.8 in OCT and 48.96 in OCF. While for RSA, the average penalty rate is greatest with 27.33 in OCT and 74.78 in OCF.
Figure 5.19: Using HMAC-SHA, plotted query process time in AV mode in milliseconds for different number of tuples returned (in thousands) by the SELECT * query from the Employees.salaries table.

Table 5.19: Using HMAC-SHA, query process time raw data in AV mode in milliseconds for different number of tuples returned (in thousands) by the SELECT * query from the Employees.salaries table.

<table>
<thead>
<tr>
<th>No. of Tuples</th>
<th>Scheme</th>
<th>IC Fetch</th>
<th>AIC generation</th>
<th>Data Fetch</th>
<th>Verification</th>
<th>Total Query Process</th>
<th>Performance Penalty Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>50k</td>
<td>DB</td>
<td>34.8</td>
<td>34.8</td>
<td>369.6886</td>
<td>634.6886</td>
<td>18.23817816</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>67.8</td>
<td>97.4</td>
<td>159.8</td>
<td>309.6886</td>
<td>634.6886</td>
<td>18.23817816</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>154.2</td>
<td>162.4</td>
<td>179.8</td>
<td>1203.5484</td>
<td>1609.9484</td>
<td>48.84909195</td>
</tr>
<tr>
<td>100k</td>
<td>DB</td>
<td>60.2</td>
<td>60.2</td>
<td>263.6</td>
<td>438.4222</td>
<td>698.0222</td>
<td>17.8744518</td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>137.6</td>
<td>137.6</td>
<td>253.6</td>
<td>587.6222</td>
<td>841.2222</td>
<td>17.8744518</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>298.2</td>
<td>299.6</td>
<td>379.6</td>
<td>1858.7064</td>
<td>2237.3064</td>
<td>44.87542452</td>
</tr>
<tr>
<td>150k</td>
<td>DB</td>
<td>79.2</td>
<td>79.2</td>
<td>365.6</td>
<td>873.9332</td>
<td>1243.9332</td>
<td>20.11356961</td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>217.4</td>
<td>217.4</td>
<td>365.6</td>
<td>873.9332</td>
<td>1243.9332</td>
<td>20.11356961</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>621.6</td>
<td>621.6</td>
<td>571.6</td>
<td>2729.4276</td>
<td>3301.0276</td>
<td>52.42864141</td>
</tr>
<tr>
<td>200k</td>
<td>DB</td>
<td>107.8</td>
<td>107.8</td>
<td>497.8</td>
<td>1688.207</td>
<td>2186.007</td>
<td>19.1787254</td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>293.8</td>
<td>293.8</td>
<td>497.8</td>
<td>1688.207</td>
<td>2186.007</td>
<td>19.1787254</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>660.6</td>
<td>660.6</td>
<td>636.2</td>
<td>3753.4922</td>
<td>4415.0922</td>
<td>49.67617996</td>
</tr>
</tbody>
</table>
**Figure 5.20:** Using CMAC-AES, plotted query process time in AV mode in milliseconds for different number of tuples returned (in thousands) by the SELECT * query from the Employees.salaries table.

**Table 5.20:** Using CMAC-AES, query process time raw data in AV mode in milliseconds for different number of tuples returned (in thousands) by the SELECT * query from the Employees.salaries table.
**Figure 5.21:** Using RSA, plotted query process time in AV mode in milliseconds for different number of tuples returned (in thousands) by the SELECT * query from the Employees.salaries table.

**Table 5.21:** Using RSA, query process time raw data in AV mode in milliseconds for different number of tuples returned (in thousands) by the SELECT * query from the Employees.salaries table.

<table>
<thead>
<tr>
<th>No. of Tuples</th>
<th>Scheme</th>
<th>IC Fetch (Q2)</th>
<th>AIC generation (Q2)</th>
<th>Data Fetch (Q1)</th>
<th>Verification (Q1)</th>
<th>Total Query Process</th>
<th>Performance Penalty Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>50k</td>
<td>OCT</td>
<td>59.6</td>
<td>297.8</td>
<td>147.2</td>
<td>445.7996</td>
<td>950.3996</td>
<td>27.31033333</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>108.8</td>
<td>1010</td>
<td>37.3</td>
<td>880.0096</td>
<td>2371.8096</td>
<td>68.15544828</td>
</tr>
<tr>
<td>100k</td>
<td>OCT</td>
<td>90.2</td>
<td>689.6</td>
<td>218.6</td>
<td>637.822</td>
<td>1596.222</td>
<td>25.51863757</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>224.8</td>
<td>2091.2</td>
<td>464.6</td>
<td>1572.294</td>
<td>4352.8774</td>
<td>72.19093555</td>
</tr>
<tr>
<td>150k</td>
<td>OCT</td>
<td>144.8</td>
<td>893.2</td>
<td>380</td>
<td>865.4458</td>
<td>2283.4458</td>
<td>28.83136836</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>248.8</td>
<td>3217.2</td>
<td>651.8</td>
<td>2899.6334</td>
<td>6517.6334</td>
<td>82.29082576</td>
</tr>
<tr>
<td>200k</td>
<td>OCT</td>
<td>271.4</td>
<td>1139.8</td>
<td>494</td>
<td>1676.194</td>
<td>2980.394</td>
<td>27.64713997</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>435.8</td>
<td>4119.2</td>
<td>681.2</td>
<td>2995.6335</td>
<td>8231.6335</td>
<td>78.36204862</td>
</tr>
</tbody>
</table>
As was shown when analyzing the ICDB Basic model, a different interpretation can be done for the results in Figures 5.21, 5.20 and 5.19 (and Tables 5.21, 5.20 and 5.19). The process rate is derived based on the same Equation 5.3 used in ICDB Basic model. However, the Total Processing Time in this case (AV mode in the DMV model) includes IC fetch time, AIC generation time, data fetch time and also integrity Verification time.

Figure 5.22 and Table 5.22 show the process rate for HMAC-SHA is 2.72 MB/sec in OCT and 1.05 MB/sec in OCF. CMAC-AES has similar but slightly higher process rates, with 2.83 MB/sec in OCT and 1.16 MB/sec in OCF. RSA takes longer to process the user data with a process rate of 1.88 MB/sec in OCT and 0.69 MB/sec in OCF. Similar to the ICDB Basic model, CMAC-AES has the highest (best) process rate amongst the three different algorithms.

Compared to the ICDB Basic model, the process rates for both CMAC-AES and HMAC-SHA are decreased (slower) than those in the Basic model (See Figure 5.6 and Table 5.6). This result is expected since in the AV mode of the DMV model, there are more query process steps than the basic model has. However, the process rate for the RSA algorithm in the DMV model, compared to the ICDB Basic model, is improved significantly from 0.501 MB/sec to 1.88 MB/sec in OCT and from 0.134 MB/sec to 0.69 MB/sec in OCF. This result is also expected. Though there are more query process steps in the DMV model, the verification step in the ICDB client does not require the time-consuming recalculation of all the ICs before computing the AIC. The ICDB client is able to compute the AIC from data directly due to the RSA’s multiplication homomorphic property.
Figure 5.22: A chart plotted the process rates for the AV mode in DMV model showing the speed the user data (size in MB) can be processed by three different algorithms in OCT and in OCF. The leftmost bar labeled DB is the process rate for a standard SQL database.

Table 5.22: A table with process rate raw data for the AV mode in DMV model showing the speed the user data (size in MB) can be processed by three different algorithms in OCT and in OCF. The data in the row labeled DB is the process rate for a standard SQL database.

<table>
<thead>
<tr>
<th></th>
<th>OCT</th>
<th>OCF</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB</td>
<td>51.41</td>
<td></td>
</tr>
<tr>
<td>HMAC-SHA</td>
<td>2.72</td>
<td>1.05</td>
</tr>
<tr>
<td>CMAC-AES</td>
<td>2.83</td>
<td>1.16</td>
</tr>
<tr>
<td>RSA</td>
<td>1.88</td>
<td>0.69</td>
</tr>
</tbody>
</table>
**INSERT**

To perform an INSERT query in AV mode of the DMV model, the same steps are performed as in the INSERT query processing of the basic ICDB model described in Section 5.4.1. Since each data along with its corresponding IC unit must be inserted into the cloud database as in the basic model, the ICDB cloud application plays no role in the INSERT query in the DMV model. Thus, the performance penalty for the INSERT query in the DMV model is the same as that in the Basic model.

**DELETE**

In the AV mode of the DMV model, the ICDB DELETE query must fetch the data to be deleted back to the ICDB client to ensure these data are valid and to record the serial numbers in the local ICRL file. Thus, an ICDB DELETE query takes the following steps:

<table>
<thead>
<tr>
<th>ICDB DELETE query process steps in the AV mode of DMV model:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Query conversion: The ICDB client generates an SQL SELECT query from the DELETE query and converts it into two ICDB SELECT queries $Q_1$ and $Q_2$ as described in Algorithms B and D in Chapter 4, and then issues the queries $Q_1$ and $Q_2$ to the Cloud database server and the ICDB cloud application, respectively.</td>
</tr>
<tr>
<td>2. SELECT queries $Q_1$ and $Q_2$ processing: The processes of $Q_1$ and $Q_2$ are the same as the SELECT query process described earlier in the AV Mode of the DMV model, which includes IC fetch, AIC generation, data fetch, and verification.</td>
</tr>
</tbody>
</table>
3. DELETE query execution: If all data fetched is verified, the ICDB client will issue the DELETE query and the database server will execute the DELETE.

4. ICRL update: Revoke the serial numbers in the ICRL after the DELETE operation is done.

This section presents the performance penalty of the ICDB DELETE query, using a DELETE query on the **Employees.salaries** table in the aggregate verification mode of the DMV model described in section 3.2. The experimental results are shown in three figures and three tables, one for each algorithm (HMAC-SHA, CMAC-AES, and RSA). Each figure shows three data points a) DELETE query process time (only query execution) for a standard database, b) The corresponding SELECT query process time plus DELETE query execution time for an ICDB OCT counterpart, and c) The corresponding SELECT query process time plus DELETE query execution time for an ICDB OCF counterpart. From Figures 5.23, 5.24, 5.25 (and Tables 5.23, 5.24, 5.25), we have noted these implications:

1. The DELETE query execution takes more time than the corresponding ICDB SELECT query process plus ICRL update in all six combinations of our experiment because the DELETE query requires the database server to check the referential integrity constraint for each data to be deleted.

2. Time taken to update the ICRL file after successful deletion is negligible, compared to the total query process time.

3. For HMAC-SHA and CMAC-AES, the result is similar to the basic model where DELETE query execution time is comparatively larger than the corresponding
ICDB SELECT query process time. However, the experimental result for RSA is totally opposite to the result in the basic model. This is because the DELETE query execution time is actually more than the corresponding ICDB SELECT query process time (SELECT query process includes verification) in the AV mode of the DMV model. This opposite result is because the homomorphic property of RSA allows the ICDB client to generate the AIC directly without having to regenerating all the ICs in the aggregate verification mode.

4. The average performance penalty rate for DELETE query using CMAC-AES is 1.29 in OCT and 1.7 in OCF. The average penalty rate for DELETE query using HMAC-SHA is 1.29 in OCT and 1.8 in OCF. For RSA, the average penalty rate is slightly higher with 1.43 in OCT and 2.13 in OCF, but is improved significantly over the performance of the DELETE query in the basic model due to the RSA homomorphic property and because there is no need to regenerate all ICs.
Figure 5.23: Using HMAC-SHA, plotted query process time in AV mode in milliseconds for different number of tuples deleted (in thousands) by the DELETE * query from the Employees.salaries table.

Table 5.23: Using HMAC-SHA, query process time raw data in AV mode in milliseconds for different number of tuples deleted (in thousands) by the DELETE * query from the Employees.salaries table.
Figure 5.24: Using CMAC-AES, plotted query process time in AV mode in milliseconds for different number of tuples deleted (in thousands) by the DELETE * query from the Employees.salaries table.

Table 5.24: Using CMAC-AES, query process time raw data in AV mode in milliseconds for different number of tuples deleted (in thousands) by the DELETE * query from the Employees.salaries table.
Figure 5.25: Using RSA, plotted query process time in AV mode in milliseconds for different number of tuples deleted (in thousands) by the DELETE * query from the Employees.salaries table.

Table 5.25: Using RSA, query process time raw data in AV mode in milliseconds for different number of tuples deleted (in thousands) by the DELETE * query from the Employees.salaries table.
**UPDATE**

Similar to the Basic model, the UPDATE operation in the AV Mode of the DMV model mirrors both DELETE and INSERT since it is functionally equivalent to first DELETE a data item (or a set of data items) and then INSERT a new data item (or a new set of data items). Hence, the experimental result for the UPDATE query process can be analyzed from the DELETE and INSERT query process described earlier.
JOIN

To perform a JOIN query in the AV mode of DMV model requires the following steps:

ICDB JOIN query process steps in the AV mode of the DMV model:

1. Query Conversion: The ICDB client converts an SQL JOIN query into two ICDB JOIN queries, $Q_1$ and $Q_2$ (See Algorithms B and D in Chapter 4) and then issues the query $Q_1$ to the cloud database server and the query $Q_2$ to ICDB cloud application.

2. Query $Q_2$ processing has two major steps: IC fetch and AIC generation in the ICDB cloud application. This process is the same as the $Q_2$ process in the SELECT query described earlier in the AV mode of DMV model.

3. Query $Q_1$ processing has two major steps: Data fetch and Verification in the ICDB client. Again, this process is the same as the $Q_1$ process in the SELECT query described earlier in the AV mode of DMV model.

To analyze a JOIN query in the AV mode of the DMV model, attributes from the Employees.employees and the Employees.salaries tables were joined together. The JOIN query used for the experiment was `SELECT * FROM employees, salaries where employees.emp_no = salaries.emp_no`. Figures 5.26, 5.27, 5.28 and Tables 5.26, 5.27, 5.28 show the experimental results.

The time for AIC generation is greater than the IC fetch time in the cloud application. For RSA, compared to HMAC-SHA and CMAC-AES, the larger time required for AIC generation is because RSA needs to perform modular multiplications over large size ICs (1024 bits) to generate the AIC, whereas MAC algorithms just need to combine the ICs (128 bits) and hash them to generate the AIC. Similarly,
verification time required by the ICDB client is more than the data fetch time. For RSA, compared to HMAC-SHA and CMAC-AES, verification time is less because RSA does not need to regenerate all ICs but generates the AIC directly from the data (because of the homomorphic property) for verification.

The average performance penalty rate for the JOIN query in the AV mode using CMAC-AES is the lowest among all three algorithms with 15.53 in OCT and 45.2 in OCF. The average penalty rate is slightly higher for HMAC-SHA with 18.47 in OCT and 53.18 in OCF. Both MAC algorithms in the AV mode have an increased average penalty rate as compared to the basic model. However for RSA, the average penalty rate is 24.97 in OCT and 81.17 in OCF, which is a huge improvement compared to the basic model with the average penalty rate for RSA being 89.05 in OCT and 407.41 in OCF. Again, the reason for this improvement is because RSA’s homomorphic property allows the ICDB client to generate the AIC directly from the data without the need to regenerate all ICs in the aggregate verification mode.
**Figure 5.26:** HMAC-SHA plotted query processing time for a query joining the `Employees.employees` and the `Employees.salaries` tables, showing the time required (in milliseconds) for the number of tuples selected (in thousands) in aggregate verification mode.

**Table 5.26:** HMAC-SHA plotted query processing time for a query joining the `Employees.employees` and the `Employees.salaries` tables, showing the time required (in milliseconds) for the number of tuples selected (in thousands) in aggregate verification mode.
Figure 5.27: CMAC-AES plotted query processing time for a query joining the Employees.employees and the Employees.salaries tables, showing the time required (in milliseconds) for the number of tuples selected (in thousands) in aggregate verification mode.

Table 5.27: CMAC-AES plotted query processing time for a query joining the Employees.employees and the Employees.salaries tables, showing the time required (in milliseconds) for the number of tuples selected (in thousands) in aggregate verification mode.

<table>
<thead>
<tr>
<th>No. of Tuples</th>
<th>Scheme</th>
<th>Time (ms)</th>
<th>IC Fetch</th>
<th>AIC generation</th>
<th>Data Fetch</th>
<th>Verification</th>
<th>Total Query Process</th>
<th>Performance Penalty Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>50k</td>
<td>DB</td>
<td>65</td>
<td>65</td>
<td>65</td>
<td>534.7792</td>
<td>933.4592</td>
<td>14.3691077</td>
<td>113.441538</td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>97.58</td>
<td>87.9</td>
<td>214.2</td>
<td>535.4592</td>
<td>933.4592</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>145.86</td>
<td>353.7</td>
<td>273.7</td>
<td>2354.212</td>
<td>3127.302</td>
<td>48.11541538</td>
<td></td>
</tr>
<tr>
<td>100k</td>
<td>DB</td>
<td>124</td>
<td>124</td>
<td>124</td>
<td>1119.5232</td>
<td>1911.9632</td>
<td>45.14905866</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>155.04</td>
<td>237.9</td>
<td>399.5</td>
<td>1119.5232</td>
<td>1911.9632</td>
<td>42.15378548</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>292.06</td>
<td>449.8</td>
<td>621.23</td>
<td>3905.8574</td>
<td>5271.9574</td>
<td>44.15378548</td>
<td></td>
</tr>
<tr>
<td>150k</td>
<td>DB</td>
<td>177</td>
<td>177</td>
<td>177</td>
<td>1614.6506</td>
<td>2102.5908</td>
<td>45.09163164</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>222.02</td>
<td>291.8</td>
<td>704.08</td>
<td>1614.6506</td>
<td>2102.5908</td>
<td>39.13508616</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>454.82</td>
<td>743.5</td>
<td>944.34</td>
<td>6014.8834</td>
<td>7920.1435</td>
<td>45.13508616</td>
<td></td>
</tr>
<tr>
<td>200k</td>
<td>DB</td>
<td>225</td>
<td>225</td>
<td>225</td>
<td>2251.0052</td>
<td>3763.2952</td>
<td>45.72535644</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>364.48</td>
<td>245.7</td>
<td>902.02</td>
<td>2251.0052</td>
<td>3763.2952</td>
<td>37.12535644</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>452.2</td>
<td>763</td>
<td>1068.28</td>
<td>7846.443</td>
<td>10129.923</td>
<td>45.02188</td>
<td></td>
</tr>
</tbody>
</table>
Figure 5.28: RSA plotted query processing time for a query joining the Employees.employees and the Employees.salaries tables, showing the time required (in milliseconds) for the number of tuples selected (in thousands) in aggregate verification mode.

Table 5.28: RSA plotted query processing time for a query joining the Employees.employees and the Employees.salaries tables, showing the time required (in milliseconds) for the number of tuples selected (in thousands) in aggregate verification mode.
Functional Query

Similar to the ICDB basic model, to perform a functional query in the AV mode of DMV model, all the data items used to evaluate the aggregate value must be fetched and verified by the ICDB client. Thus, an ICDB functional query in the AV mode of the DMV model will need to perform the following steps:

<table>
<thead>
<tr>
<th>ICDB Functional query process steps in the AV mode of the DMV model:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. SELECT operation: The ICDB client will need to issue SELECT queries $Q_1$ and $Q_2$, where the processing steps for $Q_1$ include data fetch and verification, whereas the processing steps for $Q_2$ include IC fetch and AIC generation. Check the detailed steps required for a SELECT query in the AV mode of the DMV model described earlier.</td>
</tr>
<tr>
<td>2. Query execution in local: The ICDB client now is able to compute the aggregate value over the fetched data. The aggregate function could be SUM, MIN, MAX, AVG, or COUNT.</td>
</tr>
</tbody>
</table>

Figures 5.29, 5.30 and 5.31 and Tables 5.29, 5.30 and 5.31 show the experimental results for the functional query in the AV mode for the DMV model. The functional query used for this experiment includes four aggregate operations (SUM, MIN, MAX, AVG) in a single Functional query. The data shows that the process time for a functional query in the AV mode is similar to the corresponding SELECT query (including $Q_1$ and $Q_2$) process time. The only additional time is the time for the ICDB client to locally compute the aggregate values over the fetched and verified data. This local computation is insignificant.
The average performance penalty rate for functional query using CMAC-AES in the AV mode is the lowest among all three algorithms with 41.7 in OCT and 70.52 in OCF. The average penalty rate is slightly higher for HMAC-SHA with 49.66 in OCT and 81.95 in OCF. Both MAC algorithms have an increased performance penalty rate in the AV mode compared to their penalty rate in the basic model. However for RSA, the average penalty rate is 62.54 in OCT and 117.24 in OCF, which is a huge improvement comparing to the basic model with the average penalty rate for RSA at 197.38 in OCT and 534.33 in OCF. Again, the reason for this improvement is because RSA’s homomorphic property avoids the need to regenerate all ICs in the aggregate verification mode.
Figure 5.29: HMAC-SHA plotted query processing time against the number of tuples returned using a SELECT Query corresponding to the Functional Query.

Table 5.29: HMAC-SHA plotted query processing time against the number of tuples returned using a SELECT Query corresponding to the Functional Query.

<table>
<thead>
<tr>
<th>No. of Tuples</th>
<th>Scheme</th>
<th>IC Fetch</th>
<th>AIC generation</th>
<th>Data Fetch</th>
<th>Verification</th>
<th>Total Query Process</th>
<th>Performance Penalty Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>50k</td>
<td>DB</td>
<td>17.4</td>
<td></td>
<td></td>
<td></td>
<td>17.4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>67.8</td>
<td>100.6</td>
<td>159.8</td>
<td>447.3412</td>
<td>775.5312</td>
<td>44.57075862</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>107.94</td>
<td>212</td>
<td>125.86</td>
<td>926.4222</td>
<td>1372.2222</td>
<td>78.86344483</td>
</tr>
<tr>
<td>100k</td>
<td>DB</td>
<td>30.1</td>
<td></td>
<td></td>
<td></td>
<td>30.1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>97.4</td>
<td>126.2</td>
<td>253.6</td>
<td>1001.0922</td>
<td>1478.2922</td>
<td>49.11269767</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>108.5</td>
<td>236.6</td>
<td>265.72</td>
<td>1664.3822</td>
<td>2275.2022</td>
<td>75.58811296</td>
</tr>
<tr>
<td>150k</td>
<td>DB</td>
<td>39.6</td>
<td></td>
<td></td>
<td></td>
<td>39.6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>136</td>
<td>224.2</td>
<td>365.6</td>
<td>1458.3462</td>
<td>2184.1462</td>
<td>55.15405455</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>161.28</td>
<td>306.2</td>
<td>400.12</td>
<td>2566.3172</td>
<td>3433.9172</td>
<td>86.71508081</td>
</tr>
<tr>
<td>200k</td>
<td>DB</td>
<td>53.9</td>
<td></td>
<td></td>
<td></td>
<td>53.9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>187.6</td>
<td>256</td>
<td>497.8</td>
<td>1744.185</td>
<td>2641.965</td>
<td>49.8249833</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>192.78</td>
<td>555.4</td>
<td>445.34</td>
<td>3477.1536</td>
<td>4670.6736</td>
<td>86.65442672</td>
</tr>
</tbody>
</table>
Figure 5.30: CMAC-AES plotted query processing time against the number of tuples returned using a SELECT Query corresponding to the Functional Query

Table 5.30: CMAC-AES plotted query processing time against the number of tuples returned using a SELECT Query corresponding to the Functional Query
Figure 5.31: RSA plotted query processing time against the number of tuples returned using a SELECT Query corresponding to the Functional Query

Table 5.31: RSA plotted query processing time against the number of tuples returned using a SELECT Query corresponding to the Functional Query

<table>
<thead>
<tr>
<th>No. of Tuples</th>
<th>Scheme</th>
<th>IC Fetch</th>
<th>AIC generation</th>
<th>Data Fetch</th>
<th>Verification</th>
<th>Total Query Process</th>
<th>Performance Penalty Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>50k</td>
<td>DB</td>
<td>17.4</td>
<td></td>
<td>17.4</td>
<td></td>
<td>17.4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>90.6</td>
<td>273.8</td>
<td>147.2</td>
<td>424.8366</td>
<td>905.4366</td>
<td>52.03658621</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>76.16</td>
<td>842.4</td>
<td>261.1</td>
<td>698.252</td>
<td>1877.912</td>
<td>107.925977</td>
</tr>
<tr>
<td>100k</td>
<td>DB</td>
<td>30.1</td>
<td></td>
<td>30.1</td>
<td></td>
<td>30.1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>90.2</td>
<td>560.8</td>
<td>218.6</td>
<td>957.8404</td>
<td>1827.1804</td>
<td>60.7036677</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>157.36</td>
<td>1549.2</td>
<td>325.22</td>
<td>1271.4308</td>
<td>3303.2108</td>
<td>109.7412226</td>
</tr>
<tr>
<td>150k</td>
<td>DB</td>
<td>40.6</td>
<td></td>
<td>40.6</td>
<td></td>
<td>40.6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>144.8</td>
<td>916.8</td>
<td>380</td>
<td>1313.3244</td>
<td>2754.9244</td>
<td>69.56879798</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>174.16</td>
<td>2480.6</td>
<td>456.26</td>
<td>2282.7696</td>
<td>5413.7896</td>
<td>134.1866661</td>
</tr>
<tr>
<td>200k</td>
<td>DB</td>
<td>53.9</td>
<td></td>
<td>53.9</td>
<td></td>
<td>53.9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OCT</td>
<td>271.4</td>
<td>1248.6</td>
<td>493</td>
<td>1643.416</td>
<td>3666.716</td>
<td>67.8425974</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>395.06</td>
<td>3072.4</td>
<td>476.84</td>
<td>2458.3448</td>
<td>6312.7448</td>
<td>117.1195696</td>
</tr>
</tbody>
</table>
Summary

Tables 5.32 and 5.33 list the memory penalty rates and the average performance penalty rates for all of the experiments in both the ICDB basic and the DMV models.

We also summarize our experiments for all twelve different combinations of ICDB schemes based on their memory penalty and performance penalty.

Table 5.32: Memory penalty rates for all experiments in both the ICDB basic and the DMV models

<table>
<thead>
<tr>
<th>Database</th>
<th>Scheme</th>
<th>Basic model</th>
<th>DMV model</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>SHA</td>
<td>AES</td>
</tr>
<tr>
<td>Employees</td>
<td>OCT</td>
<td>1.53</td>
<td>1.53</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>3.15</td>
<td>3.15</td>
</tr>
</tbody>
</table>

Table 5.33: Average performance penalty rates for all experiments in both ICDB basic and DMV models

<table>
<thead>
<tr>
<th>Query</th>
<th>Scheme</th>
<th>Basic model</th>
<th>DMV model</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>SHA</td>
<td>AES</td>
</tr>
<tr>
<td>Select</td>
<td>OCT</td>
<td>13.29</td>
<td>11.07</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>31.39</td>
<td>25.24</td>
</tr>
<tr>
<td>Insert</td>
<td>OCT</td>
<td>2.69</td>
<td>2.68</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>5.14</td>
<td>4.94</td>
</tr>
<tr>
<td>Delete</td>
<td>OCT</td>
<td>1.25</td>
<td>1.25</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>1.62</td>
<td>1.53</td>
</tr>
<tr>
<td>Functional</td>
<td>OCT</td>
<td>36.05</td>
<td>30.09</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>53.46</td>
<td>47.82</td>
</tr>
<tr>
<td>Join</td>
<td>OCT</td>
<td>13.56</td>
<td>12.18</td>
</tr>
<tr>
<td></td>
<td>OCF</td>
<td>36.36</td>
<td>31.73</td>
</tr>
</tbody>
</table>

OCT vs. OCF.

1. In all cases, OCT has a lower memory penalty and performance penalty than OCF does.

2. However, a very important advantage OCF maintains over OCT is that OCF is able to detect whether a particular field entry/data is corrupted, while OCT is only able to detect whether a tuple is corrupted.
3. Though OCT costs less memory and has less performance penalty, all queries need to fetch entire tuples even if the query only requests values from a single column. There is an opportunity for further research work to investigate reducing the volume of additional information related to the data, using the Authenticated Data Structures (ADS) such as the Merkel Hash Trees described in [5]. In such structures, instead of fetching the entire tuple, one need only fetch the requested data and the hash value of the data required to compute the root of the Hash Tree. In order to support this operation, though, the ICDB client has to compute multiple hashes before uploading the data. Also, the cloud database server itself has to compute multiple hashes or store precomputed hashes of each leaf or internal node. The change in the database structure and query processing will make the ICDB no longer transparent to the cloud servers and thus we did not adopt the ADS structure in this research. Nonetheless, This may still be considered for future research.

**HMAC-SHA vs. CMAC-AES vs. RSA.**

1. The large (1024 bits) ICs for RSA incur more memory penalty than the MAC algorithms with 128-bit ICs.

2. CMAC-AES has a slightly lower performance penalty rate than HMAC-SHA in all experiments.

3. Among the three cryptographic algorithms, in the basic model, RSA has the maximum performance penalty rate. IN OCT, RSA’s performance penalty rate is about 2 (INSERT operation) to 10 (SELECT operation) times more than the penalty rates of MAC algorithms. IN OCF, RSA’s performance penalty rate is
about 4 (INSERT operation) to 15 (SELECT operation) times more than the penalty rates of MAC algorithms.

Similarly, in the AV mode of DMV model, RSA again has the highest performance penalty rate. IN OCT, RSA’s performance penalty rate is about 1.1 (DELETE operation) to 1.5 (SELECT operation) times more than the penalty rates of MAC algorithms. IN OCF, RSA’s performance penalty rate is about 1.2 (DELETE operation) to 1.7 (SELECT operation) times more than the penalty rates of MAC algorithms.

We can see that in the AV mode, RSA still performs worse than MAC algorithms but it gets much closer to the performance of MAC algorithms. The reason for this is because the RSA’s homomorphic property allows the ICDB client to generate the AIC directly without having to regenerate all ICs in the verification process in the AV mode.

**SELECT vs. INSERT vs. DELETE vs. JOIN vs. Functional**

1. Among all query types, based on our experiments in the basic model, the ranking of incurred performance penalty, from the least to the greatest, is DELETE < INSERT << SELECT ≈ JOIN < Functional, where the symbol << means a notable (significant) increase.

2. Similarly for the AV mode of the DMV model, the ranking of the performance penalty is the same, i.e., DELETE < INSERT << SELECT ≈ JOIN < Functional.
Basic Model vs. DMV Model.

1. Introducing the AV mode of the DMV model on top of the basic ICDB model has reduced the network overhead since the ICDB client need no longer fetch ICs along with the data from the database servers through the network.

2. Since the cloud database server has to store all the ICs and serials in both models, the memory penalty for both models is the same.

3. For the DMV model when compared to the basic model, though MAC algorithms did not improve (actually worse) their performance, there was a huge performance improvement for the RSA algorithm. We see that RSA did not fare very well in the basic model. It has comparatively large integrity code size, and takes a much longer time to verify than HMAC-SHA or CMAC-AES does. However, in the aggregate verification (of the DMV model), the results are comparable to CMAC-AES and HMAC-SHA.
**Ranking all the ICDB Schemes:** We have suggested a ranking for all the ICDB schemes from our experimental results based on the incurred memory penalty and performance penalty. Table 5.35 provides a quick view of the rankings for all the schemes.

1. The ranking is based on a scale of Low to High as shown in the table 5.34. A ‘High’ ranking in the scale refers to the worst result (highest penalty rate) and a ‘Low’ ranking refers to the best result (least penalty rate) in our experiment.

2. Basic-OCF-RSA has High rankings in both metrics and thus is the worst, whereas DVM(AV Mode)-OCT-HMAC and DVM(AV Mode)-OCT-CMAC have low rankings in both metrics and thus are the best. On the other hand, OCF-RSA schemes have better functionality which allow finer integrity protection granularity to the level of each field entry and the allow the cloud database server to perform the homomorphic operation (homomorphic multiplication of numeric data) on behalf of the data owner.

3. The performance penalty for RSA is reduced significantly in the AV mode of the DMV model compared to the basic model due to the homomorphic property of RSA.

4. Another major advantage of the AV mode of the DMV model compared to the basic model is the reduction of network overhead by transmitting a single AIC instead of many individual ICs over the network.
Table 5.34: Ranking Scale

<table>
<thead>
<tr>
<th>Scale</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low (1)</td>
</tr>
<tr>
<td>Moderate Low (2)</td>
</tr>
<tr>
<td>Intermediate (3)</td>
</tr>
<tr>
<td>Moderate High (4)</td>
</tr>
<tr>
<td>High (5)</td>
</tr>
</tbody>
</table>

Table 5.35: Ranking of different ICDB schemes in either Basic or DMV model

<table>
<thead>
<tr>
<th>MODELS</th>
<th>Granularity</th>
<th>Algorithms</th>
<th>Memory Penalty</th>
<th>Performance Penalty</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic model</td>
<td></td>
<td>OCF</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>HMAC-SHA</td>
<td>Moderate Low (2)</td>
<td>Moderate Low (2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CMAC-AES</td>
<td>Moderate Low (2)</td>
<td>Moderate Low (2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RSA</td>
<td>High (5)</td>
<td>High (5)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OCT</td>
<td>Low (1)</td>
<td>Low (1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HMAC-SHA</td>
<td>Low (1)</td>
<td>Low (1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CMAC-AES</td>
<td>Low (1)</td>
<td>Low (1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RSA</td>
<td>Intermediate (3)</td>
<td>Moderate High (4)</td>
</tr>
<tr>
<td>DMV model (AV Mode)</td>
<td></td>
<td>OCF</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>HMAC-SHA</td>
<td>Moderate Low (2)</td>
<td>Moderator Low (2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CMAC-AES</td>
<td>Moderate Low (2)</td>
<td>Moderate Low (2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RSA</td>
<td>High (5)</td>
<td>Intermediate (3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OCT</td>
<td>Low (1)</td>
<td>Low (1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HMAC-SHA</td>
<td>Low (1)</td>
<td>Low (1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CMAC-AES</td>
<td>Low (1)</td>
<td>Low (1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RSA</td>
<td>Intermediate (3)</td>
<td>Moderate Low (2)</td>
</tr>
</tbody>
</table>
CHAPTER 6

CONCLUSION AND FUTURE WORK

In this thesis, we proposed an ICDB approach to protect data integrity for outsourced databases in the cloud. The approach inserts a cryptographically generated integrity code for each data item to be protected. The way we construct the integrity code ensures that nobody except the data owner is able to modify the data and at the mean time generates a matching integrity code for it. Using such an integrity code, any forged data can be detected. In addition, our ICDB approach assigns a unique serial number to each integrity code and proposes a scheme similar to the X.509 standard to ensure the freshness of outsourced data, where the X.509 standard is a scheme to manage all unexpired public key certificates.

We have implemented an ICDB working prototype which was used to conduct empirical experiments to evaluate the memory and performance penalty for each ICDB scheme. We have shown all the experimental results and analyzed their indications/implications in Chapter 5.

In addition to empirical experiments, we also investigated the pricing schemes of existing database service providers so that we are able to suggest which ICDB scheme(s) may be the best choice economically for each pricing scheme in the real world service providers.

Cloud Services Schemes: Different Cloud Service Providers have their own
pricing schemes. Reviewing the pricing schemes of three major cloud providers with the highest market share, Amazon Web Services (AWS) [27], Google Cloud Platform [29] and Microsoft Azure [28] has helped us to relate our work to the real cloud environment. Cloud Relational Database Services (RDS) charge customers based on the storage, data inflow/outflow and number of instances. Some cloud providers such as Microsoft Azure offer a scheme in package (with a fixed rate of charge) with maximum database instances per pool, maximum storage per pool, maximum transaction per pool for an elastic pool of databases. Google cloud offers free inbound data but charges for all outbound data. Similarly, Amazon Web Services (AWS) has its own scheme for RDS pricing. It charges per hour of the instance used and GB per month of storage. As with Google cloud, AWS too is free for inbound data and charges per GB of outbound data.

**ICDB Scheme choices based on Cloud Services Schemes:** If the database size and the number of transactions for the database will never exceed the maximum limit offered by the cloud service package, then all ICDB schemes can be considered. In this case, if the database application requires integrity protection level down to each data field, then OCF schemes will be the choice. Furthermore, if the database application requires the database server to perform homomorphic operations on behalf of the data owner, then the ICDB schemes using the RSA algorithm will be the choice.

For cloud services that charge only for outbound data, the ICDB schemes in the AV mode of DMV model is the best choice since the outbound data is almost the same as a standard SQL database with just additional data for serial numbers. This will keep the cost similar to standard SQL databases.

If the database size and amount of inflow/outflow data (which directly impact on the database size) are unpredictable, then the choice of service should be the one
with an on-demand pricing scheme that charges based on the usage. CMAC-AES in OCT has the minimum performance penalty and only increases the database size by approximately 1.53 times, compared to the standard database. Thus, CMAC-OCT should be the best choice in this case.

Table 5.35 listing all 12 ICDB schemes with their performance and memory rankings should provide useful information for users to decide which ICDB scheme and which database service provider can benefit them the most.

**Future Work**

1. Although the experimental results presented in this thesis provide an adequate level of understanding of the ICDB performance, they are not exhaustive. Additional tests could be conducted separately. For instance, the benchmarks provided have only been tested on the Employees [21] sample database, which is approximately 196.4 MB in size. More databases with larger sizes can be tested to gain a better understanding of performance.

2. The experiment could be performed on the real cloud DB service provider and cloud application to study the performance in a real cloud environment.

3. The implementation provided is configured to only communicate with a MySQL database, but other database options can be tested (e.g., PostgreSQL, SQLite).

4. We have only used a SELECT * query for our experiments. There are notable performance differences between OCT and OCF, namely that OCF does not necessarily need to return entire tuples when queried but in this case (SELECT
*) it has to. Thus, experiments based on a different SELECT query could reduce the performance gap between OCT and OCF.

5. All the experimental results provided in this thesis are based on the integrity verification of returned query results. Incomplete query results cannot be detected with the current ICDB models. Thus, future research is necessary to assure the completeness of queried data returned from the cloud database server.
REFERENCES


APPENDIX A

EMPLOYEES DATABASE SCHEMA
-- Sample employee database

See changelog table for details

Copyright (C) 2007,2008, MySQL AB

Original data created by Fusheng Wang and Carlo Zaniolo

http://www.cs.aau.dk/TimeCenter/software.htm

http://www.cs.aau.dk/TimeCenter/Data/employeeTemporalDataSet.zip

Current schema by Giuseppe Maxia

Data conversion from XML to relational by Patrick Crews

This work is licensed under the

Creative Commons Attribution-Share Alike 3.0 Unported License.

To view a copy of this license, visit

http://creativecommons.org/licenses/by-sa/3.0/ or send a letter to

Creative Commons, 171 Second Street, Suite 300, San Francisco,
California, 94105, USA.

DISCLAIMER

To the best of our knowledge, this data is fabricated, and
it does not correspond to real people.

Any similarity to existing people is purely coincidental.

CREATE TABLE employees (  
emp_no INT NOT NULL,
CREATE TABLE employees (  
birth_date DATE NOT NULL,  
first_name VARCHAR(14) NOT NULL,  
last_name VARCHAR(16) NOT NULL,  
gender ENUM ('M','F') NOT NULL,  
hire_date DATE NOT NULL,  
  PRIMARY KEY (emp_no)  
);

CREATE TABLE departments (  
department_no CHAR(4) NOT NULL,  
department_name VARCHAR(40) NOT NULL,  
  PRIMARY KEY (department_no),  
  UNIQUE KEY (department_name)  
);

CREATE TABLE dept_manager (  
emp_no INT NOT NULL,  
department_no CHAR(4) NOT NULL,  
from_date DATE NOT NULL,  
to_date DATE NOT NULL,  
  FOREIGN KEY (emp_no) REFERENCES employees (emp_no)  
    ON DELETE CASCADE,  
  FOREIGN KEY (department_no) REFERENCES departments (department_no)  
    ON DELETE CASCADE,  
  PRIMARY KEY (emp_no,department_no)  
)
CREATE TABLE dept_emp (  
  emp_no INT NOT NULL,  
  dept_no CHAR(4) NOT NULL,  
  from_date DATE NOT NULL,  
  to_date DATE NOT NULL,  
  FOREIGN KEY (emp_no) REFERENCES employees (emp_no)  
    ON DELETE CASCADE,  
  FOREIGN KEY (dept_no) REFERENCES departments (dept_no)  
    ON DELETE CASCADE,  
  PRIMARY KEY (emp_no,dept_no)  
);  

CREATE TABLE titles (  
  emp_no INT NOT NULL,  
  title VARCHAR(50) NOT NULL,  
  from_date DATE NOT NULL,  
  to_date DATE,  
  FOREIGN KEY (emp_no) REFERENCES employees (emp_no)  
    ON DELETE CASCADE,  
  PRIMARY KEY (emp_no,title, from_date)  
);  

CREATE TABLE salaries (  
  emp_no INT NOT NULL,  
  salary DECIMAL NOT NULL,  
  from_date DATE NOT NULL,  
  to_date DATE,  
  FOREIGN KEY (emp_no) REFERENCES employees (emp_no)  
    ON DELETE CASCADE,  
  PRIMARY KEY (emp_no)  
);
emp_no    INT    NOT NULL,
salary    INT    NOT NULL,
from_date DATE    NOT NULL,
to_date   DATE    NOT NULL,

FOREIGN KEY (emp_no) REFERENCES employees (emp_no)
    ON DELETE CASCADE,

PRIMARY KEY (emp_no, from_date)

);
APPENDIX B

HARDWARE SPECIFICATIONS
We performed all the testing for ICDB on Onyx server. Onyx is a Boise State University’s multiuser Linux server for students and faculty. Multi-user Linux computers may be accessed concurrently by more then one user. The remote connection will require two components on the computer, an X11 server and an SSH client. The server is accessible through onyx.boisestate.edu. The hardware specifications are as follows:

- Architecture: x86_64
- CPU op-mode(s): 32-bit, 64-bit
- Byte Order: Little Endian
- CPU(s): 32
- On-line CPU(s) list: 0-31
- Thread(s) per core: 2
- Core(s) per socket: 8
- Socket(s): 2
- NUMA node(s): 2
- Vendor ID: GenuineIntel
- CPU family: 6
- Model: 63
- Model name: Intel(R) Xeon(R) CPU E5-2640
- v3 @ 2.60GHz
- Stepping: 2
- CPU MHz: 1401.664
- BogoMIPS: 5203.83
- Virtualization: VT-x
- L1d cache: 32K
L1i cache: 32K
L2 cache: 256K
L3 cache: 20480K
NUMA node0 CPU(s): 0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30
NUMA node1 CPU(s): 1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31
APPENDIX C

DEVELOPMENT ENVIRONMENT:
Language: Java, Kotlin, JDBC for database connectivity

Database: MySQL

IDE: IntelliJ IDEA
APPENDIX D

ICDB COMMANDS
D.1 Initial Setup

To be able to build and run against a database, the following must be installed:

Maven

MySQL

Then run the following commands to build the project:

1. git clone https://github.com/ujjwalkarki/ICDB-CloudApp.git
2. cd IntegrityCodedDatabase/ICDB
3. make

D.2 Running the ICDB tool

The arguments for interacting with the ICDB tool is as follows:

```
icdb [-c config-file] [command] [options]
```

icdb is a bash script that simply runs the compiled jar. This can be run directly:

```
java -jar target/icdb-capsule.jar [-c config-file] [command] [options]
```

All interactions with the tool will require a config file containing a JSON object with several parameters listed below:

- `ip` - the target MySQL database IP address
- `port` - the port the database is running on
- `user` - database user
- `password` - database password (if any)
- `schema` - database schema to use (for conversion)
- `icdbSchema` - ICDB database schema name (for execution and verification)
- `algorithm` - the encryption algorithm to use (RSA, AES, SHA, RSA_AGGREGATE, AES_AGGREGATE and SHA_AGGREGATE)
granularity - use code per field or code per tuple (FIELD or TUPLE)
macKey - 128-bit MAC key encoded as a base64 string
rsaKeyFile - PEM file containing public and private RSA keys

For convenience, a config file is given at ./ICDB/config.json, which will be loaded by default if the -c option is not specified.

The default config provides the following JSON object:

```
"ip": "localhost",
"port": 10154,
"user": "msandbox",
"password": "msandbox",
"schema": "employees",
"icdbSchema": "employees_icdb",
"algorithm": "SHA",
"granularity": "TUPLE",
"macKey": "qyPTqFrPGUpxcIo9sz2MdQ==",
"rsaKeyFile": "key.pem"
```

D.3 Commands Available

1. convert-db - Converts an existing DB to an ICDB (both schema and data)
2. convert-query - Converts a DB query to an ICDB query
3. execute-query - Executes an ICDB query and verifies all returned data
D.3.1 Convert DB Command

The convert-db command has 4 phases, any of which can be skipped: –skip-duplicate, if set, the duplicate DB step will be skipped –skip-schema, if set, the schema conversion step will be skipped –skip-data, if set, the data conversion step will be skipped –skip-load, if set, the data load step will be skipped. The command is:

```
```

Example:
```
cd project-root/ICDB
icdb convert-db
```

D.3.2 Convert Query Command

The convert-query command takes the SQL query as an input and converts it to an ICDB query. The conversion requires: -q ”The SQL query, passed in as a string”. The command is:

```
convert-query [-q query]
```

Example:
```
cd project-root/ICDB icdb convert-query -q "SELECT * FROM employees;"
```

D.3.3 Execute Query Command

The convert-data command takes a SQL query as input, executes, then verifies any returned data. -q ”The SQL query, passed in as a string” THe command is:

```
execute-query [-q query]
```

Example:
```
cd project-root/ICDB icdb execute-query -q "SELECT * FROM employees;"
```
APPENDIX E

QUERY CONVERSION:
We used different mySQL queries to convert it to the ICDB queries which are listed below:

**E.1 MySQL Select Queries:**

1. SELECT * FROM departments;

2. SELECT emp_no, birth_date, first_name, last_name, gender, hire_date FROM employees WHERE gender='M';

3. SELECT dept_no, from_date, to_date FROM dept_emp;

4. SELECT emp_no FROM salaries WHERE salary>60000;

5. SELECT * FROM titles;

6. SELECT departments.dept_no,departments.dept_name, dept_manager.emp_no, dept_manager.from_date,dept_manager.to_date FROM departments INNER JOIN dept_manager ON departments.dept_no=dept_manager.dept_no;

**E.1.1 ICDB Select Queries (OCF):**

1. SELECT dept_no, dept_name,dept_no_ic,dept_no_serial, dept_name_ic, dept_name_serial FROM departments;

2. SELECT emp_no, birth_date, first_name, last_name, gender, hire_date,emp_no_ic, emp_no_serial, birth_date_ic, birth_date_serial, first_name_ic, first_name_serial, last_name_ic, last_name_serial, gender_ic, gender_serial, hire_date_ic, hire_date_serial FROM employees WHERE gender = 'M';
3. SELECT dept_no, from_date, to_date, emp_no, dept_no_ic, dept_no_serial, from_date_ic, from_date_serial, to_date_ic, to_date_serial, emp_no_ic, emp_no_serial FROM dept_emp;

4. SELECT emp_no, salary, from_date, emp_no_ic, emp_no_serial, salary_ic, salary_serial, from_date_ic, from_date_serial FROM salaries WHERE salary > 60000;

5. SELECT from_date, title, emp_no, to_date, from_date_ic, from_date_serial, title_ic, title_serial, emp_no_ic, emp_no_serial, to_date_ic, to_date_serial FROM titles;

6. SELECT departments.dept_no, departments.dept_name, dept_manager.emp_no, dept_manager.from_date, dept_manager.to_date, departments.dept_no, dept_manager.emp_no, dept_manager.dept_no, departments.dept_no_ic, departments.dept_no_serial, departments.dept_name_ic, departments.dept_name_serial, dept_manager.emp_no_ic, dept_manager.emp_no_serial, dept_manager.dept_no, departments.dept_name, departments.dept_no, dept_manager.emp_no, dept_manager.dept_no_ic, dept_manager.dept_no_serial FROM departments INNER JOIN dept_manager ON departments.dept_no = dept_manager.dept_no;

E.1.2 ICDB Select Queries(OCT):

1. SELECT dept_no, dept_name, ic, serial FROM departments;

2. SELECT emp_no, birth_date, first_name, last_name, gender, hire_date, ic, serial FROM employees WHERE gender = 'M';

3. SELECT emp_no, dept_no, from_date, to_date, ic, serial FROM dept_emp;

4. SELECT emp_no, salary, from_date, to_date, ic, serial FROM salaries WHERE salary > 60000;
5. SELECT emp_no, title, from_date, to_date, ic, serial FROM titles;

6. SELECT departments.dept_no, departments.dept_name, departments.ic, departments.serial, dept_manager.emp_no, dept_manager.dept_no, dept_manager.from_date, dept_manager.to_date, dept_manager.ic, dept_manager.serial FROM departments INNER JOIN dept_manager ON departments.dept_no = dept_manager.dept_no;

INSERT query conversion is not included in the appendix due to the large IC values in ICDB Query for each of the inserting attributes or tuples.

E.2 MySQL DELETE Queries

1. DELETE FROM departments WHERE dept_no='d006';

2. DELETE FROM salaries WHERE salary>60000 AND salary<65000;

3. DELETE FROM employees WHERE gender='M';

4. DELETE FROM titles;

5. DELETE FROM employees WHERE gender='M' AND last_name='Terkki';

6. DELETE FROM titles WHERE title!='Staff';

E.2.1 ICDB Delete Verification Queries (OCF):

1. SELECT dept_no, dept_name, dept_no_ic, dept_no_serial, dept_name_ic, dept_name_serial FROM departments WHERE dept_no = 'd006';

2. SELECT emp_no, salary, from_date, to_date, emp_no_ic, emp_no_serial, salary_ic, salary_serial, from_date_ic, from_date_serial, to_date_ic, to_date_serial FROM salaries WHERE salary > 60000 AND salary < 65000;
3. SELECT emp_no, birth_date, first_name, last_name, gender, hire_date, emp_no_ic, 
   emp_no_serial, birth_date_ic, birth_date_serial, first_name_ic, first_name_serial, 
   last_name_ic, last_name_serial, gender_ic, gender_serial, hire_date_ic, hire_date_serial 
   FROM employees WHERE gender = 'M';

4. SELECT emp_no, title, from_date, to_date, emp_no_ic, emp_no_serial, title_ic, 
   title_serial, from_date_ic, from_date_serial, to_date_ic, to_date_serial 
   FROM titles;

5. SELECT emp_no, birth_date, first_name, last_name, gender, hire_date, emp_no_ic, 
   emp_no_serial, birth_date_ic, birth_date_serial, first_name_ic, first_name_serial, 
   last_name_ic, last_name_serial, gender_ic, gender_serial, hire_date_ic, hire_date_serial 
   FROM employees WHERE gender = 'M' AND last_name = 'Terkki';

6. SELECT emp_no, title, from_date, to_date, emp_no_ic, emp_no_serial, title_ic, 
   title_serial, from_date_ic, from_date_serial, to_date_ic, to_date_serial 
   FROM titles WHERE title != 'Staff';

**E.2.2 ICDB Delete Verification Queries (OCT):**

1. SELECT dept_no, dept_name, ic, serial FROM departments WHERE dept_no = 'd006';

2. SELECT emp_no, salary, from_date, to_date, ic, serial FROM salaries WHERE salary > 60000 AND salary < 65000;

3. SELECT emp_no, birth_date, first_name, last_name, gender, hire_date, ic, serial 
   FROM employees WHERE gender = 'M';

4. SELECT emp_no, title, from_date, to_date, ic, serial FROM titles;
5. SELECT emp_no, birth_date, first_name, last_name, gender, hire_date, ic, serial
   FROM employees WHERE gender = 'M' AND last_name = 'Terkki';

6. SELECT emp_no, title, from_date, to_date, ic, serial FROM titles WHERE title
   != 'Staff';

E.3 MySQL Functional Query:

1. Select sum(salary) from salaries;

E.3.1 ICDB Functional Query(OCF):

1. Select salary, salary_ic, salary_serial, emp_no, emp_no_ic, emp_no_serial, from_date,
   from_date_ic, from_date_serial from salaries;

E.3.2 ICDB Functional Query(OCT):

1. Select * from salaries;