2015

A Comparison of Cloud Computing Database Security Algorithms

Joseph A. Hoeppner
University of North Florida, joey.hoeppner@gmail.com

A COMPARISON OF CLOUD COMPUTING DATABASE SECURITY ALGORITHMS

by

Joseph Hoeppner

A thesis submitted to the
School of Computing
in partial fulfillment of the requirements for the degree of

Master of Science in Computer and Information Sciences

UNIVERSITY OF NORTH FLORIDA
SCHOOL OF COMPUTING

December, 2015

The thesis "A Comparison of Cloud Computing Database Security Algorithms" submitted by Joseph Hoeppner in partial fulfillment of the requirements for the degree of Master of Science in Computer and Information Sciences has been

Approved by the thesis committee: Date

_____ _____
Dr. Behrooz Seyed-Abbassi
Thesis Advisor and Committee Chairperson

_____ _____
Dr. Ken Martin

_____ _____
Dr. Roger Eggen

Accepted for the School of Computing:

_____ _____
Dr. Sherif Elfayoumy
Director of the School

Accepted for the College of Computing, Engineering, and Construction:

_____ _____
Dr. Mark Tumeo
Dean of the College

Accepted for the University:

_____ _____
Dr. John Kantner
Dean of the Graduate School

ACKNOWLEDGEMENT

This thesis is dedicated to God, my family, my friends, and my Bronies. Thank you for showing me the love and support I needed to finally complete this chapter of my life <3.

CONTENTS

FIGURES

ABSTRACT

The cloud database is a relatively new type of distributed database that allows companies and individuals to purchase computing time and memory from a vendor. This allows a user to only pay for the resources they use, which saves them both time and money. While the cloud in general can solve problems that have previously been too costly or time-intensive, it also opens the door to new security problems because of its distributed nature. Several approaches have been proposed to increase the security of cloud databases, though each seems to fall short in one area or another.

This thesis presents the Hoeppner Security Algorithm (HSA) as a solution to these security problems. The HSA safeguards user's data and metadata by adding fake records alongside the real records, breaking up the database by column or groups of columns, and by storing each group in a different cloud. The efficiency and security of this algorithm was compared to the Alzain algorithm (one of the proposed security solutions that inspired the HSA), and it was found that the HSA outperforms the Alzain algorithm in most every way.

Chapter 1

INTRODUCTION

The cloud database field is a new and upcoming field of study, though the premise of distributed computing has been around for many years [Maryanski80] [Hevner79]. Though the use of the term "cloud" is still slightly nebulous, it generally denotes a distributed computing system where users pay to use hosted virtual computers from a company. Users only pay for the amount of time and memory used by their virtual computer, making cost effectiveness one of the cloud's major perks. The cloud is also the next step for distributed database systems. Because the cloud is cost effective, extremely fault tolerant, and makes data highly available, it is a natural environment for hosting multi-user databases [Delettre11]. Several security issues are also introduced because of the cloud's public nature. Solutions to these problems are currently being developed and in the future, cloud database systems may be as secure as personal computers.

One of the largest security problems that the cloud faces is that a user's data are potentially available to both the hosting agency and hackers. For this reason, security measures must be added to the cloud on top of typical data encryption to further hide users' data and metadata. This thesis examines security problems that are created or inherited by the cloud computing paradigm. Some solutions to these problems are examined, and a potential method for increasing a cloud database's security is proposed and

explained. The chapter breakdown is as follows. Chapter 1 identifies some of the security flaws inherent in the cloud and proposes a high-level solution. Chapter 2 gives a background to the cloud, and describes the essential characteristics a cloud environment must have, the types of clouds that exist, and some of the most popular cloud service models. Chapter 3 defines a detailed methodology for the solution, including diagrams and flowcharted processes. Chapter 4 explains the software development and solution implementation used to remedy the cloud's targeted security flaws. Chapter 5 presents the results of the implementation, and provides comparison methodologies to the proposed solution. Chapter 6 gives a conclusion to this thesis.

1.1 Problem

This section more clearly defines some of the problems that cloud databases face. Current papers describing cloud issues and possible solutions also are presented and examined.

1.1.1 Problem Definition

Today, cloud databases and cloud computing have seen an increase in marketability as an increasing number of companies are choosing to host their services in the cloud. While hosting data and applications in the cloud certainly increases redundancy and accessibility for users, it also simultaneously introduces security issues that must be taken into account. The more accessible data are, the more likely it is they will be discovered by unwanted parties. This concern not only goes for the data, but for the surrounding metadata as well. When one hosts high-profile or sensitive documents, all facets of

security must be taken into account to ensure the utmost security. This task becomes increasingly difficult, however, once data are trusted to a third-party organization. When someone owns their own data, they naturally put in place many failsafe protocols and contingency plans since their jobs and the livelihoods of their shareholders may depend on the data's management. When another company physically controls the data, however, one must be sure that the third party will also have the proper protocols in place in case of a security breach or unscheduled downtime.

In addition to creating the need to safeguard one's data from the hosting company, moving an enterprise's databases to the cloud also necessitates increased security against malicious users. According to "Analysis on Cloud-Based Security Vulnerability Assessment," cloud virtual computers are most vulnerable from other virtual computers hosted on the same physical machine, meaning security measures and firewalls must be put in place to limit or eliminate communication with foreign virtual machines [Li10]. Measures should also be in place to diminish the severity of any security breach, such as a data distribution scheme designed to spread out groups of data that could prove disastrous if discovered together. The following section describes a security algorithm that takes this aspect of security into account.

In order for cloud computing to reach its full potential from a business standpoint, a security methodology must be created that allows users to have full confidence that their data will be safe from both the cloud hosting company and from intruders. This system should mask all types of data, metadata included, and allow users to customize the amount of security bestowed to each type of data.

1.1.2 Cloud Data Concealment

Over the years, many different encryption schemes have been created in order to protect sensitive data. Some of these security schemes have been applied to the cloud. When sending sensitive information over the internet to the cloud, for instance, secure protocols like HTTPS are commonly used. When data are stored in the cloud, users typically encrypt them so that, in the event of a security breach, hackers must decode the data in order to use them. The paper "Cloud Computing, Security and Data Concealment" suggests that this security is not sufficient for the cloud [Delettre11]. When users upload their databases to the cloud, they enhance their databases' vulnerabilities because their data are now stored in the semi-public cloud architecture and are hosted by a possibly questionable organization. Though data may be secure during transmission and while encrypted inside of the database, an enterprise's metadata, or information about the underlying data, are still vulnerable [Delettre11].

Imagine, for instance, that a small business owner decides to store their client records in a cloud database. If the cloud provider or a rival company were to break into the cloud computer where the records are stored, they would have access to the encrypted records. They might not be able to break the encryption scheme and determine what specific data are stored in the cloud, but they would be able to determine a baseline for the amount of data that is stored in the database. Now, imagine that they later broke in a second time. Again, they would not necessarily know the business owner's data, but they could compare the amount of encrypted data currently stored with the amount of encrypted data that they took the first time. This would give malicious users an idea of how the small

business is growing or shrinking, which may give them an economic advantage if they could determine which business practices resulted in the company's growth [Delettre11].

In order to prevent hackers from taking any data, including metadata, from an enterprise, Mr. Delettre suggests randomly adding fake data to the clouds whenever real records are added [Delettre11]. He developed a relatively simple algorithm for this process. First, a value generator is created for each data type stored in the database. This is a program that is capable of creating believable data values that can be stored alongside the real records. Then, the true records to be inserted are marked in such a way that only the data's owner will be able to tell them apart from fake records. Whenever records are sent to be added to the database, a random number of fake records are also generated and are added. In order to further decrease the chances of an intruder finding the real data or spotting patterns in the ratio of real records to fake records, there is a random chance that an extra random number of fake records will be inserted into the database every time a real record is inserted [Delettre11].

Though this method will add to data and metadata security, it also has its drawbacks. First, it will be difficult to create a value generator that is capable of creating realistic strings. In order for it to be passed-off as believable, the generator must make strings that are semantically correct, given the type of data that are stored in the database. Specialized generators may need to be made for each database or data type. The second potential drawback to this security scheme is determining which data items are the user-imputed ones and which are fake data. Several solutions were mentioned in "Cloud Computing, Security and Data Concealment," though not all of them seemed like they would suffice.

The first solution was to add invisible markers to the data so that the data's owner can distinguish them from the rest of the data [Delettre11]. While that sounds like a good idea, it does not seem very practical. Nothing in computing is truly invisible. For a record to be marked, a change must be stored somewhere in memory so that it can be retrieved later during the data validation process. Given enough time, it seems like a marking pattern would emerge or the "invisible" markers would be found, and then the creation of the generator and the fake data would be in vain.

Another solution that came up in Delettre's paper was to have the user input a code that would be hashed in order to select the correct data. This is a better method because it relies on a user's knowledge instead of secret stored data that can be found, but it has all of the drawbacks of a user-defined password – passwords can be forgotten, shared with other users, guessed, etc. A solution to this problem is presented in section 1.2.

1.1.3 Multi-Cloud Database

Many steps can be taken to protect data from unauthorized users, but a different strategy must be utilized to secure data against the cloud's service provider. Since they own and are in charge of the physical computers and memory that a virtual computer is running on, cloud owners are in a prime position to steal data. One security-enhancing method suggested in [Alzain11], which utilizes multiple clouds to secure data, could be used to fix this problem. This method would also make it harder for regular users to access stored data.

In his paper, "MCDB: Using Multi-clouds to Ensure Security in Cloud Computing," Mr. Alzain suggests storing a database's records across many different clouds [Alzain11]. This not only increases a database's security, but also adds to the cloud's fault tolerance and data accessibility. In order to understand how Alzain's proposed algorithm works and how these benefits come about, consider the following example of an insertion and retrieval request scenario using Alzain's algorithm. To insert a record into the multi-cloud group, it is first sent to a server where it is broken up into groups (one group per cloud used in this scheme), and then Shamir's Secret Sharing Algorithm (SSA) is applied to each group to encrypt the data. These new encrypted values are stored in the cloud databases instead of the values they represent. To retrieve a record, each part of the record is first collected from the many different clouds in the scheme. Using interpolation, each set of encrypted values is combined to undo the SSA's encryption method. Lastly, all of the decrypted values are put together to form the record that was sent to the cloud, which is returned to the user [Alzain11].

The SSA works on the mathematical principal that any polynomial of degree $n - 1$ can be uniquely determined from at least n points taken from the polynomial. Knowing this, one is able to encrypt data by storing it inside of a randomly-generated polynomial of degree $n - 1$ (a SSA polynomial), generating n input-output pairs from the polynomial, and destroying the SSA polynomial. Because the SSA polynomial can be determined by the n input-output points generated from it, the user can decrypt their data by recombining the n data points and retrieving their data from the resulting polynomial [Shamir79]. For more information on the SSA, see section 2.4.

As mentioned previously, there are many benefits to splitting data across multiple clouds. The first benefit is an increase in fault tolerance and data accessibility. Because this algorithm is based on Shamir's SSA, the retrievability of a record is determined by *k*, the number of data keys needed to construct the secret value. As long as *k* of the total *n* clouds are reachable (where *n* is the total number of data keys), the data are still retrievable. This means that ($n - k$) clouds could crash or go offline without any affect on the data [Shamir79]. To explain this with a real-world example, imagine that an enterprise has five clouds ($n = 5$), and that three key values are needed in order to distinguish real records from fake records ($k = 3$). Two whole clouds could disappear and there would be no change to the data's accessibility. If, however, the data were hosted on just one cloud, this scenario could have been devastating to the enterprise.

Another benefit to using multiple clouds is that it has the potential to drastically increase security. Since decryption of encrypted records requires *k* data values, intruders will have to break into *k* different clouds in order to gain enough information to understand what they have stolen. If each cloud is properly secured, this will prove to be a difficult task to perform. This scheme also stops Cloud Service Providers (CSP's) from accessing a user's data. If each cloud in this scheme is purchased from a different CSP, none of the CSP's will have enough information to decode the true data values. *n* CSP's would have to collaborate or steal from each other to determine the real stored data, but they would have no reason to suspect that the data they are hosting are not a user's true data. After all, if they managed to break the database's encryption scheme (not the SSA, but the encryption scheme that protects the whole enterprise), they would see numerical data that could be

passed off as real data. These advantages make multi-cloud database systems a very viable alternative to the traditional single-cloud model.

1.2 Proposed Solution

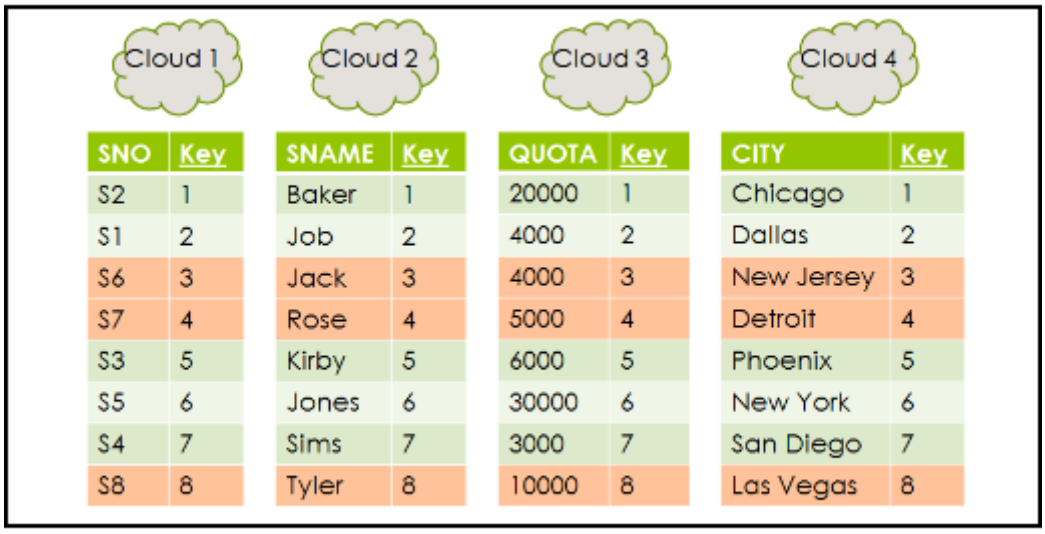
The proposed security solution relies on the use of multiple clouds and fake records to add extra layers of security to the cloud database. Each cloud used in this solution should come from a different CSP, and can be used to store a specific attribute or group of attributes of every record. For example, if each record in a database has five attributes, five clouds or five separately-stored databases should be purchased, one for each attribute. This increases the database's security because, even if a hacker were to break into one of the databases, the stolen data may not be useful without the other four associated attributes. Each cloud chosen to be included in this multi-cloud scheme is assigned a unique input value to be used in the SSA. Whenever a record is inserted into the cloud database, it is split up based on its attributes, and each attribute or attribute group is stored in a separate cloud. There is also a chance that a random amount of fake records would also be created, split up by attribute, and stored alongside the real records. Once the database's state is saved, the SSA would be used to encrypt the list of real records stored in the database, and each cloud would be given an output value from the SSA polynomial to later undo the encryption.

The insertion of records is diagramed in Figures 1.1 and 1.2. In Figure 1.1, five records (S1-S5) have been created for insertion. These five records are joined by a random amount (in this case, three) of fake records (S6-S8) that will also be inserted into the

database. Each record is assigned a unique "Key" value that is solely used for this algorithm. Each of the eight inserted records are split up by attribute to create four "Key-value" pairs, as shown in Figure 1.2. Each of these pairs is physically stored in a separate cloud for added security, though the overall database will be presented to the user as one table only containing the real records (S1-S5).

SNO	SNAME	QUOTA	CITY	Key
S1	Job	4000	Dallas	2
S2	Baker	20000	Chicago	1
S3	Kirby	6000	Phoenix	5
S4	Sims	3000	San Diego	7
S5	Jones	30000	New York	6
S6	Jack	4000	New Jersey	3
S7	Rose	5000	Detroit	4
S8	Tyler	10000	Las Vegas	8

Figure 1.1 - Eight Records Before Insertion



SNO	Key
S2	1
S1	2
S6	3
S7	4
S3	5
S5	6
S4	7
S8	8

SNAME	Key
Baker	1
Job	2
Jack	3
Rose	4
Kirby	5
Jones	6
Sims	7
Tyler	8

QUOTA	Key
20000	1
4000	2
4000	3
5000	4
6000	5
30000	6
3000	7
10000	8

CITY	Key
Chicago	1
Dallas	2
New Jersey	3
Detroit	4
Phoenix	5
New York	6
San Diego	7
Las Vegas	8

Figure 1.2 - Eight Records After Insertion

Shamir's Secret Sharing Algorithm is used to decrypt records [Shamir79]. As previously stated, each of the clouds in the multi-cloud environment contains a stored output value from a SSA polynomial. These output values can be retrieved and combined with the input values that created them to form input-output pairs. These input-output pairs could then be used to recreate the original SSA polynomial via mathematical interpolation, the list of real records could be extracted from the polynomial, and the user could be presented with just the records contained in that list. The retrieval process is diagramed in Figure 1.3. Figure 1.3 shows four clouds being contacted via SQL commands to retrieve the stored SSA polynomial input-output pairs. These pairs are then combined using mathematical interpolation as shown in the middle of the diagram to create the polynomial function from which the input-output pairs were generated. This polynomial's encrypted data can then be used to write SQL queries to reconnect the data stored in the four clouds, and to distinguish between the real and fake records.

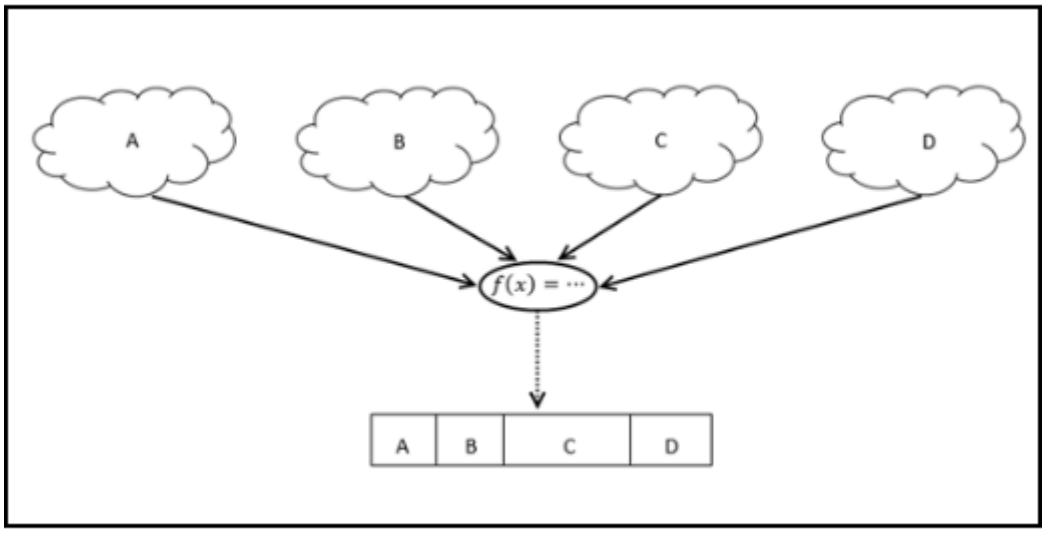


Figure 1.3 - Data Pulled from the Clouds to Recognize Real Records

This proposed security system can be customized to best suit each enterprise's need. Rather than getting a separate cloud for each table's attribute, attributes that would not be harmful if discovered together could instead be placed together in one cloud. For example, a table containing "First Name", "Last Name", and "Social Security Number" fields could be distributed into two clouds instead of three since it may not be considered risky to store a first name and last name together. Tweaks like this would not only increase the algorithm's efficiency by limiting the number of cloud connections necessary for data retrieval and storage, but would also make the proposed solution more cost-effective.

Chapter 2

BACKGROUND

The following sections describe the characteristics that every cloud environment must have, some of the most common cloud service models, and the types of clouds that can be purchased. This chapter also explains Shamir's Secret Sharing Algorithm and other relational database terms that are used throughout this thesis.

2.1 Cloud Characteristics

For a distributed database system to be considered a cloud database, it must exhibit five characteristics that all cloud computing schemes share: it must be a distributed system, must be easily expandable, must have a dynamic assignment of resources, must have high fault tolerance, and, lastly, it must have a pay-as-you-go pricing system [Delettre11]. Each of these is examined in turn to more clearly determine how the cloud differs from a traditional distributed system.

First, a cloud must be a distributed system, though it will be shown that it is a very specific type of distributed system. The fact that the cloud is a distributed system implies that it employs more than one computer connected over a network to solve a computational task. As shown below, this first cloud characteristic forms the basis

for most of the other four characteristics. The next two characteristics go hand-in-hand: the cloud must be easily expandable and must dynamically be assigned resources. As the need for computational power increases, more CPU's and memory must be allocated to the user's cloud network dynamically. The reverse must also be true. As resources are no longer needed, they can be released so that they are assignable to other cloud users. In effect, this allows users to have a computer that cannot run out of resources or computational power as long as the user can afford the extra resources. In order to create this dynamic allocation of resources, cloud computers are typically created as virtual computer images residing on a group of physical servers. As resources are requested, the hosting servers will allocate them to the requesting virtual machines. In the event that a server runs out of resources to distribute, the virtual cloud instance can be moved to a different physical server that has more resources, which also frees up all of the virtual computer's old resources for other clouds to use [Tanwer10].

Another cloud feature that naturally rises from the fact that the cloud is a distributed system is that it has a high fault tolerance. This means that an enterprise's data are backed up across several computers within the group of physical and virtual computers that make up the cloud. If one of those computers were to go offline or become unreachable, users would still be able to access their data on one of the other computers in the cloud. Lastly, users only pay for the amount of electricity, clock cycles, and memory they use. This pay-as-you-go pricing scheme allows users to perform CPU-intensive computations without having to pay for the construction of a powerful computer. They need only pay for the amount of time they spend running their programs [Delettre11]. Now that the

cloud and some of its advantages have been defined, the types of clouds and service models can be examined.

2.2 Cloud Service Models

There are typically three cloud service models: Software as a Service, Platform as a Service, and Infrastructure as a Service. These schemes define how large of a cloud environment is provided to the user and, consequently, which aspects of the system the user controls. The Software as a Service (SaaS) model allows users to host programs and their associated data in the cloud, and to only manage the details of the cloud computer that pertain to their software. SaaS becomes particularly useful when a group of users wish to run a program. Rather than having each user install the program on their local computer, the program could be hosted in the cloud for all of the users to connect to via SaaS [Vaquero09] [Jadeja12].

In contrast to SaaS, Infrastructure as a Service (IaaS) gives the user control over every aspect of the hosted cloud computer except for the maintenance of the physical computer itself. This allows users to fully control how their programs are run. They can decide everything from which operating system is used to which processes are allowed to run at a given time [Shaikh11]. The last service model, Platform as a Service (PaaS), falls between SaaS and IaaS on the scale of user control. PaaS users are in charge of the "environment" that programs are executed in, but are not responsible for the rest of the system [Jadeja12] [Vaquero09]. These three service models allow users to decide exactly

how much control they want over their cloud environment, but they are not an indication of their environment's cloud type.

2.3 Cloud Types

Clouds are generally broken up into four types: public clouds, private clouds, community clouds, and hybrid clouds. These types define which groups of users are allowed to access the computers in a specific cloud. Of the four types, public clouds are most open type of cloud. Any user or group of users is able to purchase and utilize cloud space and computation power. Private clouds, however, are reserved for a specific group or company, usually to solve a specific problem [Zheng11]. Community clouds are a slightly broader form of private cloud. They allow two or more groups or companies to share cloud resources. The difference between a community cloud and a public cloud is that the groups sharing a community cloud work towards a common goal or task, whereas public clouds do not require groups to collaborate; users are free to work independently and to fulfill their individual needs [Jadeja12]. Lastly, hybrid clouds are combinations of the other afore-mentioned cloud types [Delettre11].

2.4 Shamir's Secret Sharing Algorithm

Many security techniques rely on a user or a group of users being entrusted with a key that can be used to decrypt security measures in order to gain access to a "Secret". This Secret could be anything that the user wants to hide, like bank information or personal files. A real-world example of this setup would be to lock a file cabinet (a Secret) behind

a door and to hand out identical door keys to the Secret's trustees. The obvious drawback to this scenario is that it gives each trustee an enormous amount of power. They are all free to copy keys and to get to the file cabinet without anyone else's knowledge. If a key were to be lost, an unintended third party would have as much power over the Secret as the original trustees.

In order to overcome these issues, Adi Shamir developed the Secret Sharing Algorithm (SSA). Instead of trusting each user with the key to the Secret, Shamir's algorithm gives each user a part of the key and requires them to combine their keys in order to recover the Secret. This prevents one user from controlling the Secret because it necessitates the cooperation of the users to decrypt the Secret. It also resolves the issue of key loss and duplication. Since access to the Secret requires multiple unique keys, duplication of a single key will not create any advantage. If a user were to lose their key and someone else were to find it, the new person would also be unable to get to the Secret without the help of the other users [Shamir79].

The Secret Sharing Algorithm is based upon the mathematical fact that, for any *n* number of points on the Cartesian plane, there is only one unique polynomial function of degree *n-1* that goes through all *n* points. Because they uniquely define the polynomial they were created from, these *n* points can be combined to recreate the polynomial. These facts allows a Secret value to be hidden as the constant term of a polynomial function. To achieve this, a polynomial function of sufficient degree with random coefficients is created. It will be of the form $a * x^{n-1} + b * x^{n-2} + \cdots + c * x^2 + d * x + 0$, having 0 as its constant term. This allows the Secret value, S, to be converted into a numerical form

and be added as the polynomial's constant term, giving a polynomial of the form $a * x^{n-1} + b * x^{n-2} + \cdots + c * x^2 + d * x + S$. *n* points from this function can then be saved, allowing the Secret value to only be determined via the *n* points after the polynomial is destroyed. Mathematical interpolation is used to recombine the *n* points and recreate the polynomial function containing the Secret value. It is then a trivial task to retrieve the Secret from the polynomial [Shamir79].

Some considerations must be made before building the Secret Sharing polynomial. The minimum number of keys, *k*, needed to decrypt the Secret, and the total number of keys available, *n*, must be determined. These values define the portion of the total number of keys that will need to be collected in order to retrieve the Secret, so they must be picked carefully. *k* should not be so small when compared to *n* that it is trivial to construct the Secret; it should also not be so large when compared to *n* that it is too difficult or is impossible to determine the Secret. Once suitable values for *n* and *k* are selected, the Secret is converted into a number, *S*. A polynomial with random coefficients of degree $k - 1$ is then created with *S* as its constant term. *n* keys are created from the function by plugging *n* input values into the function and saving of each input's corresponding output. After the polynomial function is destroyed, a user must collect at least *k* of the *n* available keys, as stated previously, to determine the original function containing the *S* term [Shamir79].

2.5 Relational Database Terms

During the course of this thesis, database terms like “record” and “value” will be used. These and many other terms’ definitions have changed since they were added to the vernacular; the working set of definitions used in this thesis is as follows.

Attribute: an attribute is analogous to a column in a database table. Each attribute represents a data item whose type can range from numbers and strings to Objects. A set of attributes defines a record [Date04].

Record: a record is a set of data attributes and their associated values that describe an object or entity. Because this definition of a record implies that there is no ordering of the attributes, there is little difference between a record’s definition and the definition of a tuple. Using “record” interchangeably with the term “tuple” is justified for two reasons: most users are more familiar with the term “record,” and the ordering of a record’s attributes will not matter for the purposes of this thesis, so there would be little difference between the two terms [Date04].

Value: a value is the current data stored in a record’s attribute. Because a record’s attributes and their corresponding values are so intertwined – stored values do not make sense unless they are attached to an attribute, and attributes cannot exist without values (null can be considered a value) – these two terms will be used almost interchangeably. Simple contextual clues can be used to tell between the two terms if a distinction is needed to be made [Date04].

Key: a key is a value that distinguishes a record from all other records. Keys are typically broken up into specific groups like primary keys, super keys, and candidate keys, to name a few. In this thesis, the term "key" without a qualifier will typically mean a super key. This means the attribute in question must uniquely distinguish each record, but does not necessarily need to be minimal. Any other types of keys mentioned in this thesis will be properly described in order to set them apart from the rest of the "keys" [Date04].

Table: a table, or database table, is a set of records of the same type. Tables can be described by their arity (the number of columns in the table) and their cardinality (the number of records stored in the table). Tables maintain a uniqueness requirement over their records. That is, no two records stored within the same table can be identical. A set of tables makes up a database [Date04].

View: a view, or database view, is a virtual table built off of preexisting tables. This virtual table consists of a subset of the attributes of other tables, providing the user with a "view" of only a portion of the database. The view can be constructed such that any changes made to the view will propagate to the "base tables" (the tables the view is derived from), allowing the view to function as a real table. Views can be created to only display data relevant to a desired operation, or to only allow access to specific parts of larger tables to users who may not be allowed to see the entire table [Date04].

Base Tables: is the term used to describe the preexisting tables a database view is constructed off of.

Chapter 3

METHODOLOGY

The main pitfalls of the cloud platform today are its accessibility and the variable security provided by Cloud Service Providers (CSPs). When user's data are stored in the cloud, the data are often replicated across multiple servers and data centers to safeguard against one of the servers or data centers failing. Data accessibility is a good thing, but the more accessible data are, the lower their security may become. A form of security must be added to the data that will naturally allow for increased accessibility without incurring any additional security concerns.

The other potential drawback to using the cloud is having to place trust in the Cloud Service Provider. When a user places data in the cloud, the user is no longer in control. The user must trust that the CSP will use the utmost security when handling their data. A user may not always be able to afford a third party that much trust, though – medical records or social security information, no matter how encrypted, must never fall into the wrong hands. Again, a form of security must be applied to the data to give users peace of mind even when they no longer physically own their own data.

The Hoeppner Security Algorithm (HSA) aims to alleviate these security problems. To this end, the Algorithm first splits a user's database records into several "fragments" and

stores them in separate clouds. These fragments can be thought of as attribute-level partitions of a database table – groups of attributes from each table that are placed in separate clouds. The criteria for the attribute splitting is up to the user's discretion. They are able to store each data attribute in a separate cloud, to store groups of attributes that will not prove incriminating together in one cloud, and to break extremely sensitive attributes, such as social security numbers, into sub-attributes before placing them into separate clouds. These clouds should come from separate CSPs for maximum security. By placing the user's data in separate clouds, no one CSP or intruder will be able to view a user's entire enterprise without compromising all of the clouds employed in the Hoeppner Security Algorithm.

This dissemination of data also alleviates the data accessibility problem. The user's data will be replicated within the cloud it is stored, but it is only a portion of the data; all of the data fragments must be combined for the parts to regain their original significance.

The second security measure that the Hoeppner Security Algorithm provides is the creation and storage of fake data. Every time an insertion query is run against the data stored in the cloud, there is a chance that a random amount of fake data will be stored alongside the user's real data. The fake data are split up according to the same user-defined scheme that the real data are, and each fake data fragment is stored alongside the real fragments. The HSA is designed such that a constant number of separate clouds must be contacted by the user in order to distinguish between the real and fake data.

The insertion of a random amount of fake records further diminishes the data accessibility problem because CSPs and intruders are no longer assured that the data they collect are a user's actual data. It also decreases the amount of trust a user must place in the CSP. Each CSP no longer has enough data stored in one place for the data to become compromising to the user. If the data are properly fragmented and stored alongside fake data, no one but the user who placed it in the cloud will be able to bring all of the separate fragments of data together and recognize what is actually stored in the cloud. When the user connects to their database, though, they should only be presented with the real data, and it should appear to them as if all of their information is coming from a single data source. They should have no idea that extra security measures are being utilized behind the scenes.

3.1 Hoeppner Security Algorithm Overview

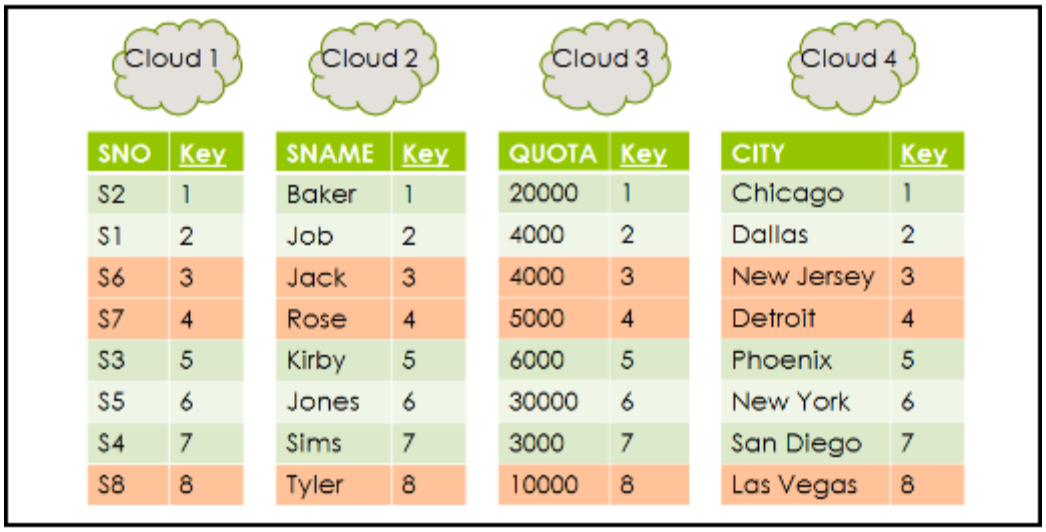
The Hoeppner Security Algorithm relies on partitioning a table's data into several tables stored in separate clouds, presumably hosted by different Cloud Service Providers. Disseminating the data will increase data availability (the clouds will naturally replicate a user's data across multiple host machines for easy access), but maintain data obscurity. For maximum security, data should be partitioned such that no group of attributes that would prove incriminating together is stored in the same cloud. For instance, Social Security Numbers (SSNs) should be stored in an entirely separate cloud from any information that would tie the SSN to its owner's identity. After they are partitioned, each record (including the fake records) is assigned a unique identifier. The identifier is stored

with each of the record's fragments in the separate clouds, allowing the fragments to be recombined later.

This process is depicted in Figures 3.1 and 3.2. The first figure is a database table containing real records (green) and fake records (red). Each record in the table is assigned a unique key value, and then the records are broken up to create the tables in the second figure. The HSA splits up the table based on the user's preference. It can be divided into any number of tables, each of which can be stored in a cloud of their choice. In Figure 3.2, the table is split into four fragments, one for each attribute, and each is stored in a separate cloud. Each attribute fragment partitioned from the same original record shares the same key value. That is why "S2", "Baker", "20000", and "Chicago" all have a key value of 1. They are all fragments of the same record. These key values allow the table's fragments to be easily combined via a JOIN query to construct the original table in Figure 3.1. Data Fragmentation will be discussed further in section 3.2.

SNO	SNAME	QUOTA	CITY
S1	Job	4000	Dallas
S2	Baker	20000	Chicago
S3	Kirby	6000	Phoenix
S4	Sims	3000	San Diego
S5	Jones	30000	New York
S6	Jack	4000	New Jersey
S7	Rose	5000	Detroit
S8	Tyler	10000	Las Vegas

Figure 3.1 - Records before Fragmentation



SNO	Key
S2	1
S1	2
S6	3
S7	4
S3	5
S5	6
S4	7
S8	8

SNAME	Key
Baker	1
Job	2
Jack	3
Rose	4
Kirby	5
Jones	6
Sims	7
Tyler	8

QUOTA	Key
20000	1
4000	2
4000	3
5000	4
6000	5
30000	6
3000	7
10000	8

CITY	Key
Chicago	1
Dallas	2
New Jersey	3
Detroit	4
Phoenix	5
New York	6
San Diego	7
Las Vegas	8

Figure 3.2 - Records after Fragmentation

Now that fragmentation has been addressed, the discussion can turn to how the Hoeppner Security Algorithm uses fragmentation to store and retrieve data. To make the user's experience as seamless as possible, the first action taken when the user connects to their cloud group is to collect only the real records and consolidate them into a single database view for convenience. In order for the consolidation to take place, some information must be stored in each cloud defining which records are real and which ones are false information. The identities of the real records cannot be stored explicitly, though, since that would undermine all of the security measures that the Hoeppner Security Algorithm brings. Instead, the SSA is used to encrypt this information. While the Hoeppner Security Algorithm is running, a list of real records is maintained. When the database is committed, the list is first encoded into an integer called the "Secret value". The encoding process is described in section 3.6. Next, the Secret value is encrypted using the SSA to create input-output pairs. These pairs can later be combined during the decryption process to determine which records are real records, as described in section 2.4. The

generated input-output pairs (not the Secret value itself) are then stored in separate clouds. That way, sufficient information exists to determine which records are real records, though this information is obscured by its fragmented and distributed nature. An attacker would have to connect to a sufficient number of clouds and combine the input-output pairs stored there to determine which records are real and fake, which should be an insurmountable task given that the attacker will not necessarily know how many clouds the information is stored in, which clouds are being used, nor that the data stored in the clouds are fragmented. The more clouds that an attacker has to compromise in order to determine the Secret value, the more robust the security will be. As another added layer of security, the input values from the input-output pairs are never stored in a database, just the output values are. Only the user knows the input values. So not only will an attacker have to compromise a host of cloud databases to have enough information to decode the original table stored there, they will also have to guess or steal the input values that are only ever entered on the user's secure personal computer. These values could be set to randomly rotate during every database commit to further guard the user's data.

Once the input-output pairs from the SSA are stored in the clouds and the database connection is closed, the user must later be able to pull all of that data back together in order to reconstruct the View of real records. A simple process handles the View's reconstruction. First, a connection is made to the cloud tables to collect the output values stored there. These are coupled with the tables' input values to build SSA input-output pairs. Next, the input-output pairs are consolidated via mathematical interpolation to determine the original SSA polynomial from which they were created. Once the

polynomial is recreated, the Secret constant term is extracted from the polynomial function and the equation is discarded. This process is described in section 2.4. The Secret integer term is lastly decoded into the list of real records, which is used to combine the record fragments into a View consisting of only the real records. The decoding process is described in section 3.6. The View is created when the algorithm starts its initial cloud connection process and is dropped when the algorithm completes its execution, which ensures that the real representation of the user's data only exists for as long as the user is interacting with it.

These processes are depicted at a high level in Figure 3.3. Figure 3.3 shows the three fundamental phases taken during the Hoeppner Security Algorithm: the Setup/Retrieval phase, the Database Operations phase, and the Commit/Shutdown phase.

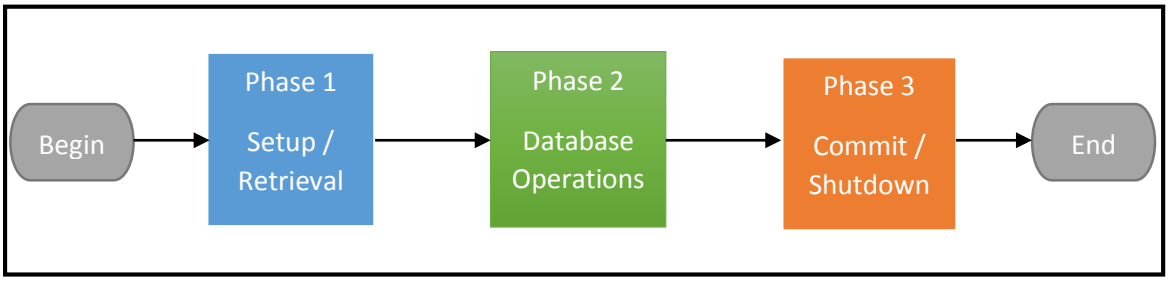


Figure 3.3 –The Hoeppner Security Algorithm at a High Level

Phase 1, the Setup phase, is responsible for creating a local working copy of all of the remote cloud database tables (the tables stored in the clouds) used to construct the database View, retrieving the input-output pairs from the local tables, combining them to determine the Secret value, and constructing the database View of real records from the Secret value.

Phase 2 allows the user to interact with the View as if it was a normal, non-distributed database after the View is constructed. Once a user is ready to commit their changes or to shut down their database connection, the third phase, Commit/Shutdown, is enacted. Phase 3 creates the Secret value from the real records stored in the database, stores the output values created from applying Shamir's SSA to the Secret value into the local table copies, moves the updated local tables to replace their counterparts in the remote cloud databases, and then destroys the database View and any locally copied tables if the database is being shut down.

3.2 Data Fragmentation

As previously mentioned, users have the option of splitting their database into fragments across a group of clouds. Doing so increases security, as information stored in separate databases hosted by separate companies is harder to associate together. There are many fragmentation schemes that a user is able to choose when using the Hoeppner Security Algorithm.

3.2.1 The Attribute Fragmentation Scheme

The first and most simple fragmentation scheme is the "Attribute Fragmentation Scheme". It is the standard fragmentation scheme used by the HSA. Within this scheme, a database's tables are broken up attribute by attribute, each of which is stored in a separate cloud. Figure 3.2 uses this fragmentation scheme. In this Figure, each of the table's four attributes are separated and stored in their own cloud.

3.2.2 The Attribute-Group Fragmentation Scheme

The second scheme is the “Attribute-Group Fragmentation Scheme”. In this scheme, users can select groups of attributes to be stored together inside of the same cloud. Looking back to Figure 3.2, an example of the Attribute-Group Fragmentation Scheme would be to store the “SNO” and “SNAME” attributes in the same table within Cloud 1 rather than storing them in two separate clouds. The Attribute-Group Fragmentation Scheme allows users to decrease the number of clouds necessary for the Hoeppner Security Algorithm, though it is only recommended when storing attributes that will not prove incriminating together. For instance, it would be safe to store a client’s First Name and Last Name attributes in the same cloud, but it would not be safe to store those two in the same cloud as the client’s Social Security Number.

3.2.3 The Sub-Attribute Fragmentation Scheme

The last scheme is the “Sub-Attribute Fragmentation Scheme”, or the “Abbassi Fragmentation Method”. The Sub-Attribute Fragmentation Scheme grants users the most security by allowing them to split attributes into sub-attributes by the number of characters within an attribute, and store each sub-attribute into its own cloud. An example of the Abbassi Fragmentation Method is to fragment a Social Security Number attribute into three sub-attributes: one consisting of the first 3 digits, one containing the next 5 digits, and one consisting of the last digit. These three sets of digits are then stored in separate clouds for added data obscurity. The user is able to determine the number of sub-attributes they would like to create and how the fragmentation is to occur. In the previous

example, the user could have also split the SSN into nine one-digit sub-attributes instead, if they wanted. This gives the user a great amount of flexibility for any dataset.

These three fragmentation schemes can also be combined to give user to the right amount of clouds and security for their particular needs. If they so wished, a user would be able to store a client's First Name and Last Name together in Cloud 1, Address in Cloud 2, Phone Number in Cloud 3, and SSN fragmented into groups of 4 and 5 digits in Clouds 4 and 5. Using these three fragmentation schemes, users are able to customize the security of the cloud to meet their specific needs.

3.3 Phase 1 - Setup/Retrieval

Now that the Hoeppner Algorithm has been discussed at a high level, each of its phases will be examined in turn. When the user starts up the program, the Setup/Retrieval phase (Figure 3.4) should be initiated. The setup procedure first communicates with the group of cloud databases in order to copy into the current working database each remote table that will be used in the View's creation (Phase 1.0). The tables copied from the clouds into the local database will be referred to as "base tables" since they form the basis off of which the View will be defined. Copying the base tables locally will decrease query latency in the long run since the network transmission costs to communicate with all of the clouds are limited to just once at startup and at shutdown. If any of the remote base tables do not exist (i.e. if a base table has never been created in the cloud), the missing tables are created locally in Phase 1.2. Any base tables created this way will initially be empty, but may be filled with records during Phases 2 and 3.

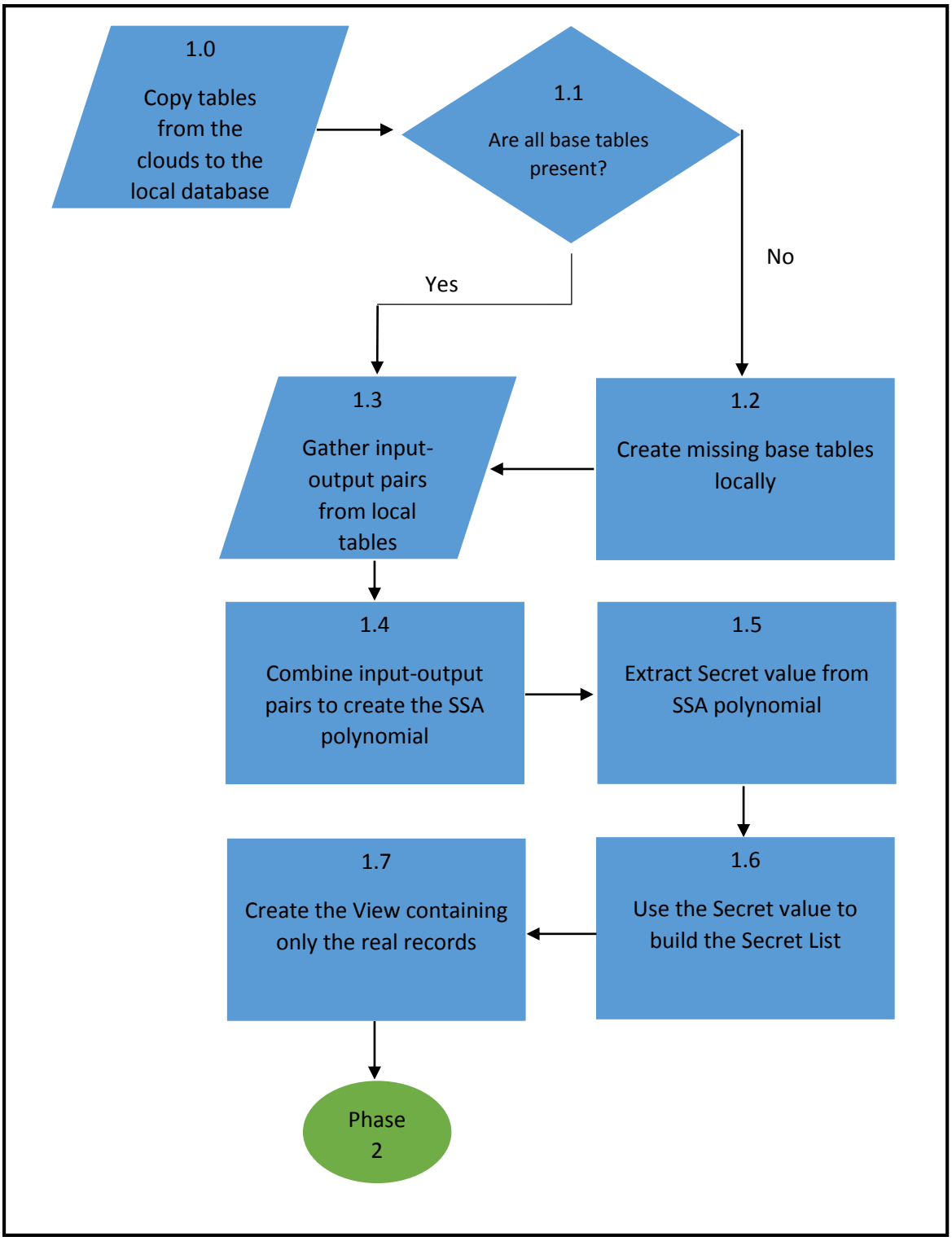


Figure 3.4 – Hoeppner Algorithm's Setup/Retrieval Phase Expanded

The base tables used in the HSA were each assigned an input value for the SSA polynomial. Each base table's input value and stored output value can therefore be combined to create SSA input-output pairs. In Phases 1.3 and 1.4, a sufficient number of these input-output pairs are collected and combined via mathematical interpolation to recreate the SSA polynomial from which they were generated. The constant term of the SSA polynomial, the Secret value, is then retrieved in Phase 1.5, as described in section 2.4. This Secret value contains encoded information about which records are real and which are fake, which will later be used in the creation of the View. Phase 1.6, the interpolation process, is depicted in Figure 3.5, where input-output pairs are being pulled from four clouds in order to recreate the SSA polynomial and determine the Secret value.

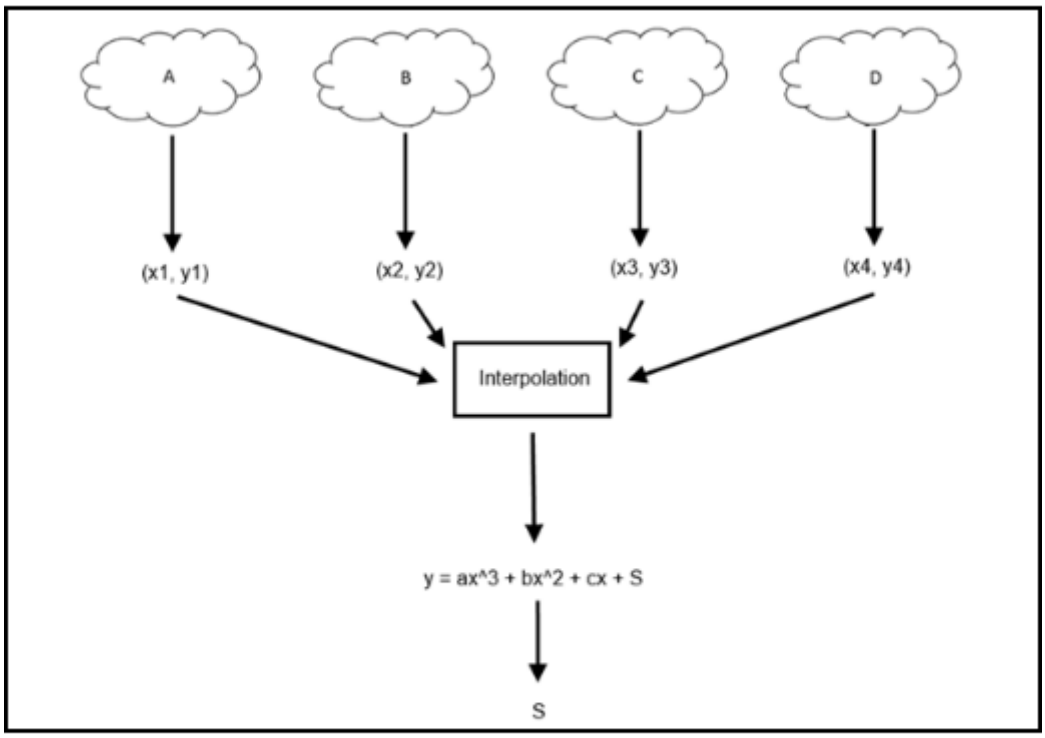


Figure 3.5 - Hoeppner Methodology's Interpolation Procedure

Phase 1.7 is the next step in the program's initialization. It is during this phase that the database View is created for the user. The list of real records determined from the Secret value is used to build a CREATE VIEW query against all of the locally copied base tables. Only the real records are collected during this process. The process to decode the Secret value into the list of real records used to create the View is described in section 3.6. Once the View is created, the user is able to interact with it as if it is just another table in the database. The only difference is that View-altering queries must be broken down to affect the View's base tables instead. For example, every time a record is inserted into the View, the INSERT query must be transformed into several queries, one to insert a fragment of that record into each of the View's base tables. Any changes made to the base tables would then be propagated to the View. The query transformation process takes place in Phase 2.

3.4 Phase 2 – Database Operations

Once the View is set up, the user is able to perform typical Database Operations on it. This procedure is defined in Figure 3.6's flowchart.

The user first starts in Phase 2.0 by issuing a query to the DBMS as they normally would in a non-distributed environment. If it is a commit or a quit command, Phase 3 is initiated. Otherwise, the rest of Phase 2 is processed.

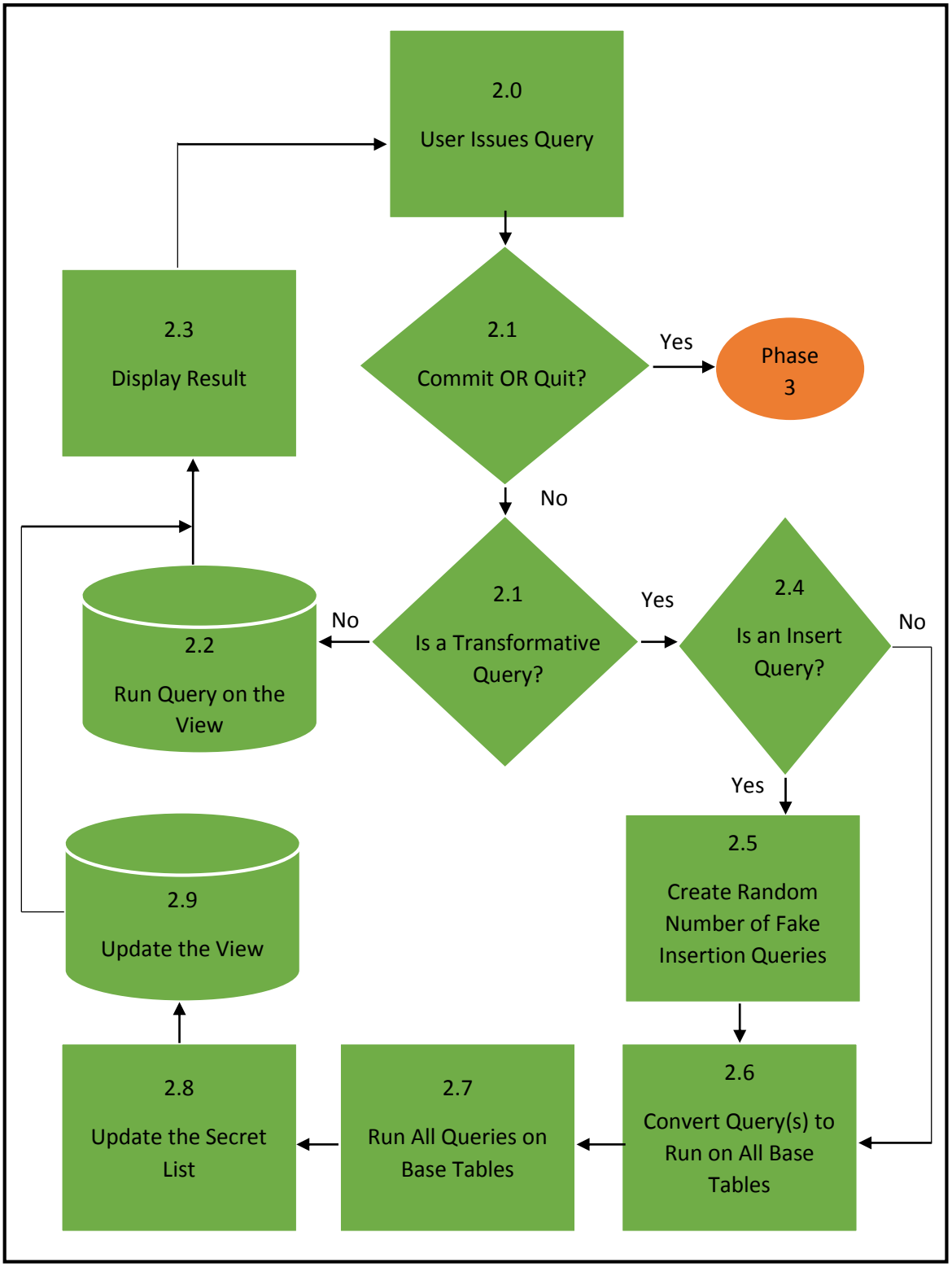


Figure 3.6 – Hoeppner Algorithm's Database Operations Phase

If the user's query is not transformative in nature (meaning it will not change the View's base tables), the query is run against the View in Phase 2.2 and the result is displayed in 2.3. If, however, the query will change the base tables, the query must be analyzed further. In Phase 2.4 of Figure 3.6, it is determined whether the query is an INSERT query. If it is, a random number of fake INSERT queries are created to be run alongside the real query. Each record to be inserted, real or fake, is assigned a unique key identifier that is stored alongside all of the record's fragmented pieces in the View's base tables. This allows the data fragments to be recombined into a whole record within the View later.

In Phase 2.6, the user's original query is broken down into equivalent queries to be run against the View's base tables. For insertion queries, this is where record fragmentation takes place. The query is analyzed to determine which of the base tables it will affect, and the original query's VALUES clause is broken down to build insertion queries for each of those tables. This process is slightly different for modification and deletion queries. In this case, the query's WHERE clause is analyzed to determine the key values of the affected records. Then, a delete or modification query is created for each of the View's base tables. These queries have a WHERE clause specifying the affected keys' values so only the relevant records will be changed. Once all of the queries are created and run on the underlying base tables, the list of real records and the View are updated to reflect the changes to its base tables (Phases 2.8 and 2.9). Lastly, the query's results are displayed to the user.

All of the fake insertion queries generated in Phase 2.5 are also broken down in Phase 2.6 so that they can be applied to the relevant base tables. To add another level of security to the HSA, each of the fake INSERT queries generated during Phase 2.6 has a random chance of being canceled. This allows for a different number of fake records to be stored in each base table. For instance, if a fake record is inserted into a View with four base tables, four fake INSERT queries will be generated. If three of the four queries succeed, one of the base tables will have less records than the others. Adding randomness to the insertion process disassociates the clouds' sizes from each other and ensures that a constant proportion between real and fake records cannot be discovered.

3.5 Phase 3 – Commit/Shutdown

When the database is to be saved and or closed, the Commit/Shutdown procedure is used to perform the reverse of the Setup procedure (Figure 3.7).

The Commit/Shutdown phase first takes the list of real records in Phase 3.0 and encodes it into a "Secret" integer value. The encoding process is described in section 3.6. The Secret value is then used as the constant term in a SSA polynomial with random coefficients. Each base table's input value is used as an input to the SSA polynomial, creating a set of input-output pairs in Phase 3.2. The polynomial is then destroyed. Lastly, the output values from the polynomial are stored in the local base tables for retrieval during the setup phase.

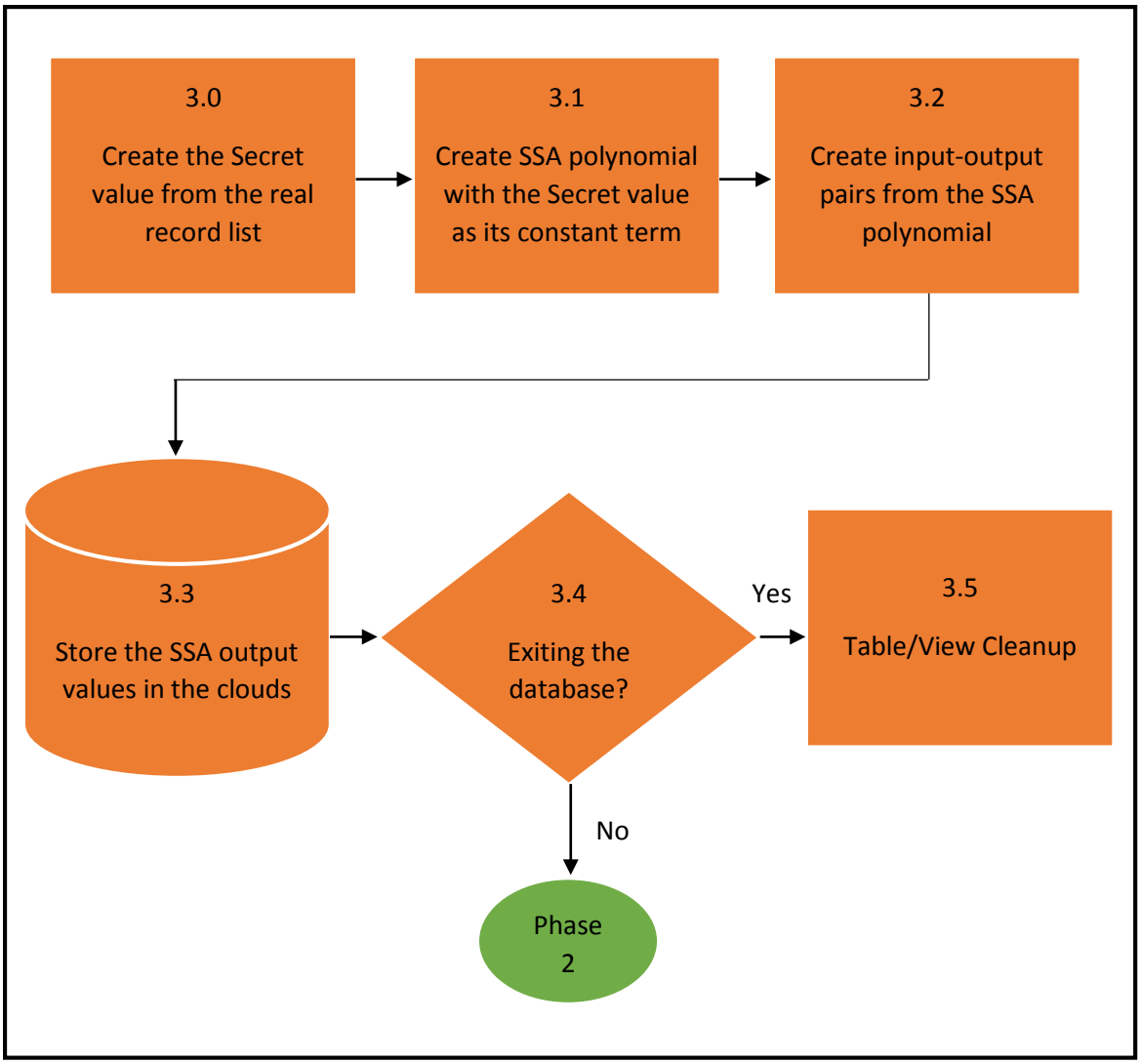


Figure 3.7 – The Hoeppner Algorithm's Commit/Shutdown Phase Expanded

Once all of the output values are saved, the cleanup phase (Phase 3.5) is initiated if the database connection is to be shut down. At this time, the View is deleted and all of the local base tables are pushed out into the clouds, replacing any old tables they were pulled from during the Setup/Retrieval Phase. If a base table was created locally and does not have a counterpart in the cloud, it is copied into the cloud to start the cycle of having remote cloud tables to pull from and replace. After the base tables have successfully been moved to the clouds, their local copies are also destroyed. These deletions ensure that the

View and the local base tables only exist for as long as the program is running. As previously described, the procedure of generating the input-output pairs and storing them in the cloud tables is diagrammed below in Figure 3.8.

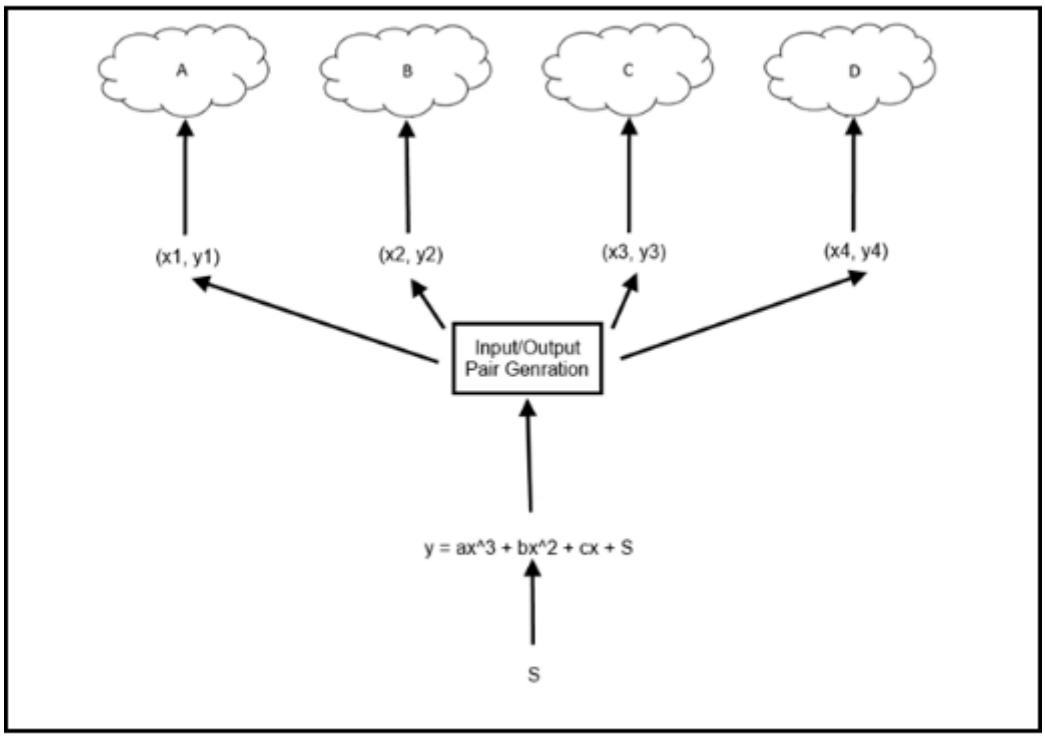


Figure 3.8 - Hoeppner Methodology's Secret Commiting Procedure

3.6 Secret Value Explained

The biggest question left unanswered by the high level overview is how the list of real records is encoded to create the Secret value. The Secret value is a binary number based off of the unique key value that every record, both real and fake, is assigned when it is inserted into the database. The binary Secret starts off with no value when the Hoeppner Security Algorithm is initialized. Each time a record is inserted into the View, its unique

key value is added to the list of real records if it was a real record. When the time comes to build the Secret value, the list of real records is assessed. For each unique key value assigned, one bit is appended to the left-most position of the Secret value. The bit is a 0 if the key value does not appear in the list (i.e. the record is fake), and a 1 if the key value is in the list (i.e. the record is real). Since the unique key value assigned to each record is based on the order that the records are inserted in, a correlation is able to be made between a record's key value and its representative bit in the Secret value. For example, the first record inserted into the database will have a key value of 1, and the 1^{st} bit from the right in the Secret value will correspond to whether that record is a real or fake one. This correspondence makes selecting the real records from the fake ones a trivial task – for each n^{th} bit of the Secret binary number from the right to the left: if the bit is a 1, add the record with a key value of n to the list of real records; otherwise do nothing.

While having a binary Secret value makes separating the real records from the fake ones a simple task, it complicates the process of evaluating the SSA polynomial function from which the input-output keys are derived. Large binary numbers have many more digits than their base-10 counterparts, which makes performing mathematical operations on them difficult if they were just assumed to be in base-10 (e.g. adding 512 to 1101001101 takes up more memory than adding 512 to 845). To simplify the polynomial's evaluation, the binary Secret is converted into its base-10 representation when it is added as the SSA polynomial's constant term. Having all of the polynomial's terms represented in base-10 also adds another level of obscurity to the HSA. If an intruder is able to recreate the polynomial function that contains the Secret value, they may not know that the Secret is actually a binary number represented in base-10.

3.7 Hoeppner Security Algorithm Example

This section gives a walkthrough of how the Hoeppner Security Algorithm performs its three Phases on a sample dataset. In these examples, it is assumed that all operations are taking place in the distributed cloud tables shown in Figure 3.2, and that the user is already connected and presented with the database View named “view”.

3.7.1 Insertion Example

The first step a user would take to insert a record is to issue a query similar to:

```
INSERT INTO view VALUES('S9', 'Megan', 7500, 'Tampa');
```

Because this is an insert query, some random number of fake insert queries are also generated and issued. All of these queries, both real and fake, are assigned a unique key value and are broken down based on the structure of the View’s base tables. In this case, Figure 3.2 shows that the View is composed of four base tables – one for each attribute. This means that the user’s query will be transformed into four queries:

```
INSERT INTO view VALUES('S9', 9);
INSERT INTO view VALUES('Megan', 9);
INSERT INTO view VALUES(7500, 9);
INSERT INTO view VALUES('Tampa', 9);
```

In this case, ‘9’ is the query’s unique key value. The transformation process is also carried out on the fake queries. Once all of the queries have been modified to affect the View’s base tables, the four real queries are submitted. The fake ones, though, are each

given a random chance of being submitted. This, as previously stated, helps to break any association between the amount of data and the number of real records in the database.

At this point, all of the user's data and the randomly-selected fake data are fragmented into the four local tables, which will later be stored in four separate clouds. This brings us to Phase 2.8 of Figure 3.6. The Secret List of real records must be updated. Because the user's record was assigned the unique key value '9', '9' is added to the Secret List to join the green records from Figure 3.2. With the addition of the '9', the List's contents are: '1', '2', '5', '6', '7', '9'. Once the Secret List is updated, a CREATE VIEW query is issued to update the View:

```
CREATE OR REPLACE VIEW view AS SELECT sno, sname,
quota, city FROM cloud1, cloud2, cloud3, cloud4 where
cloud1.key = cloud2.key AND cloud2.key = cloud3.key
AND cloud3.key = cloud4.key AND (cloud1.key = 1 OR
cloud1.key = 2 OR cloud1.key = 5 OR cloud1.key = 6 OR
cloud1.key = 7 OR cloud1.key = 9);
```

This query joins all of the record fragments based on matching unique key values, and selects only the ones with real keys. Once the View is updated, it can be presented to the user.

3.7.2 Shutdown Example

Now, let's assume that the user wishes to close their database connection. They would start by issuing a "quit" command, which starts the Shutdown Phase. The first major part of this phase is to encrypt the Secret List. To this end, a binary string is created from the Secret List, where each bit from the right to the left represents a record's unique key

value (the first bit from the right represents the key value ‘1’, the second bit represents the key value ‘2’, etc.), and is a ‘1’ bit if and only if the record with that unique key value is a real record. For this current example, the binary string would be “101110011” because the real records have key values of ‘9’, ‘7’, ‘6’, ‘5’, ‘2’, and ‘1’. The binary string is next converted into a base-10 number, 371 in this case. This base-10 number is known as the Secret value

In Phase 3.1 of Figure 3.7, the Secret value is passed through the SSA for encryption. The number of input-output pairs to be created from the SSA must be determined, clouds must be selected to store the output values from the pairs, and a minimum number of input-output pairs necessary to run the SSA must be set. All of these decisions are up to the user, so in this example it is assumed that all four of the clouds will store output values, and that all four values will be necessary for the SSA. Since four input-output pairs will be needed to decrypt the Secret value, the SSA will require a third-degree polynomial with random coefficients containing the Secret value as its constant term to be built: $40x^3 + 3.4x^2 - 1.12x + 371$. Each of the four clouds now apply their input value to this SSA polynomial. For this example, let’s assume that the first cloud has an input value of 1.1, the second has an input value of 2.0, the third has an input value of 0.3, and the fourth cloud has 5 as its input. These four values, when applied to the polynomial, create the input-output pairs (1.1, 427.122), (2.0, 702.36), (0.3, 372.05), and (5, 5450.4). The output values from these pairs are then stored in their respective clouds (427.122 to cloud1, 702.36 to cloud2, 372.05 to cloud3, and 5450.4 to cloud4), and the Table/View Cleanup phase is initiated. The Table/View Cleanup phase copies all of the local tables into the clouds and then removes both the local tables and the View.

Dropping all of the tables in the local database ensures that all of the user's information only exists together for as long as the user is interacting with it, and no longer.

3.7.3 Startup Example

The final example in this chapter explains the Startup Phase. At the start of this phase, all four of the tables copied into the clouds at the end of the Shutdown Phase are copied locally. No new base tables must be created at this point because all four base tables were successfully copied from the clouds. That means it is time to start the crux of Phase 1: the decryption of the Secret value and the building of the Secret List.

To undo the SSA's work and decrypt the Secret value, the input-output pairs generated during the Shutdown Phase must be collected. To this end, the output values are pulled from the four local tables and are married with each table's input value, giving us the input-output values of (1.1, 427.122), (2.0, 702.36), (0.3, 372.05), and (5, 5450.4). These four points uniquely identify the third-degree polynomial from which they were generated, so they are able to be recombined via mathematical interpolation to recreate the original SSA polynomial created in Phase 3. There are many ways to perform this interpolation. One of the most publically well-known interpolation methods is to use a system of equations to solve for the polynomial's missing coefficients, so it is the approach that will be presented here. Because the four input-output values were generated from a polynomial of degree 3, the desired solution will take the form $y = a * x^3 + b * x^2 + c * x + S$, where S is the Secret value. Next, the four input-output pairs are plugged into this equation to create four equations with four unknown variables:

$427.122 = 1.331 * a + 1.21 * b + 1.1 * c + S$,

$702.36 = 8 * a + 4 * b + 2 * c + S$,

$372.05 = 0.027 * a + .09 * b + 0.3 * c + S$, and

$5450.4 = 125 * a + 25 * b + 5 * c + S$.

Through a process of substitution and elimination, one is able to determine that $a = 40$, $b = 3.4$, $c = 1.12$, and $S = 371$, which is what was expected.

Now that the Secret value, S, is known, it can be transformed back into the Secret List on which the View is based. The first step in this process is to convert it from a base-10 number into a base-2 string. In this example, 371 converts into the binary string "101110011". To create the Secret List, the binary string is traversed from the rightmost bit (bit #1) to the leftmost bit (bit #9), and each bit is analyzed. If the current bit is a '1', its bit number is added to the Secret List; if the bit is a '0', it is not added. Since bits 1, 2, 5, 6, 7, and 9 are '1's, the Secret List will contain the values '1', '2', '5', '6', '7', and '9'.

The next step in the Startup Phase is to create the View. This is a relatively simple process now that the Secret List is populated. Just as in section 3.7.1's insertion example, a CREATE VIEW query is issued to build the View:

```
CREATE OR REPLACE VIEW view AS SELECT sno, sname,
quota, city FROM cloud1, cloud2, cloud3, cloud4 where
cloud1.key = cloud2.key AND cloud2.key = cloud3.key
AND cloud3.key = cloud4.key AND (cloud1.key = 1 OR
cloud1.key = 2 OR cloud1.key = 5 OR cloud1.key = 6 OR
cloud1.key = 7 OR cloud1.key = 9);
```

All of the View's base tables and attributes are listed in this query, and all of the real records' key values are specified in the OR clauses at the end of the query. Once the View is built, the user is able to interact with it as if it is a non-distributed table.

Chapter 4 will now discuss the HSA's implementation. An analysis of this specific implementation's performance will follow in Chapter 5.

Chapter 4

IMPLEMENTATION

This chapter examine an implementation of the Hoeppner Security Algorithm in the Java programming language. This implementation, henceforth known as the "Po1 Program" or "Program" for short, generally follows the description of the Hoeppner Security Algorithm from Chapter 3 with some language-specific changes and considerations. Pseudocode snippets taken from the phases of the Po1 Program's execution (as outlined in Chapter 3) will be analyzed in this chapter.

4.1 Phase 1 – Setup/Retrieval Implementation

Similar to Figure 3.1's high-level flowchart in Chapter 3, the implementation for the Hoeppner Security Algorithm is broken up into 3 main phases: the Setup/Retrieval Phase, the Database Operations Phase, and the Commit/Shutdown Phase. The first high-level phase, Setup/Retrieval, is described in this section. The entire process is outlined in Figure 4.1.

```
setupPhase()
   1. Connect to the local database
   2. Pull all of the distributed tables from The Cloud to the local database
   3. Combine any fractional tables into composite tables
   4. Perform secretSetup()
   5. Build the View from the Real Records List
end setupPhase
```

Figure 4.1 - Setup/Retrieval Phase Pseudocode

When the Program is executed, its first task is to create a database Engine object. This object is given a database's connection information, which is used to establish a link to the database server using the JDBC (Java Database Connectivity) API. All queries to be executed in that database are passed to the Engine object, who in turn runs them on the server and returns the query's results. In this case, the Engine created will be used to communicate with the trusted local database.

Once the Engine is set up, all of the remote cloud tables must be copied locally to improve querying time and security, as in Phase 1.0 of Figure 3.4. Two methods exist to push and pull data to and from the clouds. The first, tablePushPull, requires Oracle's SQL*Plus program to be installed on the local computer. All of the Po1 Program's implementation and testing was carried out using SQL*Plus databases, so optimizations were created for if SQL*Plus is present on the computer running the Program. SQL*Plus contains a non-standard COPY query specifically designed to move tables from one Oracle database to another. Because this is non-standard, though, Java and the JDBC API do not support it. If, however, the user does not have SQL*Plus, manualTablePushPull is called instead. This function utilizes the JDBC API to manually select all of the records

from the cloud databases, create local copies of the remote tables in the local database, and insert all of the remote records into the local tables. This is naturally less efficient as it uses Java as an intermediary language instead of doing all of the work and query optimizations directly in SQL, but it is the simplest way to copy all of the tables between separate cloud databases. Database links could be created to facilitate moving the data through SQL without needing SQL*Plus, but connecting the cloud databases to the secure local one would somewhat defy the point of fragmenting data across multiple clouds. A user's tables should always remain as separate as possible so that outsiders cannot draw logical connections between them.

The next process in the setupPhase pseudocode is step 3, which builds composite tables from any fragmented tables that the user listed in the Program configuration file. This process is what allows users to utilize the Abbassi Fragmentation Method because it combines all of the fragmented records into whole records that can later be displayed in the View (see section 3.2.3). The buildTableProductions method performs this task. The creation process is guided by the table production rules defined by the user. This process is outlined in Figure 4.2.

```
buildTableProductions()
   for each table production rule
      1. create a new table
      2. combine the table fragments into the new table
   end for each
end buildTableProductions
```

Figure 4.2 - buildTableProductions Pseudocode

Every production rule that the user specifies in the configuration file is first broken up to determine the composite table's name and all of the fragmented tables that will combine to make it. The method next constructs the composite table in the local database. Then, it builds an INSERT INTO query, which will concatenate all of the fractional table's attributes together based on their unique key values and insert the resulting records into the newly-created table. Recall that all of a record's fragments share the same unique key value, so this process simply puts them all back together in the same way that they were initially broken apart. Once the full query is constructed, the Engine object executes it to populate the new table, and the next table production rule is analyzed.

Once all of the required tables are constructed, secretSetup, the implementation of Phases 1.3 through 1.6, is executed. It analyzes the input-output pair data stored in the local tables to recreate the Secret value.

```
secretSetup()
   1. gather all of the output keys from local tables
   2. form input-output pairs from the output keys
   3. interpolate the input-output pairs to recreate the polynomial
   4. extract the Secret from the polynomial
   5. evaluate the Secret
end secretSetup
```

Figure 4.3 - secretSetup Pseudocode

The routine starts by connecting to all of the tables containing output key values and collecting them. These values are paired with their table's input value to form input-output pairs. Given a sufficient amount of these pairs, one is able to determine the

original SSA Polynomial from which the pairs came. This is based on the mathematical fact that any polynomial of degree *n* can be uniquely determined from *n + 1* points from the polynomial. To determine the Polynomial, the key pairs are next passed to the interpolate method. This method uses linear algebra to perform the interpolation process described in section 3.7.3. During this process, an input value matrix, 'A', is constructed from the tables' input values, a variable vector, 'x', is built to store the unknown coefficients, and a solution vector, 'b', is created from the tables' output values. These matrices model the relationship $b = A * x$. An LUP decomposition (an LU decomposition with a permutation matrix, P) is then performed on the 'A' matrix, and forward and backward substitutions are utilized to solve for the original coefficients used in the Secret Sharing polynomial. The constant term of the polynomial is extracted in step 4 of the secretSetup method, and becomes the Secret value [Cormen01, pages 742-754].

secretSetup calls the evaluateSecret method in step 5, passing it the Secret value. Just as section 3.6 described, it converts the Secret from a base-10 number into a binary string and adds a record to the list of real records for every '1' bit in the string. The secretList variable is then able to be used by the rest of the class' methods. The pseudocode for this method is as follows.

```
evaluateSecret()
   1. create a binary string out of the Secret value
   for each digit in the binary string
      if the digit is a '1'
         2. add the digit's index to the Real Records List
      end if
   end for each
end evaluateSecret
```

Figure 4.4 - evaluateSecret Pseudocode

This method relies on the fact that each bit in the binary string created in step 1 corresponds to a record's key value – the i^{th} bit from the right end of the string represents the record containing a key value of i – and the bit's value represents whether the record is real or not – a '1' corresponds to 'real' and a '0' to 'fake'. This knowledge can be used to easily transform the binary string into a list of the real records to be placed into the View.

Once the list of real records is known, execution returns to the startup method, who submits a CREATE VIEW query to the Engine. This builds a view only consisting of real records, as described in Chapter 3's Phase 1.7.

4.2 Phase 2 – Database Operations Implementation

Once the View is created, the Program enters Phase 2 – Database Operations. During this phase, the user is able to issue queries against the View as if it were a normal table in a non-distributed database. Submitted queries are split up into two broad categories: ones that will change the View's underlying base tables, and ones that will not. Queries of the

second variety, select statements for instance, are carried out directly against the View. On the other hand, queries that will change the base tables (transformative queries) must first be broken down attribute-wise into alternative queries that can be applied to each of the tables that make up the View. The View is then reconstructed from the updated base tables, and the result is displayed for the user.

The method to handle an INSERT query, Figure 4.5's viewInsert, first assigns the record in question a unique key value and separates the query's attributes into *n* groups – one group for each of the View's base tables. These groups of attributes are used to make *n* insert statements– one for each base tables. These *n* insert queries all share the same unique key value so they can later be joined back together in the View. If the record is a real one, its *n* insert statements are submitted to the Engine immediately, and its key value is added to the list of real records, the Secret List. If it is fake, each of its *n* insert queries have an independent random chance of being submitted. This adds some variability in the amount of fake data that are stored across the clouds. Once all of the queries are processed, the View is updated for the user.

```
viewInsert()
    for each of the View's base tables
        collect all of the table's attributes
        build an insert query from the attributes
        if the query is fake
            have a random chance of executing the query
        else
            execute the query
            add the record's key value to the Real Records List
        end if
        update the view
    end for each
end viewInsert
```

Figure 4.5 - Pseudocode for Inserting a Record into the View

The viewInsert method is called for the insertion of both real and fake records. When an INSERT query is issued, a random number of fake records are fabricated and are passed to viewInsert. This is a different random chance of insertion from the one in Figure 4.5. The former random chance determines the number of fake records that will be attempted to be inserted, and the latter determines on which base tables the insertion will take place. Having two levels of randomness helps to disguise the user's metadata footprint because it stops the ratio of real to fake records from being determined, especially since each fragment of a fake record has a random chance of being inserted.

4.3 Phase 3 – Commit/Shutdown Implementation

The last database phase, the Commit/Shutdown Phase, is triggered any time the database is saved or the program is exited. The commit procedure, which encompasses Phases 3.0 through 3.3, is called regardless of whether the database connection is being closed or

not. It is a very simple method – it calls the imposeSecret method to create and store the Secret Sharing polynomial's output values in the local tables before saving any changes to the local database.

The imposeSecret method, like the secretSetup method, calls 3 other methods to do its work. The first, createSecret, creates the Secret value from the list of real records. It initializes a binary string to the empty string and then sorts the list of real records. Next, starting from the largest real key value and going to the smallest, a '1' is appended to the string for every real record, and the number of zero's between the current real record's number and the next real record's number is computed and appended to the string. For example, if '5' and '2' were in the list of real records, a '1' would be appended to the string for the '5' record, and then two '0's would be appended to the string because records '4' and '3' are fake and do not appear in the list. This process is continued until all of the records are accounted for in the binary string. To make this process work, a temporary '0' is added to the Real Records List so that the proper number of trailing 0's will be printed. The temporary '0' is taken out of the list at the end of the algorithm so that a '1' is not added to the binary string for it. The loop that handles this logic is as follows.

```
for each value in Real Records List
    prepend a '1' to the binary string
    calculate the distance between this value and the next value in the List
    while distance > 0
        prepend a '0' to the binary string
        decrement the distance
    end while
end for
```

Figure 4.6 Pseudocode for Building the Binary Secret String

Once the binary string is created, it is parsed into a base-10 integer. This Secret integer is then passed into the createKeys function. This function creates the list of random SSA Polynomial coefficients, and uses Horner's method to evaluate the Polynomial at each table's input value. Horner's method states that any polynomial of the form $a * x^n + b * x^{n-1} + c * x^{n-2} + \cdots + e * x + f$ can be expressed as $(((a * x + b) * x + c) * x \ldots) * x + e) * x + f$ [Lafore03]. This expression allows the polynomial to be easily expanded with one loop – at each iteration, the running sum is multiplied by 'x' (the table's input value), and the next randomly-generated coefficient in the list is added to it. This process continues until the n^{th} coefficient has to be added. Since the Secret value is what we are trying to encode in this SSA Polynomial, it is added as the final coefficient. The evaluation process is carried out for each of the tables' input values. Once all of the output values are generated, each is stored in its table.

If the database is to be shut down and disconnected from, the breakdown method is executed. Once all of the commit method's previously-described actions are completed, the breakdown method uses the Abbassi Fragmentation Method to break down any

composite tables into table fragments. These fragmented tables are then able to be pushed into separate clouds. This fragmentation process is accomplished by using SQL's substring function to split each record's attributes into strings of the desired length and insert the results into the fractional tables. Once all of the fractional tables are built, the manualTablePushPull or tablePushPull method is called, just as it was during the Setup phase. The only difference is that the method is given the string "push" as an argument. This tells it to perform the same actions as in the Setup phase, but in reverse. This time, all data are to be saved in the cloud databases instead of the local one. Once all of the tables are successfully copied into the clouds, the View and all of the local tables are deleted, the database's recycle bin is purged, and the connection to the local database is closed.

4.4 GUI Implementation

The Po1 Program was implemented both as a command line console application and as a windowed Graphical User Interface (GUI). This allows users to run the Program in the environment in which they are most comfortable. The GUI version contains all of the features that the console version does, and includes a Configuration Wizard to guide users through the creation of the configuration file that the Program uses for initialization. The configuration file contains thorough internal documentation, which allows console users the same ease of use when setting up their initial database configurations. This section explores the layout and features of the GUI Program and the Configuration Wizard.

When executed, the GUI program presents the user with the window shown in Figure 4.7. This GUI has three main areas – the upper text area where all of the query results and diagnostic messages are printed to the user, the lower text box where the user is able to submit queries to their database, and the bottom area where the "Execute Query" button and status indicator are housed. The status indicator displays database connectivity information: on startup it will report that the Program is not connected to the database, but it will let the user know they are connected and are able to issue queries once a connection is established.

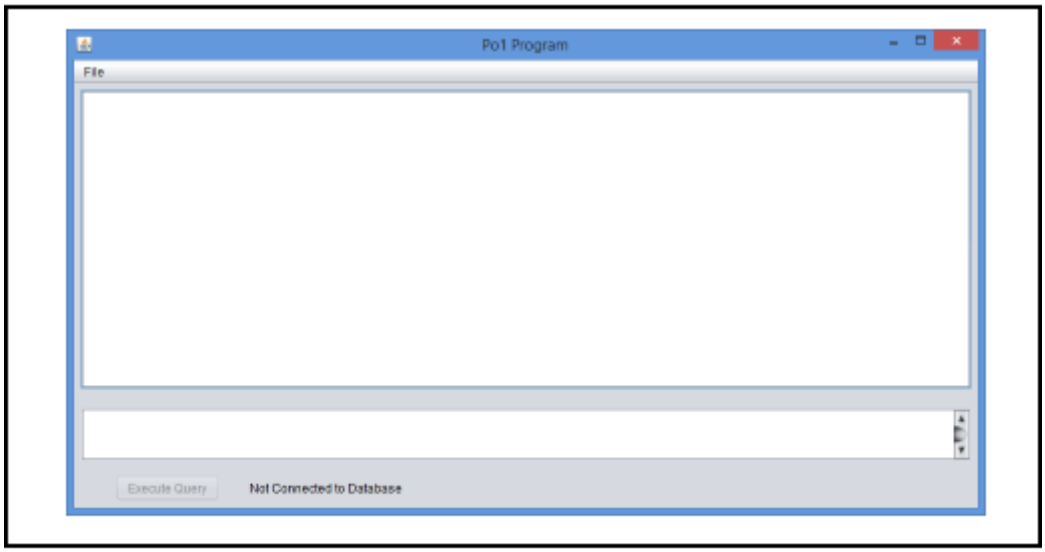


Figure 4.7 The Po1 Program GUI Startup Screen

From the "File" menu option, the user is able to launch the Configuration Wizard, connect to a database by selecting a configuration file, clear the output text area, or disconnect and close the program. The latter three options are self-explanatory, but the Configuration Wizard option merits more discussion. When launched, the Wizard's first

menu asks the user whether they would like to create a new configuration, update a preexisting configuration file, or import a table into their database. These options are seen in Figure 4.8.

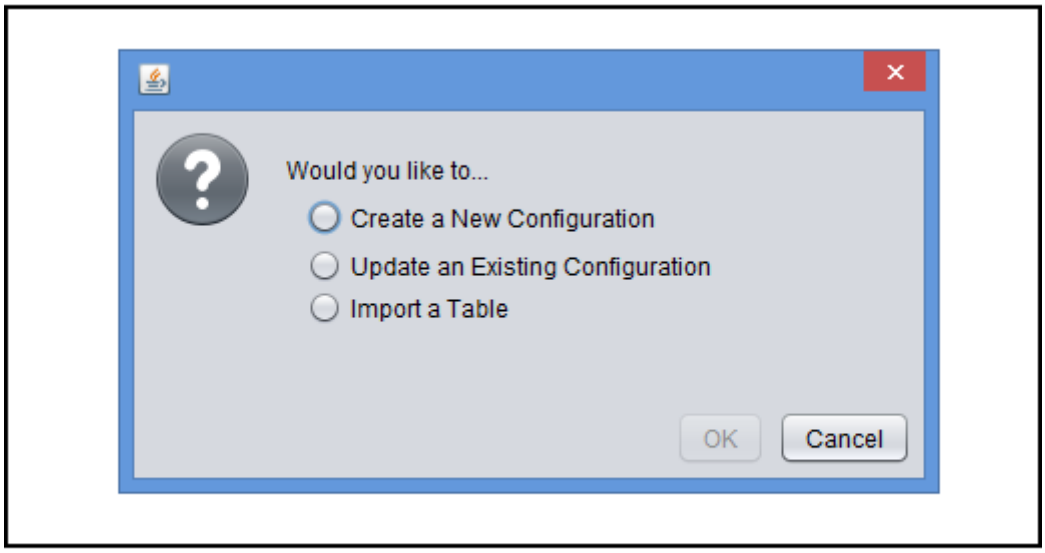


Figure 4.8 The Configuration Wizard's First Menu

If the user chooses to create a new configuration file, they will be guided through a series of windows prompting them for all of the information needed for the initial setup of their distributed cloud database. An example of one of the Wizard's prompts is included in Figure 4.9. In this window, the user is asked to supply information needed to connect to their secure local database. Example text is presented to the user in grey. Once all of the Wizard's questions are answered, the user is prompted to save and view their newly-created configuration file.

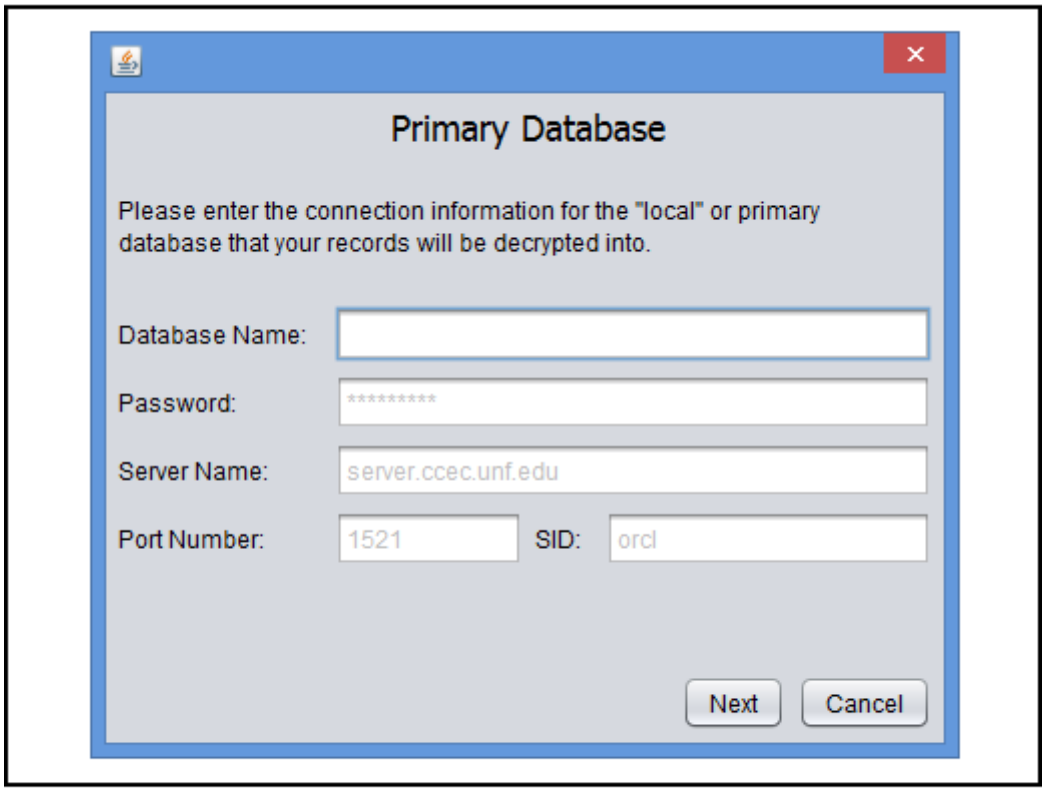


Figure 4.9 Example Configuration Wizard Menu

The other two options in Figure 4.8 provide the user with similar questions. If the user chooses to update an existing configuration file, they are first asked to browse to a configuration file. Their configuration file is then read into the Program, and is used to pre-populate many of the fields in the Wizard. Pre-population allows the user to go through the Wizard and change any responses they had previously made.

Lastly, the user has the option to import an existing table into the View that their configuration file describes. If the user selects the third option in Figure 4.8, they will be presented with one additional window in the Wizard. Similar to the window presented in Figure 4.9, the additional last prompt asks the user to specify the connection information

and name of the table to be imported. The connection information is stored in the configuration file, and is used to import the desired table into the View the next time that particular configuration file is utilized.

Next, in Chapter 5, the Hoeppner Security Algorithm's performance is evaluated against the program proposed in the Alzain paper to determine the Algorithm's merits.

Chapter 5

RESULTS

This chapter describes an implementation of the Alzain Program, which will serve as a comparative methodology for the Po1 Program (the Java implementation of the Hoeppner Security Algorithm). The specifications for the platforms and clouds that the Alzain and Po1 programs were run on are outlined, and performance data for both programs are presented. Lastly, security considerations are discussed.

5.1 Platform and Cloud Specifications

Two servers were used during the testing phase. The first, “Osprey,” was used to run the Po1 and Alzain programs. The second server, “Taurus,” hosted the cloud database accounts. Osprey is a Red Hat Enterprise Linux Server release 6.7 (Santiago) virtual machine. It has 8 GB of RAM and two Intel Xeon E5-2670 vCPU’s, each running at 2.60 GHz. Taurus is a Centos 5.1 virtual machine server running Oracle 10. It contains two CPS’s and 4 GB of RAM. Both virtual machines are multi-user computers, and are hosted on the same physical servers as other virtual machines

5.2 The Alzain Program's Implementation

The program proposed in "MCDB: Using Multi-clouds to Ensure Security in Cloud Computing," hereafter referred to as "the Alzain Program," was chosen as a comparative methodology to the Po1 Program [Alzain11]. This choice was made for several reasons. First, at their core, both are cloud security algorithms seeking to add a level of security to potentially vulnerable databases. Secondly, the Alzain Program was selected because it provided some of the inspiration for the creation of the Po1 Program. It seems only fitting that the two should be compared to determine whether the Po1 Program has been able to add value to the algorithm that helped it come into being. Thirdly, both algorithms use Shamir's Secret Sharing Algorithm (SSA) for their encryptions. Each program uses the SSA for a different purpose – the Alzain Program uses it to encrypt each record while the Po1 Program uses it to encrypt which records are real – so it will be interesting to see how the SSA's usage affects performance [Alzain11].

Some liberties had to be taken when designing and implementing the Alzain Program to allow it to be more meaningfully compared to the Po1 Program. The first change was implementing the Alzain Program in Java. The programming language an algorithm is implemented in can substantially affect its runtime. The Alzain Program was proposed to be run on a web-based HTTP server that would connect to its cloud servers on the backend and allow users to make standard HTTP requests for their data on the front end [Alzain11]. It was ported to Java in order to decrease the amount of Web data requests and their inherent latencies, which could have made drawing comparisons between the two programs difficult.

Because this chapter is focused on the core differences between security algorithms, liberties were also taken to standardize the way both programs pull cloud data, connect to databases, execute queries, and move data to the cloud. This reduces the number of variables determining each program's performance, and allows their key differences to be examined. What are these differences? Simply put, the Alzain Program uses the SSA to encrypt each record individually and stores its SSA Keys across several clouds. The Po1 Program, however, stores fake data alongside a user's real records, splits all of these records up by attribute, sub-attribute, and/or group of attributes, and uses the SSA to encrypt the realness of each record. The SSA's Keys are then stored across the clouds along with the broken-up records. One of the key performance differences in these algorithms, as will be seen later in this chapter, is the number of times that the SSA encryption is used. The Po1 Program carries out this encryption once per table, while the Alzain Program performs a SSA encryption once per record. Over time, the disparity in the total number of encryptions will lead to a major difference in the programs' execution times.

From an algorithmic standpoint, the two programs are constructed very similarly. Both start off by reading initialization data from a file, pulling all necessary tables into the secure local database, and connecting to the secure database. These operations are performed in exactly the same way in both programs. From that point, the Po1 Program performs operations to recombine fragmented tables together and determine the data types that will make up the View, while the Alzain Program moves straight into creating and populating its encrypted table. It carries out its decryption process in much the same way that the Po1 Program decrypts the Secret List of real records, though this decryption

is carried out once per record. Each record's set of keys is combined via mathematical interpolation to determine the Secret Value, in this case the base-10 representation of the entire record's contents. The Secret Value is converted into base-2 and then back into an ASCII string, which is parsed into attributes that are inserted into the database. Once all of the records are decrypted, the user is presented with their database table.

The Shutdown process for the Alzain Program also mirrors the one in the Po1 Program, though its encryption process is carried out differently. Rather than encrypting a Secret List, the records are encrypted. Each record is pulled out of the database, and all of its attributes are concatenated together with delimiting characters. The resulting ASCII string is next converted into binary and then into base-10, which is encrypted using the SSA. This process is carried out once per record, resulting in *n* sets of keys. Once all of the records are processed, each set of keys is pushed out to a different cloud and the local database is cleared before disconnecting from it.

5.3 Results and Comparisons

Many tests were carried out against the two programs to explore the Po1 Program's performance and capabilities. Each of the following subsections will describe one of the tests carried out, provide a graph of the test's results, and give an analysis of the results. All tests were carried out on a multi-user server where other users' processes were potentially running in the background. To create a level of consistency and so that statistical analysis could be carried out against the results, every data point on the following graphs was created by averaging together thirty distinct observations. This

measure helped to minimize any outlier data caused by other processes running at the same time as the tests. Fake records were not generated by the Po1 Program unless explicitly stated in the test description.

5.3.1 Attribute Splitting Test

The first test carried out was designed to see how the Po1 Program handled recombining and splitting a database table into groups of attributes in separate clouds using the Attribute-Group Fragmentation Scheme (section 3.2.2). The table to be stored in the clouds contained eight attributes describing a person's general contact information; its SQL creation command is as follows:

```
create table Contact(fname varchar(255), lname
varchar(255), gender varchar(255), address
varchar(255), city varchar(255), state varchar(255),
zip varchar(255), notes varchar(255));
```

Three scenarios were carried out: dividing the eight attributes evenly into two clouds, into four clouds, and into eight clouds. Within each scenario, the number of records stored in the table was varied from zero to seven-hundred, and the amount of time to complete the Setup and Shutdown operations were recorded. During these tests, no fake records were generated. This allowed for a one-to-one comparison about the volume of data being processed to be made without the ambiguity of the Po1 Program adding extra records pseudo-randomly.

The same tests were carried out with the Alzain Program. Because the Alzain Program does not split records up attribute-wise and store them in clouds with their SSA keys as

the Po1 Program does, comparisons were made based on how the Alzain Program stores the sets of keys generated from the SSA into the clouds. These operations are similar since both involve storing sets of encrypted data into separate clouds, so the comparison is warranted. The Alzain Program stores a set of keys in the clouds for each of its records, so the amount of data generated by both programs are also similar (recall that the Po1 Program stores all of its records and just one set of SSA keys in the clouds). As with the Po1 Program, three different scenarios were carried out with the Alzain Program to test how the number of clouds that the data are split into affects performance.

All eight of the Contact table's attributes were defined as 255-character strings to allow for a closer comparison between the two programs. The SSA keys generated by the programs must be stored as character strings to disallow truncation or rounding in the database. During the Shutdown process, the Alzain Program only generates SSA keys to be stored in the clouds, while the Po1 Program generates both SSA keys and groups of record attributes. If these record attributes had real-world datatypes (e.g. Integers, Dates, etc.), they would have to be stored separately from the SSA keys, increasing the amount of cloud storage space needed for the Po1 Program. Increasing the amount of cloud storage would also increase the amount of time necessary to pull all of the cloud data into the secure local database during the Program's execution. To standardize the timing of cloud data retrieval and to decrease the number of variables in this test, all of the Contact table's attributes were created as character strings of sufficient size to hold the generated SSA keys and the actual data in the same cloud table. In real-world practice (as shown in sections 5.3.7, 5.3.8, and 5.3.9), separate storage would be used to house the SSA keys in order to increase security and allow for more flexibility a table's datatypes.

Each of the three following graphs, Figures 5.1, 5.2, and 5.3, depicts the execution times for the Attribute-Group Fragmentation Scheme in two, four, and eight clouds, respectively. Two standard deviation vertical error bars surround each point. Because 95% of all observations will fall within the 2 standard deviation error bars and because the Po1 and Alzain Program's error bars do not overlap, it can be determined that there is a statistically significant difference between the Po1 Program's results and the Alzain Program's results.

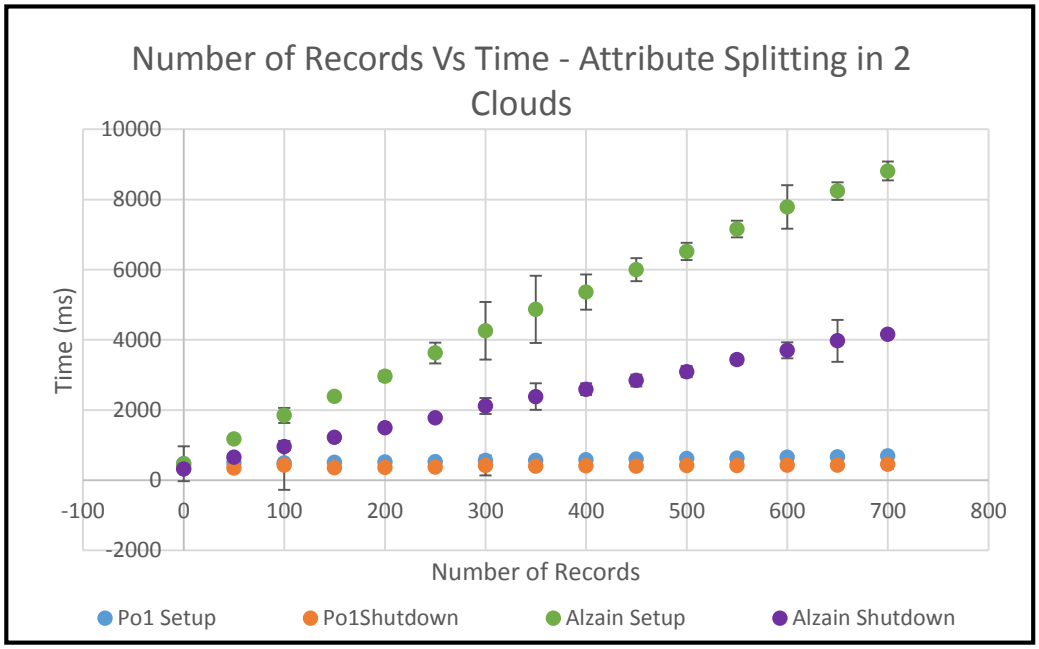


Figure 5.1 - Po1 and Alzain Attribute-Group Fragmentation Scheme Splitting 8 Attributes into 2 Clouds

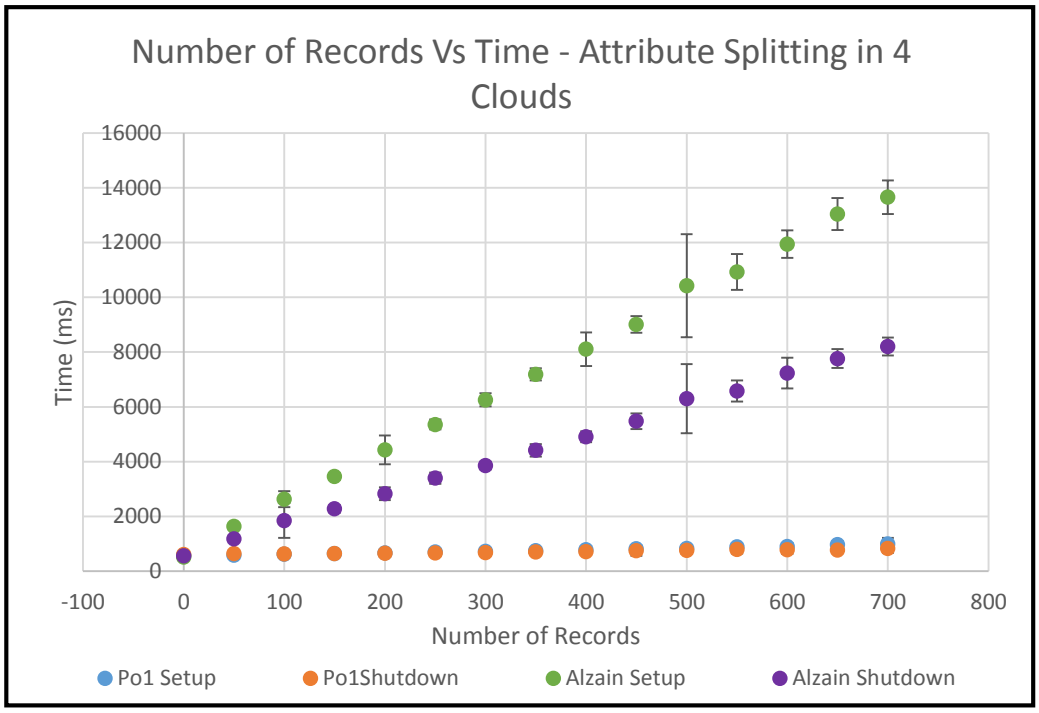


Figure 5.2 - Po1 and Alzain Attribute-Group Fragmentation Scheme Splitting 8 Attributes into 4 Clouds

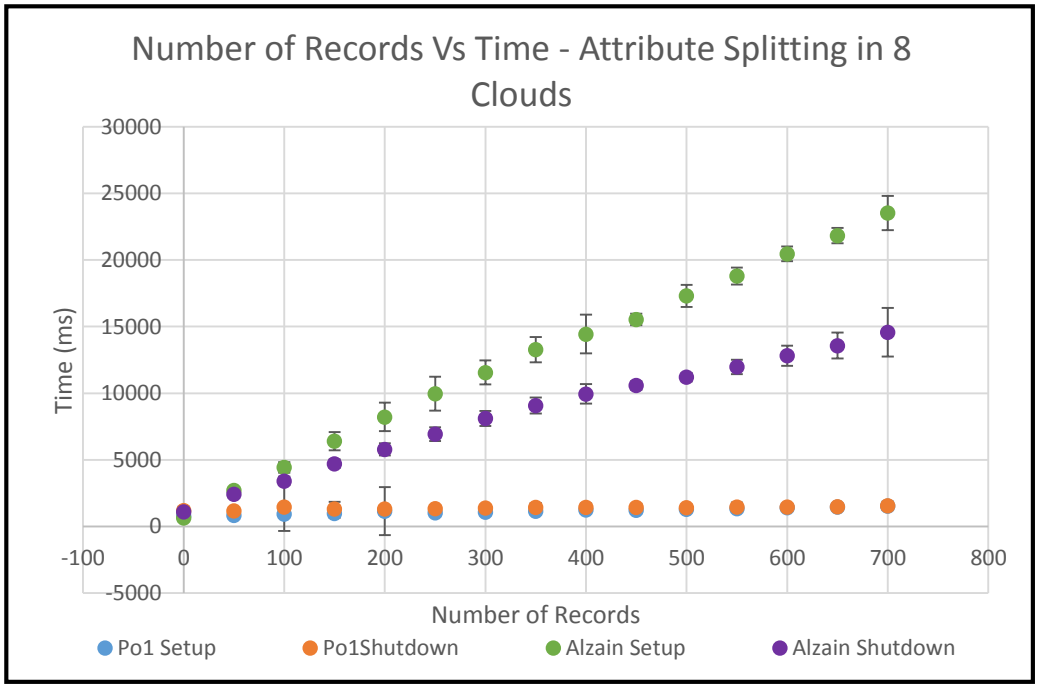


Figure 5.3 - Po1 and Alzain Attribute-Group Fragmentation Scheme Splitting 8 Attributes into 8 Clouds

The first observation to be made from this set of tests is that the Alzain Program performs much slower than the Po1 Program. This is due to the fact that it must carry out the SSA encryption and decryption process once per record. Therefore, as the amount of records increases, the amount of time needed to perform the Setup and Shutdown procedures also increases linearly with it. As more clouds are added to this scheme, the amount of time needed to complete these processes increases linearly (as will be further examined in section 5.3.4) because the number of SSA keys that are generated and processed is tied to the number of clouds used in the scheme.

These same observations can be made of the Po1 Program's performance. As the amount of records increases, so does the execution time, though not as drastically as with the Alzain Program. This is because the encryption/decryption process only takes place once for the whole table instead of once for each record. Execution time also increases for the Po1 Program as more clouds are added to its security scheme, as is to be expected. After all, extra time must be spent for each additional cloud table that is combined and decrypted to make up the View. These differences are apparent when comparing Figures 5.4, 5.5, and 5.6, the zoomed-in versions the Po1 Program's performance graphs. It is interesting to note when adding extra clouds to the Po1 Program that the Setup process's execution time tends to dominate the Shutdown time in the long run. This is because the process for decrypting records grows at a faster rate than the encryption process. Even though the Shutdown process starts off with a longer execution time in Figures 5.5 and 5.6 because of the larger overhead associated with the encryption process, the Setup process's time tends to outweigh it as the number of records stored in the clouds increases.

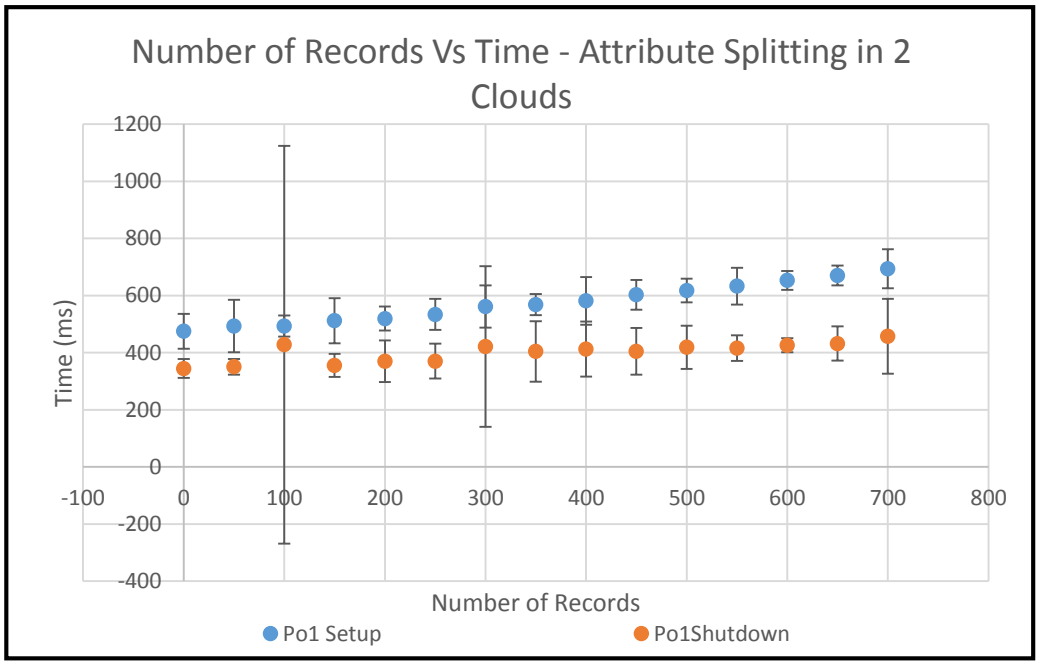


Figure 5.4 - Po1 Attribute-Group Fragmentation Scheme Splitting 8 Attributes into 2 Clouds

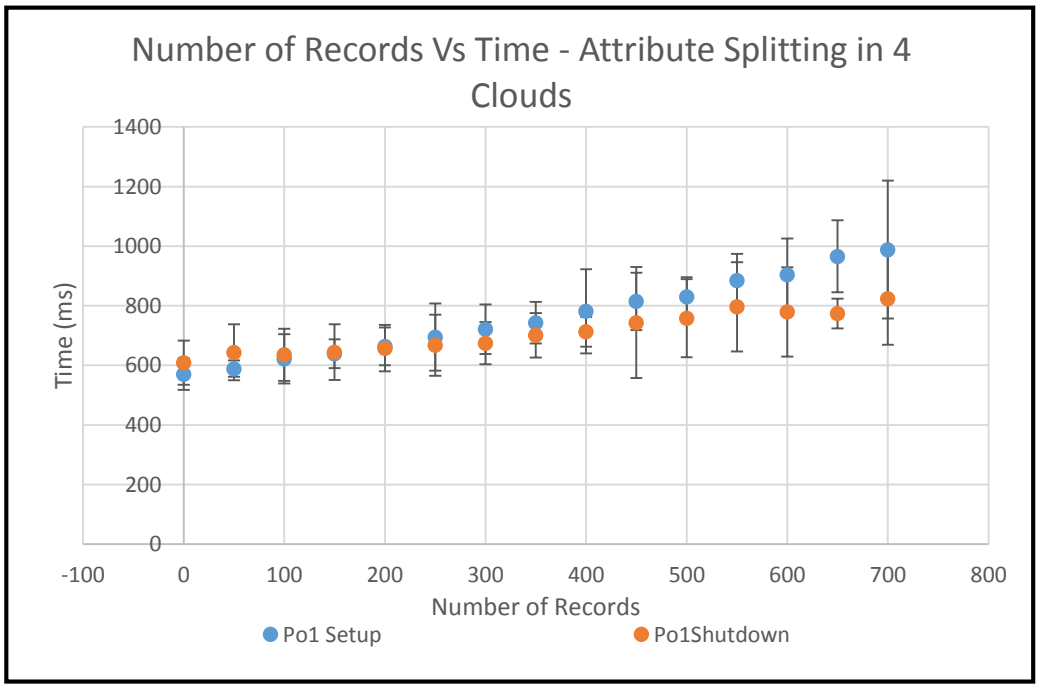


Figure 5.5 - Po1 Attribute-Group Fragmentation Scheme Splitting 8 Attributes into 4 Clouds

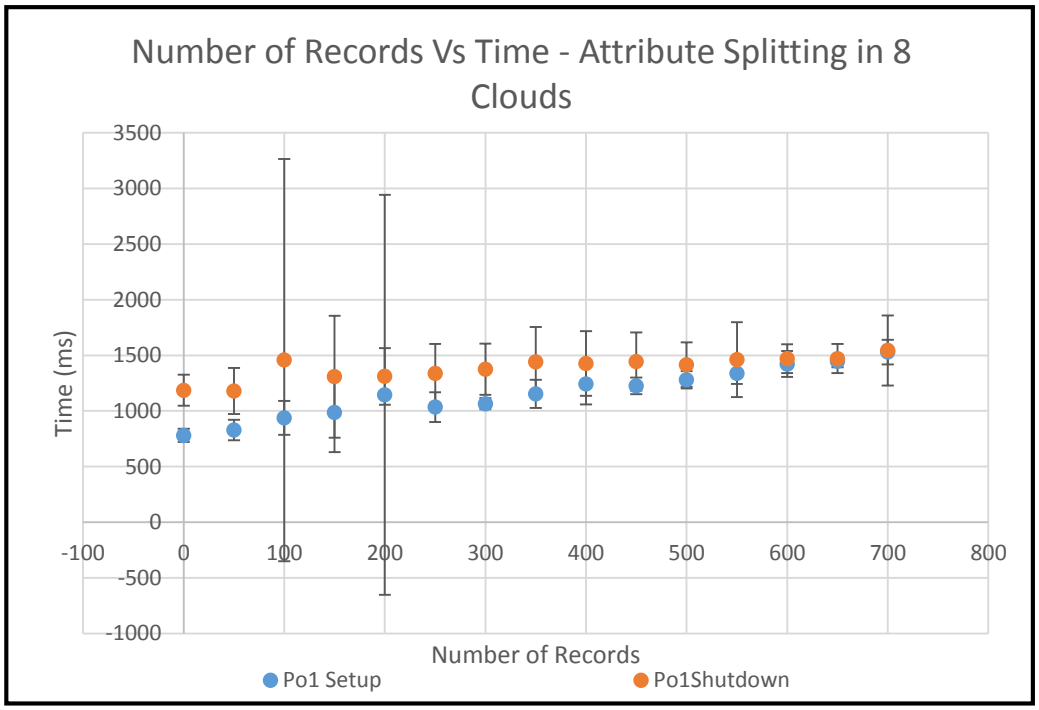


Figure 5.6 - Po1 Attribute-Group Fragmentation Scheme Splitting 8 Attributes into 8 Clouds

The results for this and many of the other tests are primarily linear. To understand why that is, one must look at the variables that influence execution time. For the Po1 Program's Setup and Shutdown phases, execution time is dependent on the number of SSA keys used and the number of records. Because the number of keys stays constant within each test (SSA keys are generally only added as clouds are added) and is a relatively small number compared to the number of records, it is the number of records that drives the execution time. The functions that facilitate the Setup and Shutdown phases exhibit a growth rate of $O(n)$ with respect to the number of records, creating a linear relationship between the number of records and execution time.

The Alzain Program behaves similarly, though different variables influence its execution time. The Alzain Program's Setup and Shutdown phases are loops that encrypt or decrypt each record individually. Each encryption and decryption is dependent on the number of SSA keys and the record's length (the number of characters that make it up). Because both of these values generally stay constant, the $O(n)$ loop processing each record dominates the growth rate and gives the Alzain Program its linear performance.

5.3.2 Sub-Attribute Splitting Test

The next test was designed to determine how the Po1 Program handles the Sub-Attribute Fragmentation Scheme. Recall that the Sub-Attribute Fragmentation Scheme was proposed in section 3.2.3 as a means to split sensitive database columns into sub-attributes. Each of these sub-attributes would only contain a subset of the characters of the original column, and would be stored in separate clouds for increased security.

In the three tests carried out by the Po1 and Alzain programs, Social Security Numbers (SSNs) were encrypted and decrypted. As in the previous test, the number of clouds that the records were stored in were varied between each test. The first test broke SSNs into two clouds, putting five digits into the first cloud and four digits into the second. The second test broke the SSN into four clouds, three of which contained two digits and one of which stored the last three digits. The last test broke the SSN down digit by digit into nine clouds, saving one digit in each of the clouds. Within each test, the number of SSNs saved in the database was varied from zero to seven-hundred, and the average

performance was calculated. The SQL creation statement to construct the encrypted "Socials" table used for these tests is as follows:

```
create table Socials(ssn varchar(255));
```

As discussed in the previous section, the "ssn" attribute was created as a 255-character string to allow the Po1 Program to store its SSA keys in the same tables as its fragmented records. If the "ssn" attribute was a nine-digit string instead, extra storage space would have been necessary for the Program's SSA keys, which would have added another variable to the tests. In the interests of minimizing the number of factors at play in the tests so that meaningful conclusions could be drawn, the "ssn" attribute's length was standardized in both tests to 255 characters.

The Alzain Program does not have the capability to break records down by attribute or sub-attribute, but it is able to logically break the records up in the form of SSA keys. To compare the performances of the two algorithms, the Alzain Program created one SSA key for every sub-attribute generated by the Po1 Program, as it did in section 5.3.1's tests. This provides a fairly good analog to the actions that the Po1 Program takes during its tests since both programs distribute their data across the clouds in an equivalent way. Figures 5.7, 5.8, and 5.9 depict the 2, 4, and 9-Cloud comparisons of the two programs' performances, respectively.

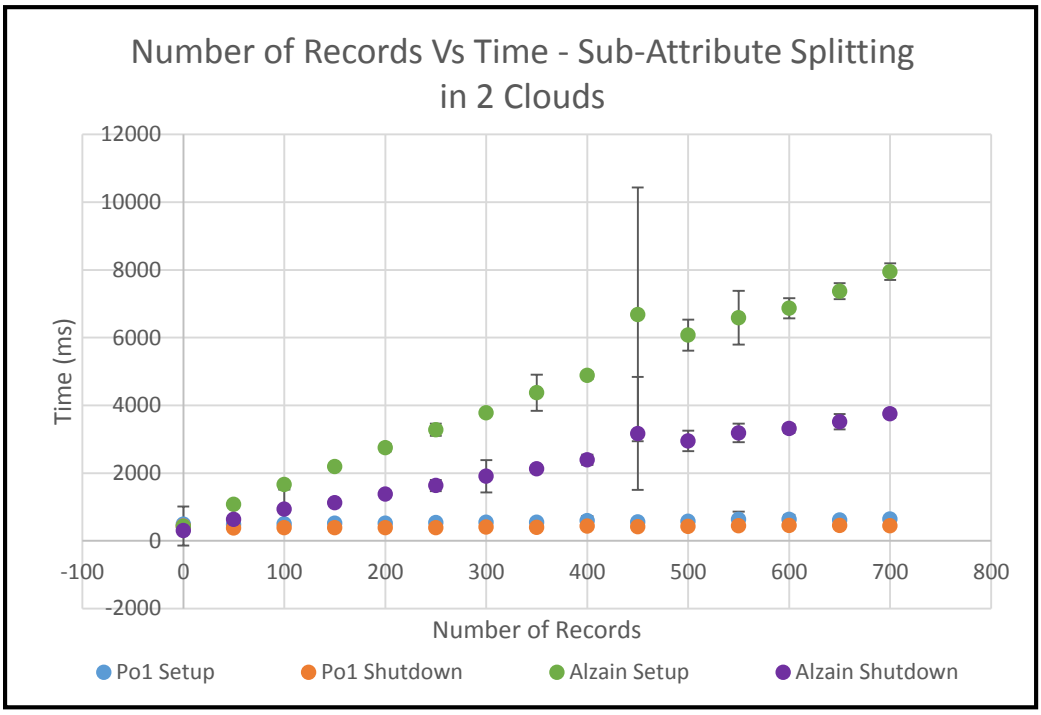


Figure 5.7 - Po1 and Alzain Sub-Attribute Fragmentation Scheme Splitting a Social Security Number into 2 Clouds.

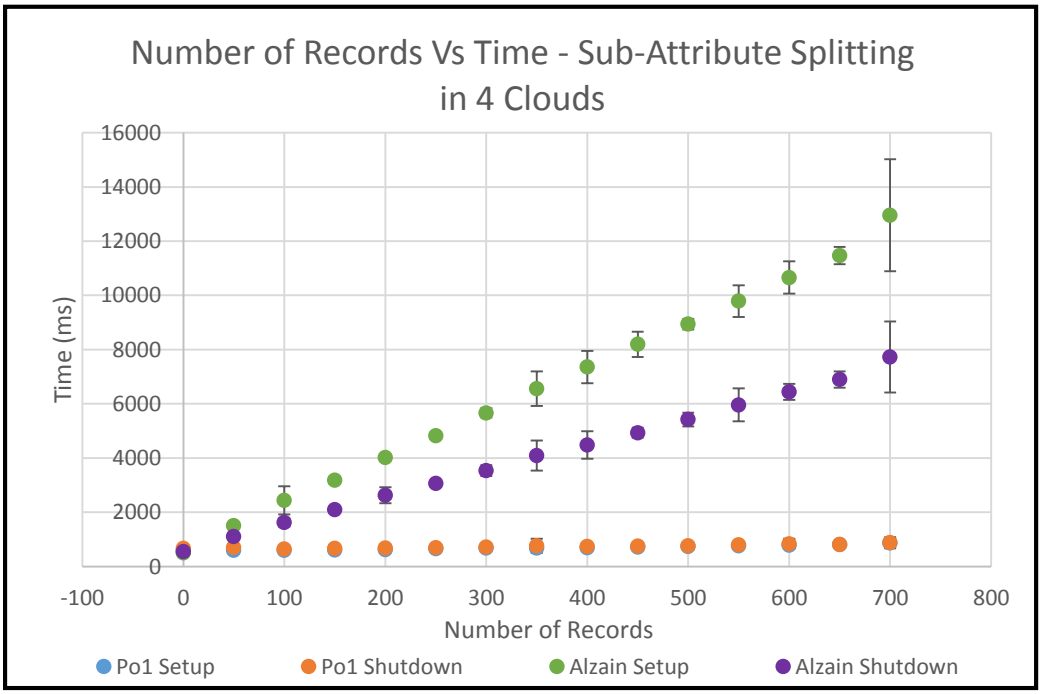


Figure 5.8 - Po1 and Alzain Sub-Attribute Fragmentation Scheme Splitting a Social Security Number into 4 Clouds.

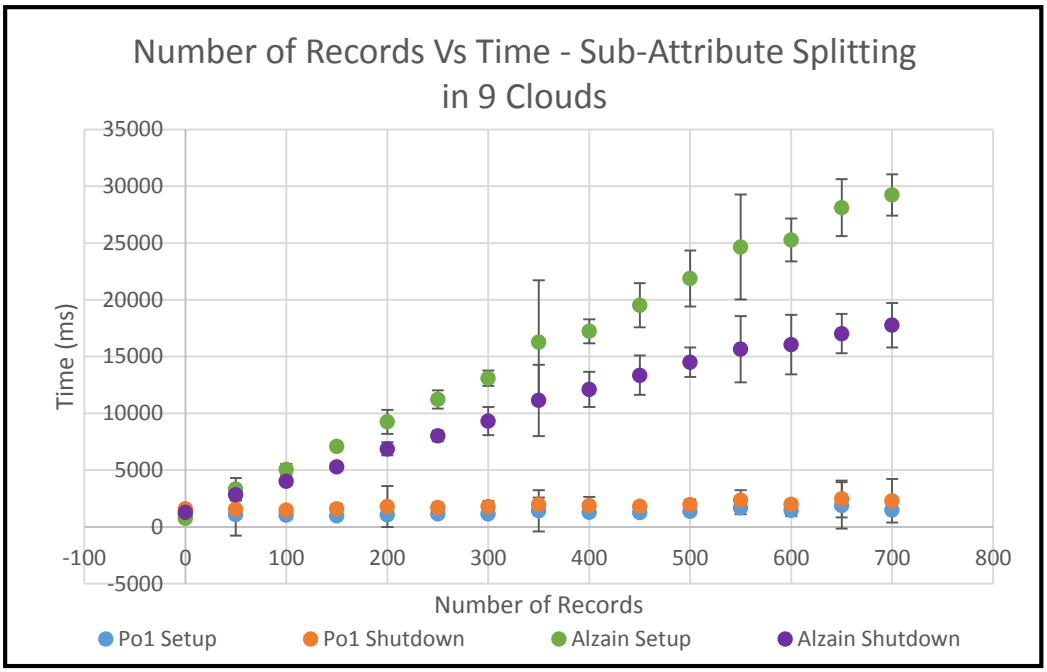


Figure 5.9 - Po1 and Alzain Sub-Attribute Fragmentation Scheme Splitting a Social Security Number into 9 Clouds.

As in the previous tests, two standard deviation error bars have been added to the data points to verify that there is a statistically significant difference between the two programs' performances. Also like in the last tests, the Po1 Program outperformed the Alzain Program. This is due to the fact that the Po1 Program only has to perform its decryption/encryption process once per run, while the Alzain Program's decryption/encryption process is run once per record. As the number of records increases, so does its runtime.

As the number of clouds were increased in the tests, both programs exhibited an increase in execution time. Several factors played into this. First, with the addition of each cloud comes an increase in network traffic time. Data have to be pushed or pulled from each

cloud added to the security scheme, which increases the overall runtime. An increase in the number of clouds also generally means an increase in the number of SSA keys that will be used in the decryption process. Adding keys not only makes the decryption computations more involved, it also implies an increased key generation time during the encryption process.

For the Po1 Program, there are tradeoffs associated with using the Sub-Attribute Fragmentation Scheme instead of the Attribute-Group Fragmentation Scheme. There is a larger overhead with the Sub-Attribute Fragmentation Scheme because extra time must be taken to recombine all of the fragmented tables before decrypting the SSA key pairs. This is why the Po1 Program's execution time when processing 0 records is higher in this section's tests than in the previous section. Despite this difference, the Sub-Attribute Fragmentation Scheme tends to run slightly better in the long run: as more records are added, the Program's execution time increases at a slower rate than when using the Attribute-Group Fragmentation Scheme. These observations can be made when comparing Figures 5.10, 5.11, and 5.12 (Figures 5.7, 5.8, and 5.9 without the Alzain Program's timing data) to Figures 5.4, 5.5, and 5.6.

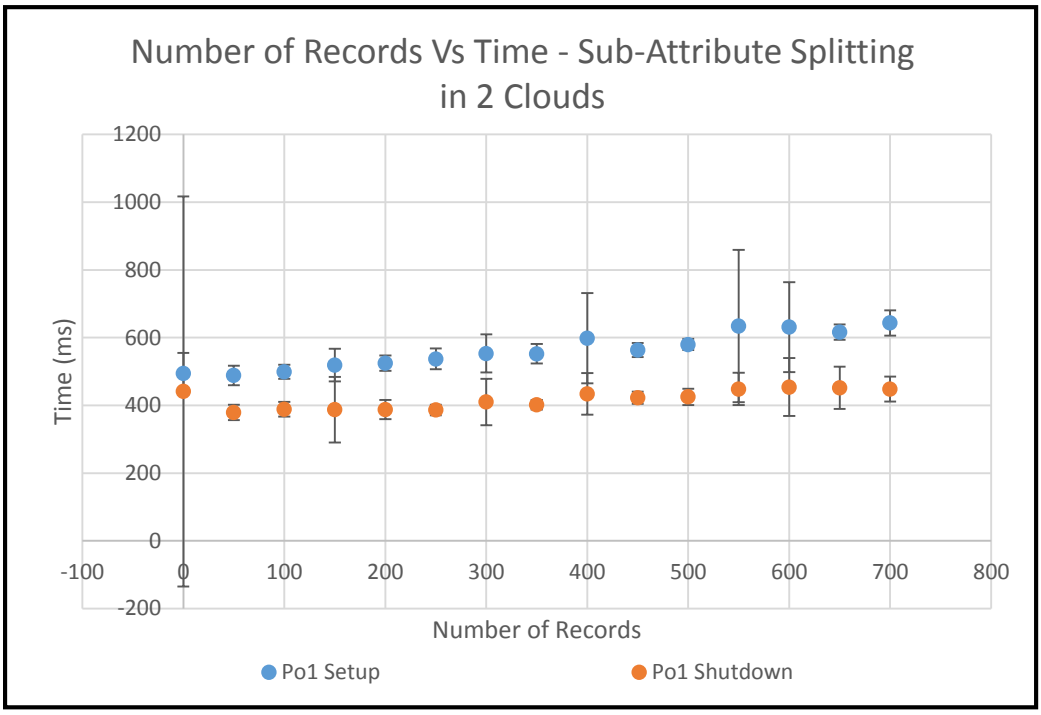


Figure 5.10 - Po1 Sub-Attribute Fragmentation Scheme Splitting a Social Security Number into 2 Clouds.

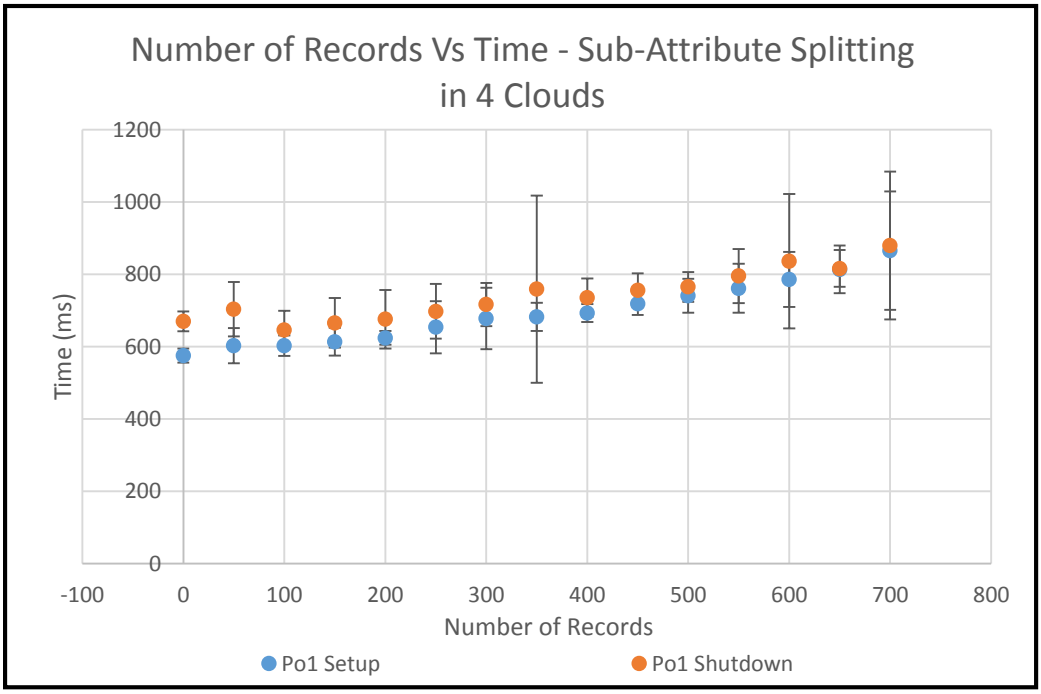


Figure 5.11 - Po1 Sub-Attribute Fragmentation Scheme Splitting a Social Security Number into 4 Clouds.

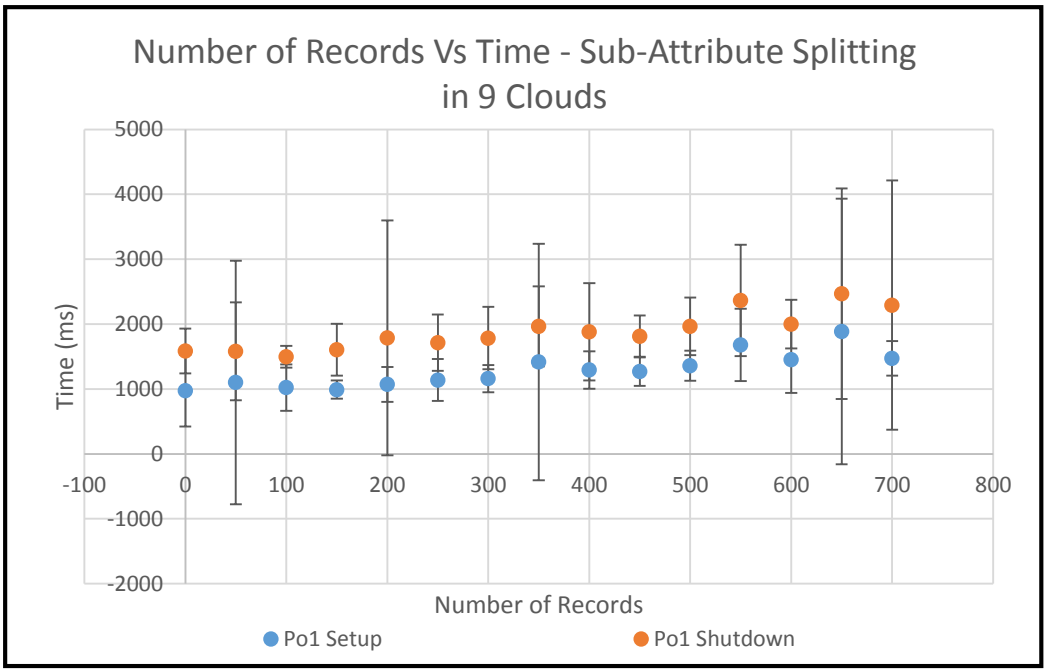


Figure 5.12 - Po1 Sub-Attribute Fragmentation Scheme Splitting a Social Security Number into 9 Clouds.

5.3.3 Data Import Test

One of the functionalities provided by the Po1 Program (and added into the Alzain Program) is the ability to import records from preexisting tables into the Program's encrypted table. This process starts by establishing a JDBC connection to the desired database so that all of the imported table's records can be copied using a SQL Select query. One by one, each of these records are then inserted into the program's encrypted table. This allows the program to perform any necessary processing to the records before they are stored. For the Alzain Program, only minor processing takes place to make sure that all attributes, regardless of datatype, are properly inserted. Much more must be done by the Po1 Program. Every record must be broken down to match the View's base table architecture before it is inserted, and after the insertion it must be added to the list of real

records. Once all of the records are imported, the rest of the Setup phase is carried out normally.

One test was carried out to study the programs' relative importing efficiencies. This test determined the amount of time necessary to import and encrypt differing amounts of records into an empty encrypted table. The Contact table split into eight clouds in section 5.3.1 was used as the empty encrypted table for this experiment. As Figure 5.13 shows, this is the one and only test where the Alzain Program outperforms the Po1 Program. The Po1 Program's Setup procedure takes the longest of all of the processes to complete because it cannot directly insert each record into its encrypted View. Instead, the Program must decompose each Insert query into 8 equivalent Insert queries, each of which inserts one attribute into a separate base table. The large overhead of creating and issuing eight times the amount of queries as the Alzain Program does is very visible in Figure 5.13, as the Po1 Setup procedure has the greatest slope.

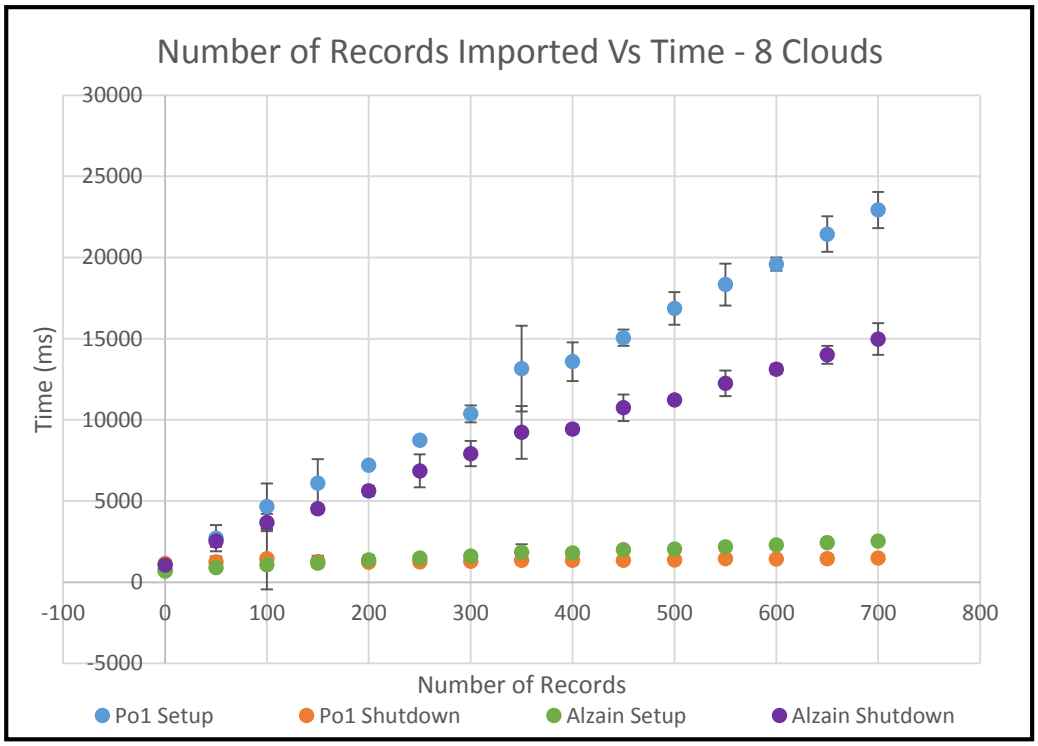


Figure 5.13 - Po1 and Alzain Programs Importing Preexisting Records into 8 Clouds

The Alzain Shutdown process has the next largest slope. This Shutdown process has a much greater time complexity than its Startup process because the Alzain Startup process does not have to perform any decryptions. Instead, it only has to insert records into its encrypted table, which greatly decreases its execution time. The Alzain Shutdown process, on the other hand, still has to perform a SSA encryption on every record, as usual. The Alzain and Po1 Shutdown timing results for this test are consistent with the ones from section 5.3.1's third test, which means that the Shutdown processes' efficiencies are not affected by importing data – only the Startup procedures are.

5.3.4 Number of Clouds Test

The Clouds test was designed to determine how changing the number of clouds a set of records is split across affects operation time. For this test, seven-hundred SSN records were inserted into the Socials table from section 5.3.2. The Socials table was then split up amongst a different number of clouds for each trial, and the Setup and Shutdown times for the Alzain and Po1 programs were recorded and averaged. The results are displayed in Figure 5.14.

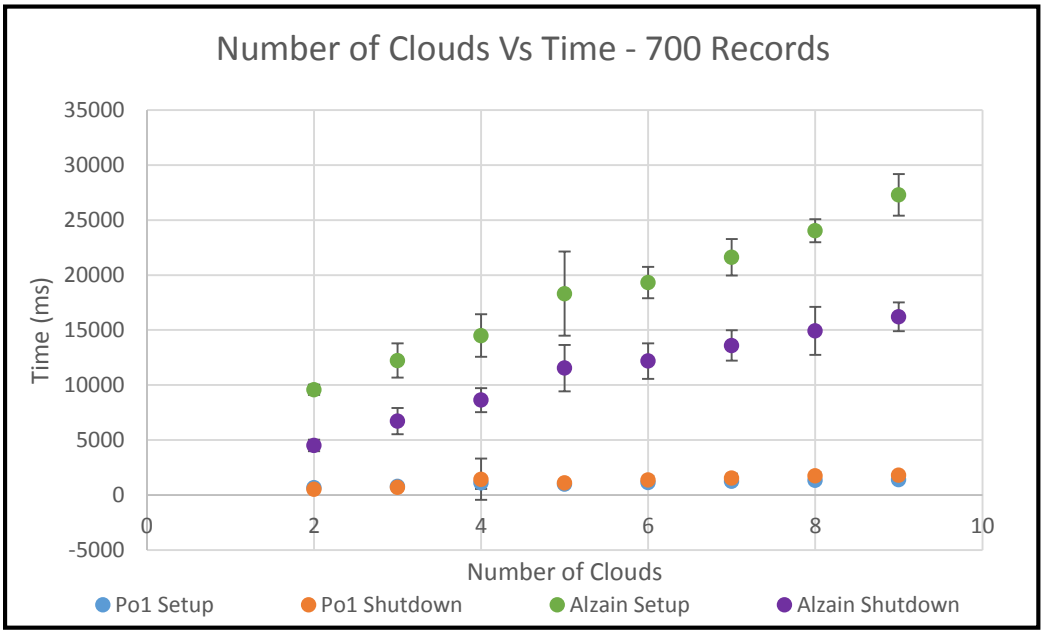


Figure 5.14 - Po1 and Alzain Programs' Performances on Differing Numbers of Clouds

As Figure 5.14 shows, there is a linear relationship between the number of clouds a table is split up into and the amount of time that the Setup and Shutdown processes take. As usual, the Alzain Startup and Shutdown procedures take the most time to complete because of their need to encrypt/decrypt each individual record. One interesting

observation to be made from this test is that it is very inexpensive, time-wise, to add more clouds to the Po1 Program's execution. Each cloud added to the Alzain Program increases its runtime by approximately 2.5 seconds, but the Po1 Program doesn't incur such large penalties. Much of the Po1 Program's security is derived from its distributed nature, so it seems only fitting that it would be inexpensive for the user to include as many clouds as necessary to protect their data.

The next five tests will only include Po1 result data. This decision was made because the tests being carried out do not have an analogous comparison to the Alzain Program, or because there would be too many variables to accurately make a comparison between the two programs.

5.3.5 Real Record Percent Test

The Real Record Percent test measures how the Po1 Program's performance changes when the percentage of real records in the database is varied while keeping the total number of records constant. The Contact table used in sections 5.3.1 and 5.3.3 was loaded with seven-hundred contact records and spread across eight clouds for this test. In each trial, a percentage of the seven-hundred records was selected to be "real", and were loaded into the View. These "real" records were selected by random to better approximate the Program's real-world use case where the real and fake records will be interspersed randomly. So, for example, thirty-five of the seven-hundred total records were randomly selected and loaded into the View during the 5% test. The timing results of the Setup and Shutdown phases are displayed in Figure 5.15.

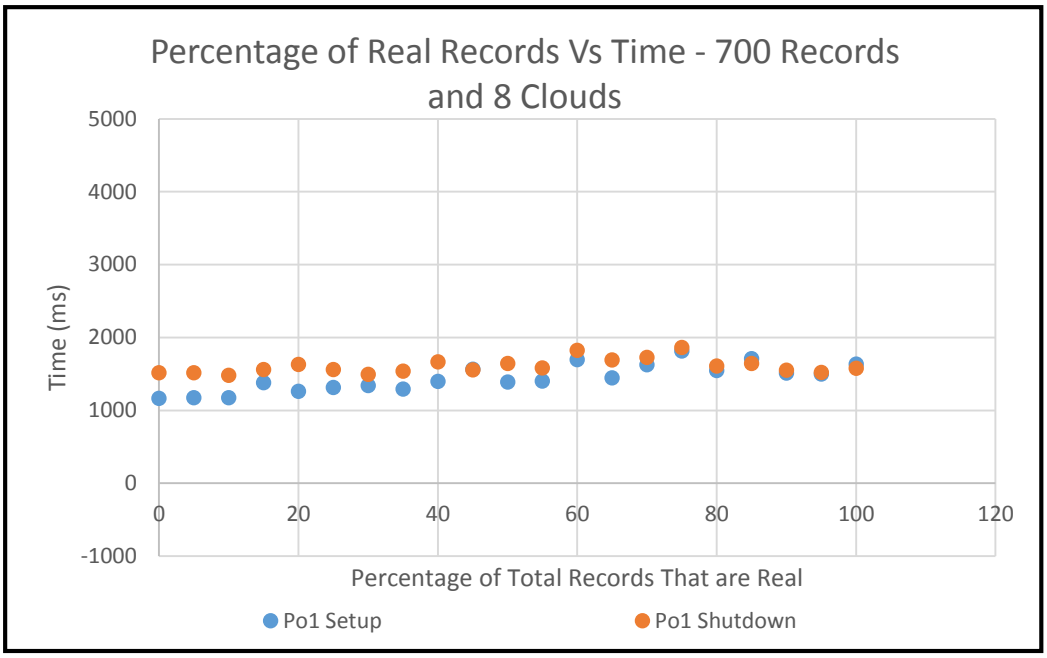


Figure 5.15 - The Po1 Program Varying the Percentage of Real Records Across 8 Clouds

According to Figure 5.15, there does not seem to be much of a correlation between the proportion of real to fake records and the reported time. This is especially true of the Shutdown process, whose value tends to stay around 1,550 milliseconds. The two processes do not appear to be affected by the percentage of real records because varying the percentage only influences the time it takes to construct the View during the Setup phase. All of the other major operations (encryption, decryption, pulling and pushing data from the clouds, etc.) perform in constant time since the number of SSA keys and the number of records being processed does not change between runs. The data variance seen in Figure 5.15 is due to changes in network latency. Since all of the other values remain constant and because the amount of time it takes to copy records to and from the clouds makes up a large portion of the overall Setup and Shutdown time, any changes in network traffic become very visible in the graph. If one ignores the slight curves in the Setup

phase's scatter plot, one will notice that it has a slightly positive slope. This slope is attributed to the increase in View setup time, depicted in Figure 5.16.

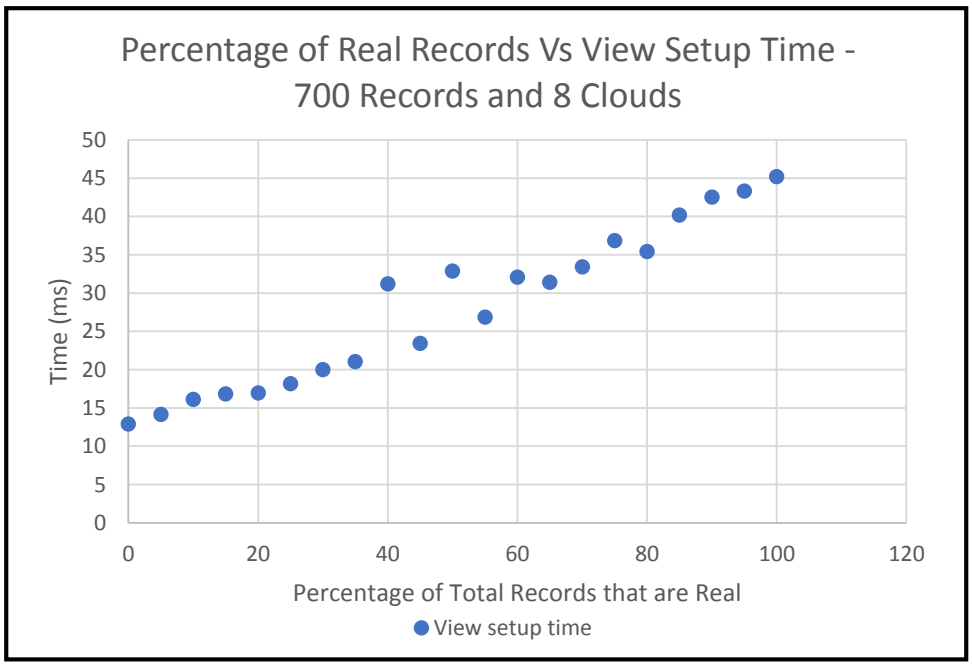


Figure 5.16 - The Effects of the Po1 Program Varying the Percentage of Real Records on View Setup Time

As Figure 5.16 shows, increasing the percentage of fake records is a very cost-effective means of increasing security. On average in this test, every decrease of 10% in the number of real records saves the user three milliseconds of execution time. The amount of time saved will vary depending on the number of clouds and the amount of records in question, but the fact that the View setup time accounts for less than 3% of the total Setup time in this test shows that adding fake records for security is not a burden on the decryption process. Fake records are also not much of a burden when being pushed and pulled from the clouds. During section 5.3.1's 8-cloud test (the test most similar to this experiment), records were copied to the clouds and back at an average rate of 0.5

milliseconds per record. The Real Record Percent tests saw record transfers at an average rate of 1.4 milliseconds per record. These data transfer measurements are dependent on network traffic, but all of the measurements made in this thesis have shown that adding fake records to increase security is quite feasible from a time-analysis standpoint.

The next three sections will describe the Po1 Program's operation on more-realistic datatypes. Sections 5.3.7 and 5.3.8 will give a realistic scenario for applying the Attribute-Group Fragmentation Scheme and Sub-Attribute Fragmentation Scheme. Section 5.3.9 will revisit the topic of importing data, this time exploring records of differing lengths and datatypes.

5.3.6 Fake Record Percent Test

This section, like the previous one, will analyze how changes to the percentage of real records stored in the clouds influences the Po1 Program's execution time. Rather than keeping the total number of records constant and varying the percentage of real records (as the previous section did), this section will keep a constant number of real records in each test and allow the total number of records to change as fake records are *added* to the real records. This distinction might seem small, but it allows the Po1 Program to be assessed from two different angles: the last section studied how the View setup time was influenced by the percentage of real records, while this section will study the effects of transferring an increasing number of fake records to and from the clouds with a constant View setup time.

Three sets of tests were carried out: one with one-hundred real records, one with two-hundred real records, and one with three-hundred real records. In each set of tests, records were stored in the Contact table defined in section 5.3.1. This table was split across eight clouds, as in the previous section. Eleven tests were run within each set, each with a different percentage of fake records added. For example, the 10% test added ten fake records to the set of one-hundred real records, twenty fake records to the set of two-hundred real records, and thirty fake records to the set of three-hundred real records. This resulted in 110% of the original records in each test, 10% of which were fake. All of the fake records were randomly interspersed amongst the real ones, as they would be in the Po1 Program's typical use case. An additional 10% of fake records (i.e., ten, twenty, or thirty records, respectively) were added in every test, culminating in the 100% test containing equal numbers of real and fake records. The Setup and Shutdown times were recorded for every test, and are presented in Figures 5.17 and 5.18.

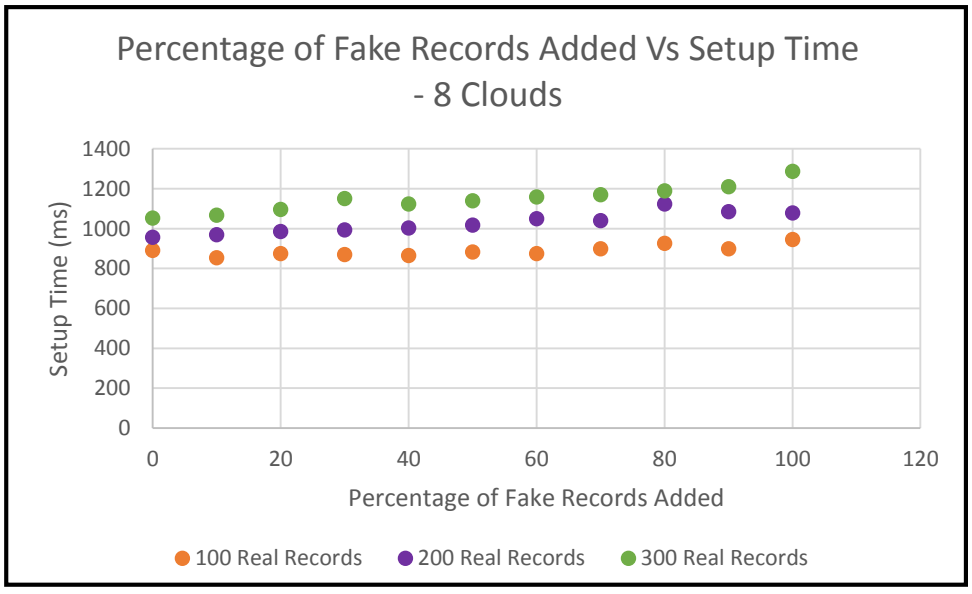


Figure 5.17 - The Po1 Program Varying the Amount of Fake Records across 8 Clouds During Setup

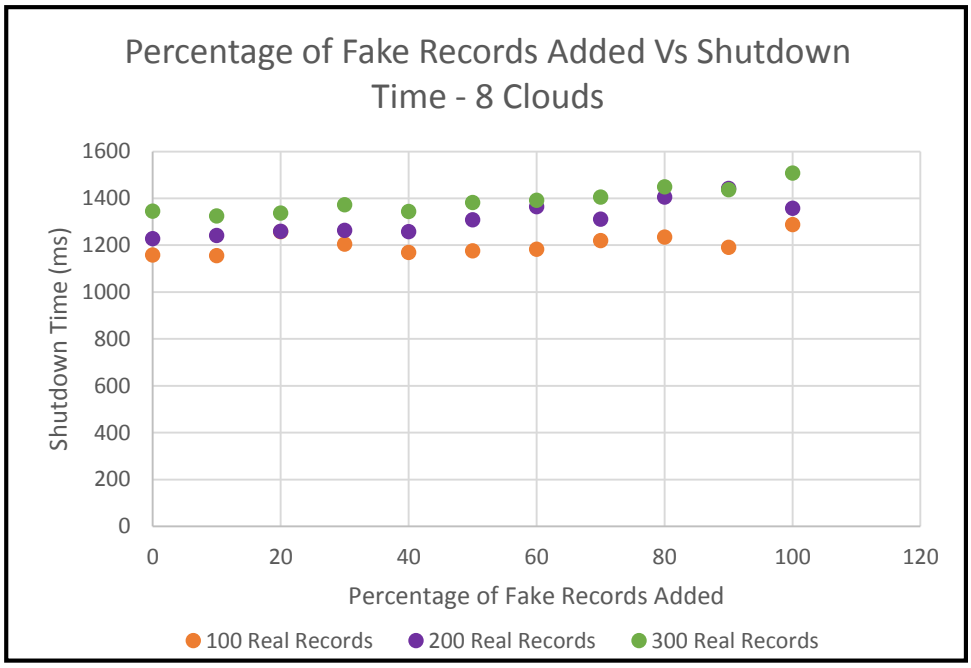


Figure 5.18 - The Po1 Program Varying the Amount of Fake Records across 8 Clouds during Shutdown

Unlike in Figure 5.15, both the Startup and Shutdown processes depicted in Figures 5.17 and 5.18 have a positive slope. Varying the amount of records used in this section's tests causes these slopes: increasing the amount of records being transferred to and from the clouds increases the execution times of the Setup and Shutdown processes. As more records are added at a time, steeper slopes are created. That is why the 300-record test grows at a faster rate than the other two tests, and why the 200-record test grows faster than the 100-record test.

Figure 5.19 is a graph of the amount of time needed to construct the View during Figure 5.17's Setup phase. As in the previous section's tests, the View setup process accounts for just a small portion of the overall Setup phase. In Figure 5.19, though, the View setup time is relatively constant. Figure 5.16's View setup time varied because the amount of

records placed in the View changed between each test. Because the amount of records in the View is constant in this section's tests, the View setup time is also constant. Therefore, the slopes seen in Figures 5.17 and 5.18 are mainly due to the increasing data transmission costs associated with moving an increasing number of records in each test. The amount of time used to push and pull data to and from the clouds is graphed in Figures 5.19 and 5.20, respectively.

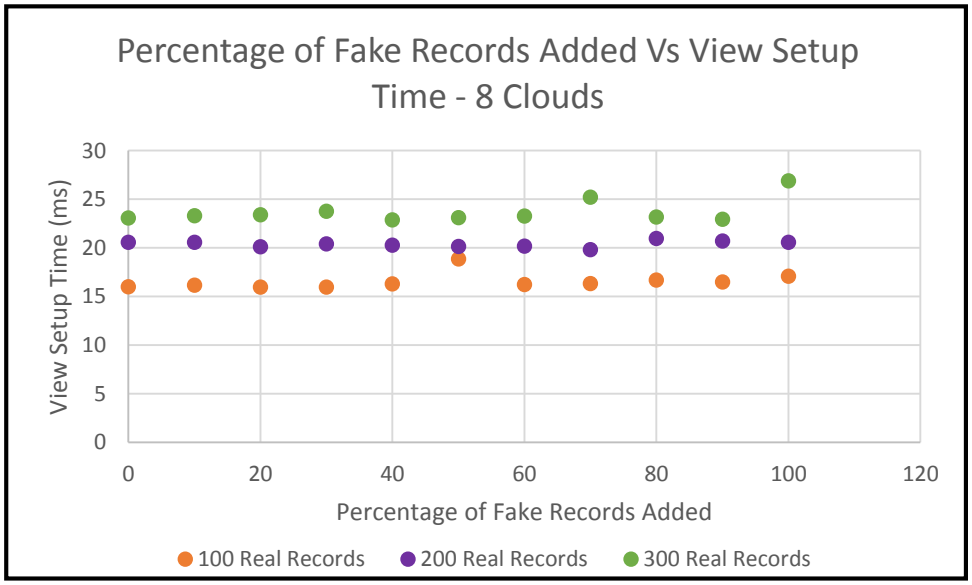


Figure 5.19 - The Po1 Program Varying the Amount of Fake Records across 8 Clouds during the View Setup Process

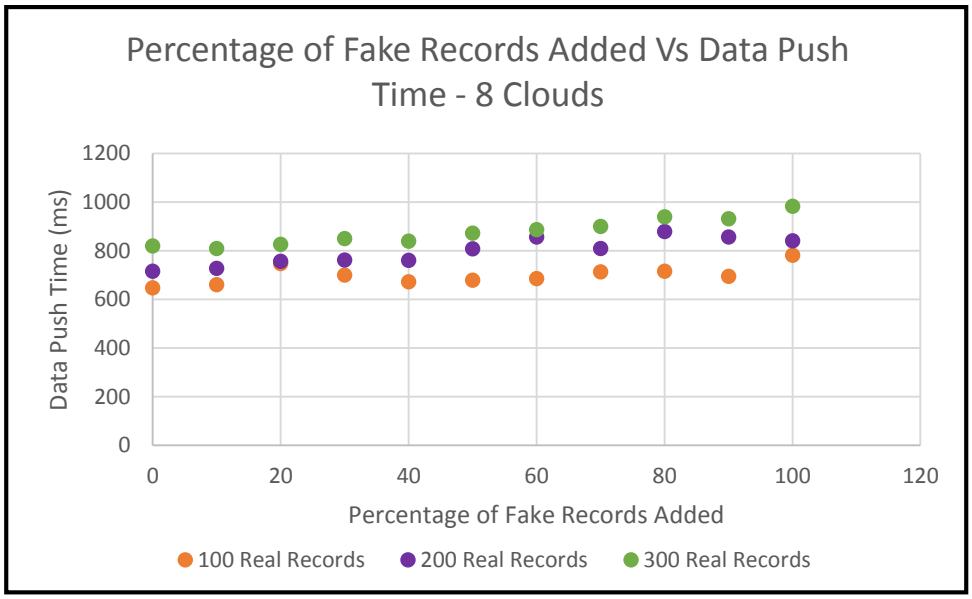


Figure 5.20 - The Po1 Program Varying the Amount of Fake Records across 8 Clouds When Sending Data to the Clouds

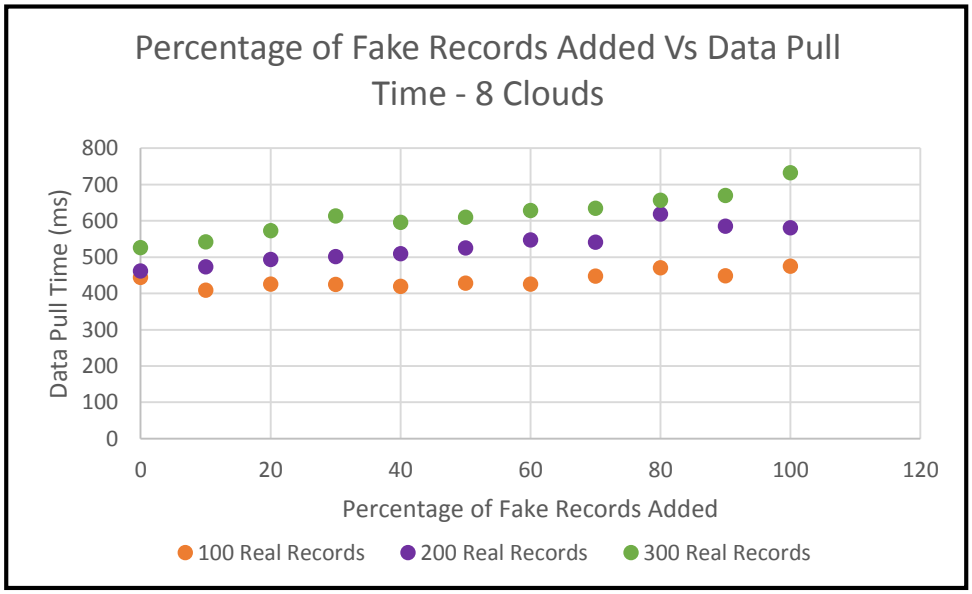


Figure 5.21 - The Po1 Program Varying the Amount of Fake Records across 8 Clouds When Copying Data from the Clouds

Figures 5.20 and 5.21 exhibit expected results – as more fake records are added to the database, more time must be expended to move them to and from the clouds. Also, as

expected, the time needed to push data to the clouds is greater than the time needed to pull it locally. This result was observed in many of the previous tests, and is one of the contributing factors to the Shutdown process' overhead costs being larger than the Setup process' (as in Figure 5.4). Based on the slopes of the test results in Figures 5.20 and 5.21, records are being transferred to and from the clouds at a rate of between 0.5 and 0.8 milliseconds per record, which is consistent with section 5.3.5's data transfer rate. The tests carried out in this section have shown once again that it is quite feasible and cost-effective for the HSA to add fake records for security purposes.

5.3.7 Real-World Attribute Splitting Test

As in section 5.3.1, this section will apply the Attribute-Group Fragmentation Scheme to groups of records and study its effect on the Setup and Shutdown processes. Instead of evenly dividing all of the records up attribute-wise across the clouds as was done in the aforementioned chapter, this section will take a more real-world approach to its data dissemination.

This section and the next two sections study the encryption of the "Medical" table. This table, which stores general medical patient information, contains eight attributes. It can be created with the following SQL creation query:

```
create table Medical(name varchar(30), gender
varchar(1), SSN varchar(9), birthday varchar(10),
address varchar(35), city varchar(25), phone
varchar(18), patientNum integer);
```

For this test, the Medical table was logically broken up and stored in six clouds: Cloud 1 stored the name and gender attributes, Cloud 2 stored the SSN attribute, Cloud 3 contained the birthday attribute, Cloud 4 stored the address and city columns, Cloud 5 contained the patient's phone number, and the patient's identification number was stored in Cloud 6. This attribute grouping scheme simulates a more realistic scenario than section 5.3.1 did because it only allows attributes to be stored in the same cloud if they would not prove to be incriminating if discovered together. That way, the amount of information an adversary would gain during a security breach would be strategically minimized to lessen the damage.

Because the datatypes for this experiment were varied, separate storage had to be created specifically to contain the Po1 Program's SSA keys. Eight keys were used in this SSA scheme. Each of these keys was stored in its own table in a separate cloud. One key was stored in each of the six clouds containing attribute information (Clouds 1 through 6), and the last two keys were stored in Clouds 7 and 8, respectively. Each of the eight key-bearing tables contained only two attributes: an integer that all Po1 records contain denoting which record number the attribute belongs to, and a 255-character string in which the SSA key was stored.

In this test, the number of records split across the six clouds was varied between each trial, and timing data were collected for the Po1 Program's Setup and Shutdown phases. The results of the Real-World Attribute Splitting Test are graphed in Figure 5.22. As was noted in section 5.3.1, the Setup phase grows at a faster rate than the Shutdown phase does because of the decryption process' time complexity. This means that, in the long

run, the Setup phase's execution time will dominate the Shutdown phase's time and eventually determine the Program's overall execution time. The next observation to be made is that this six-cloud security scheme runs more slowly than the Po1 eight-cloud scheme in section 5.3.1 (Figure 5.6). This is due to the extra data transfer time needed to copy the eight tables containing SSA keys to and from the clouds.

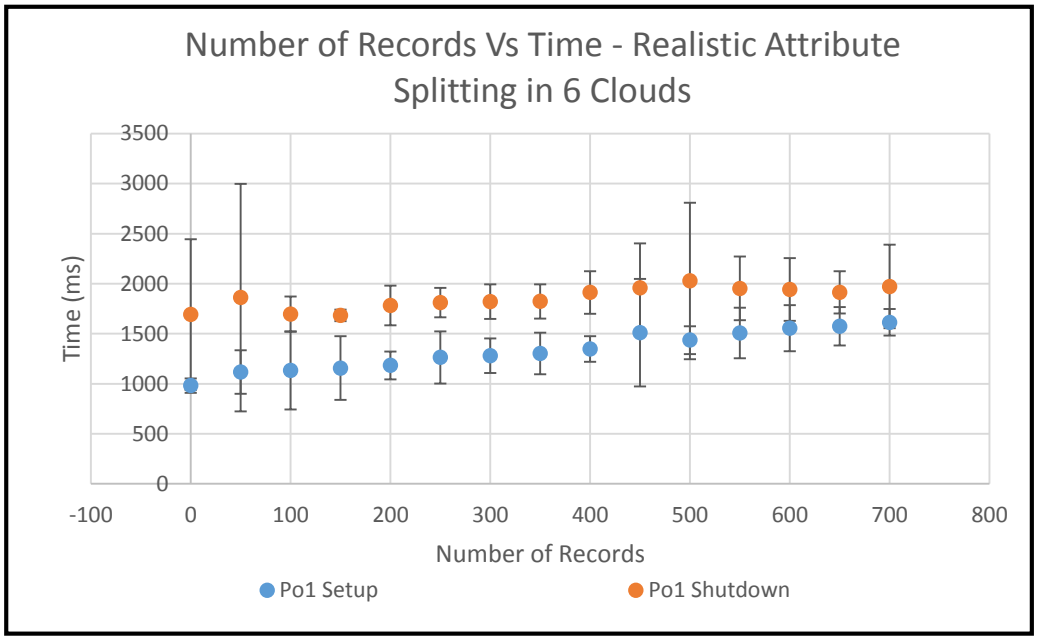


Figure 5.22 - Realistically Splitting a Table up into 6 Clouds with the Po1 Program

The difference in execution time between Figures 5.22 and 5.6 is not much to the user, but it highlights the fundamental tradeoff between security and speed of execution. By letting the user define exactly how they would like their data distributed and by allowing them to store their SSA keys separately from their data, more security is gained in this scenario than in section 5.3.1. At the same time, this test does not run as fast as section 5.3.1's 8-cloud test. Speed must often be sacrificed for an increase in security, as is the case in this test.

5.3.8 Real-World Attribute and Sub-Attribute Splitting Test

The test carried out in this section is very similar to the previous section's test. Both tests seek to securely split the Medical table amongst the clouds, though this test takes the splitting a step further. Rather than storing the table in 6 clouds, this test examines the relative efficiency of storing it into 8 clouds.

This test will distribute the Medical table's attributes into the clouds in much the same way as in the previous section. The major difference between the two tests comes from splitting the patient's Social Security Number field into sub-attributes. Instead of saving the "ssn" attribute entirely in Cloud 2, this section's test first splits it into three sets of three characters. The first set of three characters stored in Cloud 2, the second three characters are stored in Cloud 7, and the last three characters are stored in Cloud 8. The results of this tests are shown in Figure 5.23.

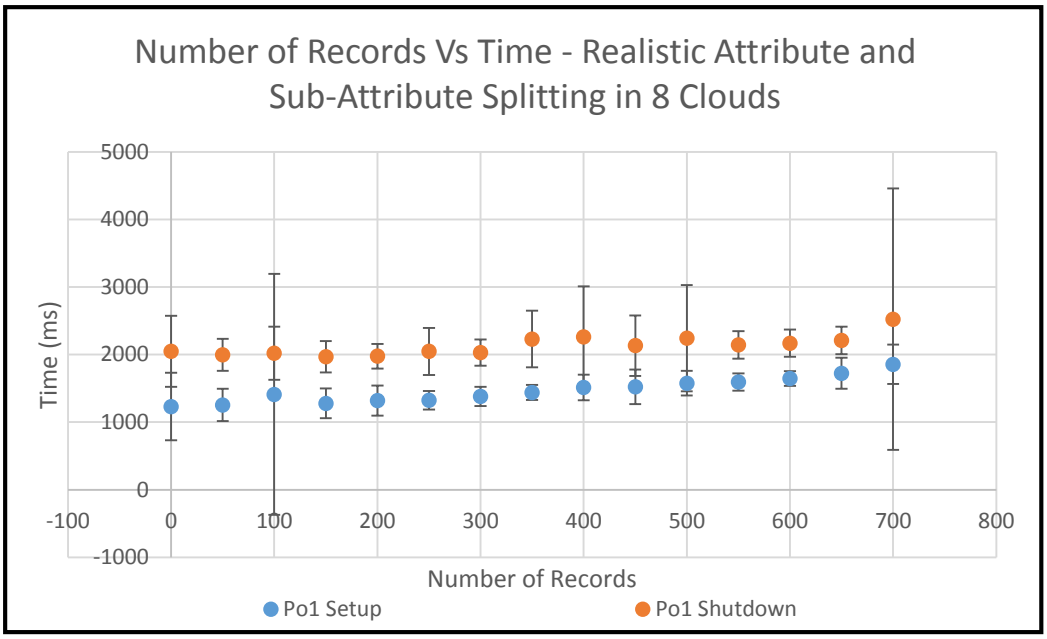


Figure 5.23 - Realistically Splitting a Table up into 8 Clouds with the Po1 Program

Figure 5.23's test results are consistent with those from the previous section. In both cases, the Shutdown phase initially takes more time to complete, though the Setup phase will dominate its performance in the long run. When the results from Figures 5.23 and 5.22 are directly compared, one will notice that there is not much of a difference, time-wise, between the two experiments. In fact, in most cases the one and two standard deviation error bars overlap between the 6 and 8-cloud tests. This means that there is no statistically significant difference between the reported execution times. In other words, adding the security of two more clouds and further protecting the user's SSN information came at no additional cost. Figure 5.24 shows this comparison, complete with one standard deviation error bars.

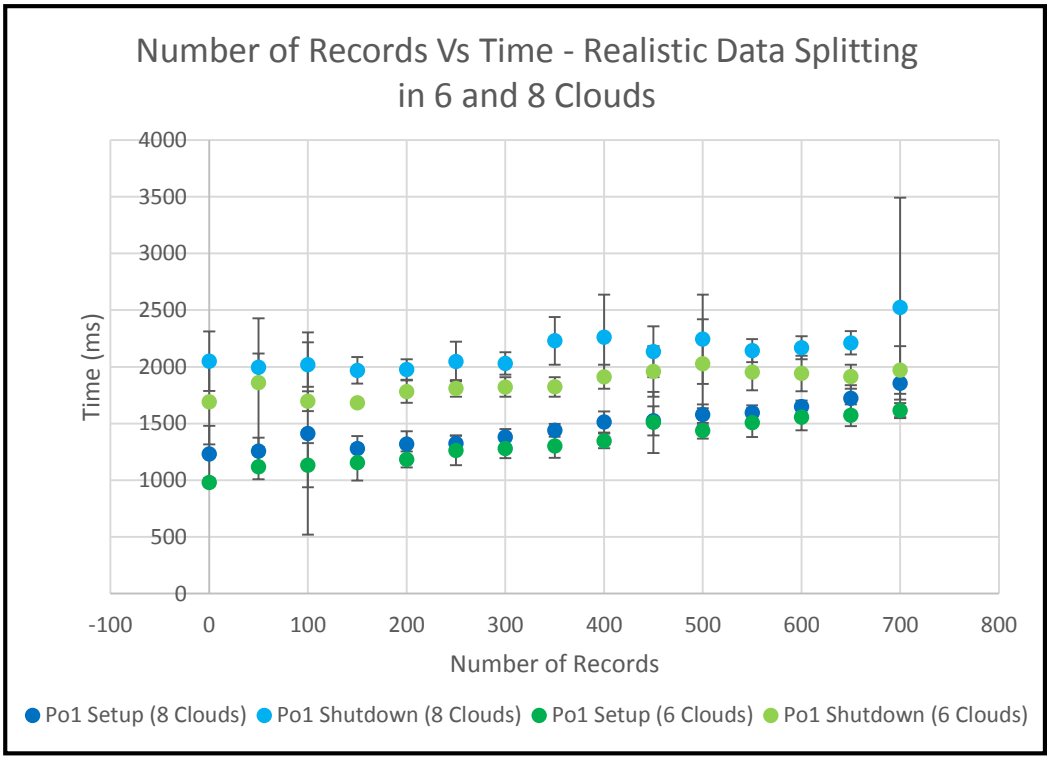


Figure 5.24 - Comparison Between Figure 5.22 and 5.23's 6 and 8 Cloud Data Splitting Schemes. One Standard Deviation Error Bars Added

5.3.9 Real-World Record Import Test

The Real-World Record Import Test explores the relative efficiency of importing preexisting tables containing varied datatypes into the Po1 Program's encrypted View. For this experiment, the Medical table used in the previous two tests was filled with differing amounts of data. In every trial, all of the Medical table's records were inserted into the empty View from a remote database. Unlike in the previous two experiments, each of the View's attributes was stored in a separate cloud for this test. This was done so that comparisons could be drawn with section 5.3.3's observations. The amount of time needed to complete the Setup and Shutdown phases were recorded, and are presented as Figure 5.25.

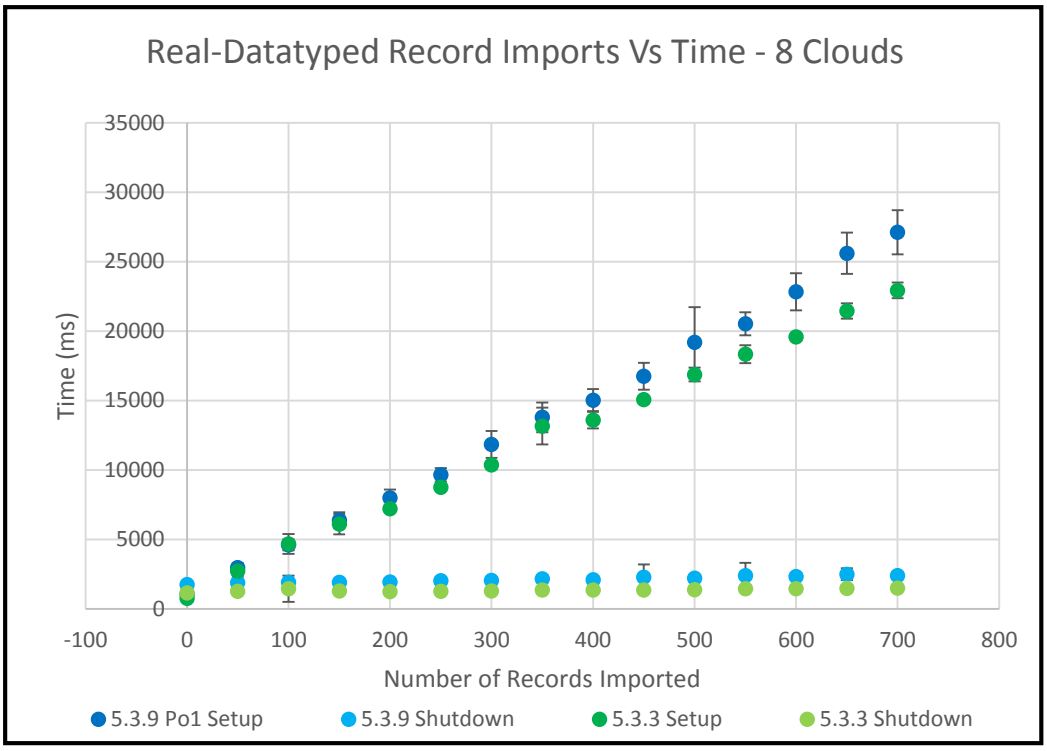


Figure 5.25 - Sections 5.3.9 and 5.3.3's Po1 8-Cloud Import Test Results Compared. One Standard Deviation Error Bars Added

As was to be expected, this chapter's experiments took more time on average to complete than section 5.3.3's import tests. This is because the SSA keys in this experiment had to be stored in separate tables from the attribute data since the attribute data's datatypes vary. Twice the number of tables had to be copied to and from the clouds in this test as compared to section 5.3.3's test, leading to an increased data transmission time and overall longer Setup and Shutdown times. Furthermore, the one standard deviation error bars in Figure 5.25 do not overlap between the two tests as the number of imported records grows, and nor do the 2-σ error bars. This means that there is a statistically significant difference between the two measurements as the number of records grows. Importing tables with varied datatypes may require more storage and longer loading

times, but it is a tradeoff worth making. Without this capability, the Po1 Program's applicability would be severely limited.

5.4 Security

Section 5.3 compares the relative efficiencies of the Po1 and Alzain programs, and found that the Po1 Program outperforms the Alzain program in most every case. This section compares the programs from a security standpoint to see whether the Po1 Program is also more secure than the Alzain Program.

One useful measure of security for key-generating algorithms is the minimum key length. Key length, described in number of bits, dictates the average time an adversary will spend brute forcing their way through an encryption. On average, an attacker will have to guess half of the possible keys in the key space before they find the correct one. Therefore, the longer the key, the larger the key space and the more combinations that will need to be tried before the correct key is guessed. Based on the current and projected computational power of encryption-breaking algorithms, the US National Institute of Standards and Technology (NIST) recommends that keys be at least 128 bits long, though 112-bit keys will suffice until the year 2030 [Katz15, page 356]. These key lengths were picked because of the staggering amount of time it would take a computer to brute force through an encryption of that size.

In order to determine how the Alzain and Po1 programs compare to each other security-wise, the minimum key lengths generated during the tests from sections 5.3.1 and 5.3.2

were recorded. Figures 5.26, 5.27, and 5.28 correspond to the 2, 4, and 8-Cloud attribute splitting schemes (Figures 5.1, 5.2, and 5.3), and Figures 5.29, 5.30, and 5.31 correspond to the 2, 4, and 9-Cloud sub-attribute splitting schemes (Figures 5.7, 5.8, and 5.9), respectively.

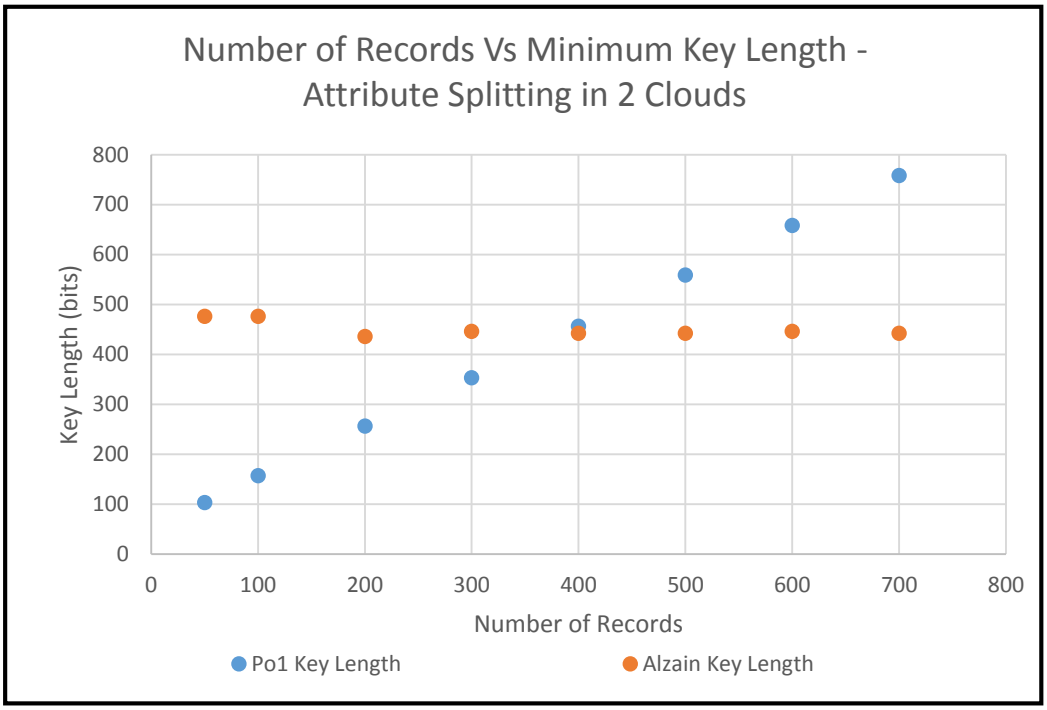


Figure 5.26 - Minimum Po1 and Alzain Program SSA Key Length Generated During the 2-Cloud Attribute Splitting Test

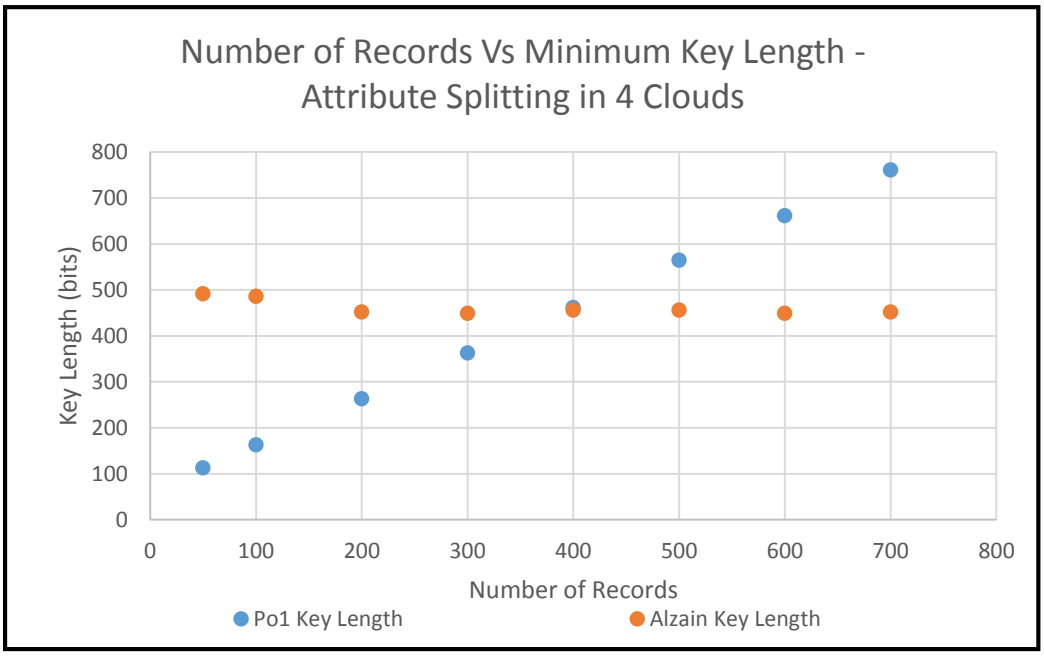


Figure 5.27 - Minimum Po1 and Alzain Program SSA Key Length Generated During the 4-Cloud Attribute Splitting Test

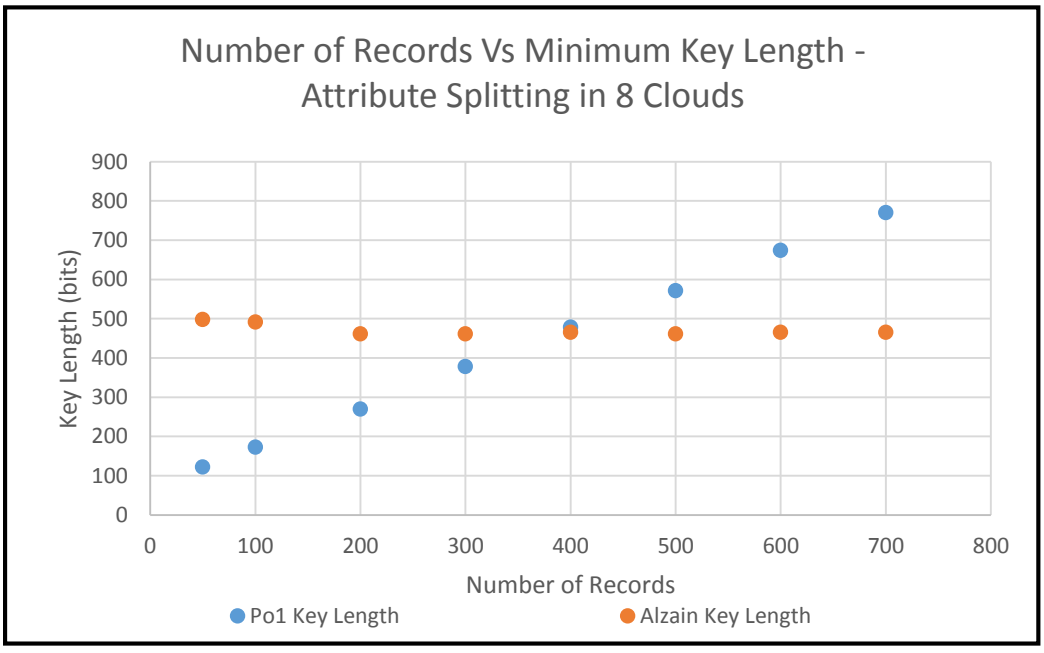


Figure 5.28 - Minimum Po1 and Alzain Program SSA Key Length Generated During the 8-Cloud Attribute Splitting Test

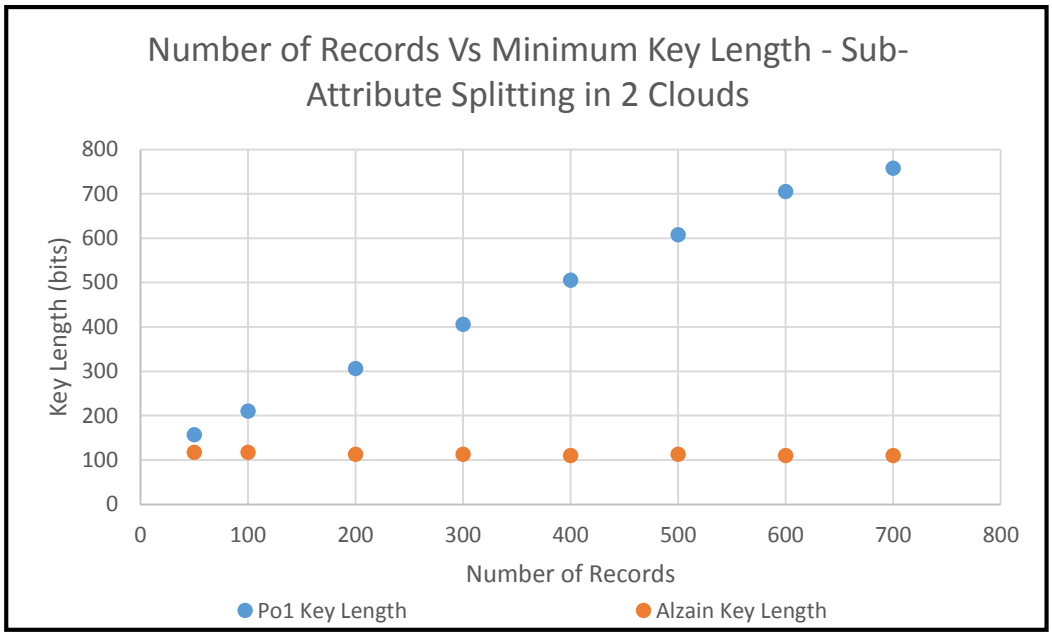


Figure 5.29 - Minimum Po1 and Alzain Program SSA Key Length Generated During the 2-Cloud Sub-Attribute Splitting Test

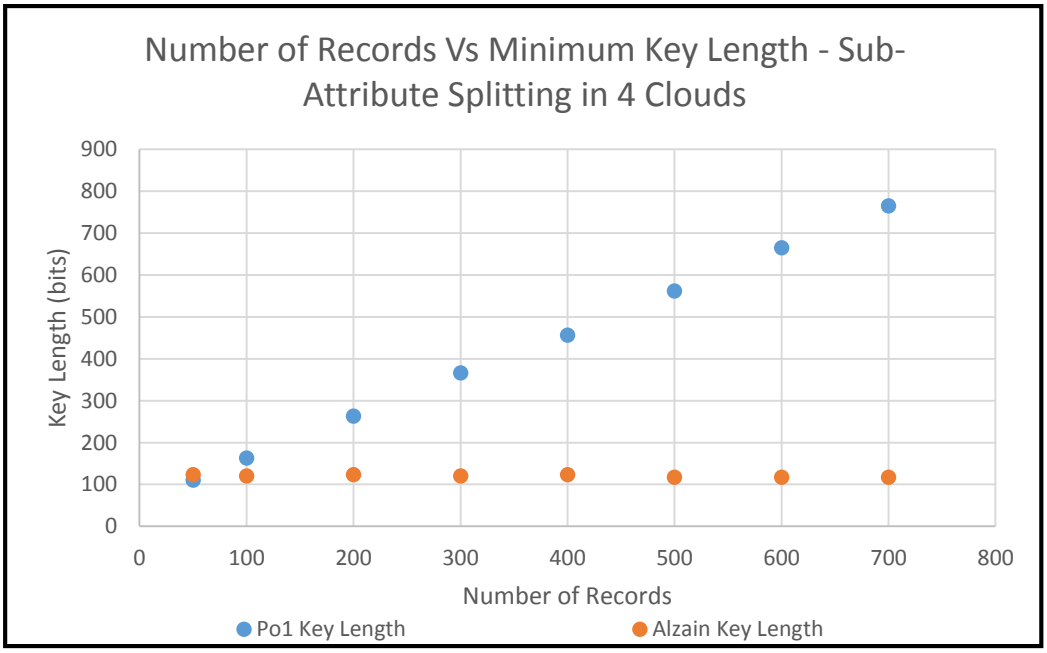


Figure 5.30 - Minimum Po1 and Alzain Program SSA Key Length Generated During the 4-Cloud Sub-Attribute Splitting Test

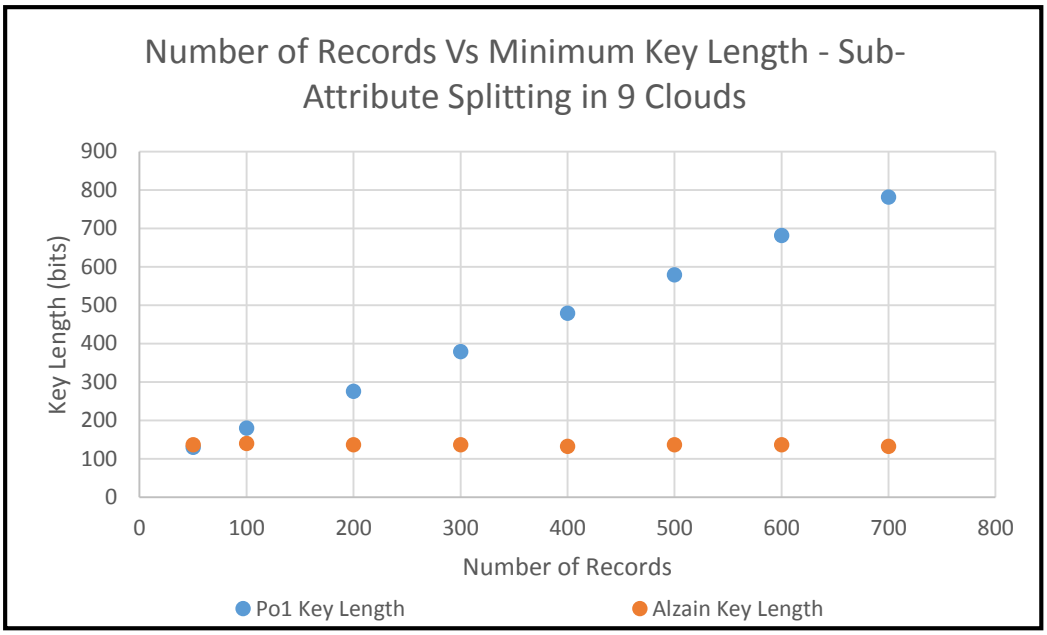


Figure 5.31 - Minimum Po1 and Alzain Program SSA Key Length Generated During the 9-Cloud Sub-Attribute Splitting Test

The first noticeable observation about these graphs is that the Po1 Program's minimum SSA key length is directly correlated to the number of encrypted records. This is because the list of real records is the Secret value being encrypted. As more records are added to the list, the Secret and its encrypted value grows. The SSA keys are generated using this Secret value, so key length increases proportionally to the number of records at an average rate of 1 bit per record. On the other hand, the Alzain Program's key length does not seem affected by the number of records. This is because the Alzain Program's encryption is based on the length of each record instead of the number of records. The Secret value that each record's SSA keys are generated from is the binary representation of the record's attributes, as described in section 5.2. The longer the record is, the more digits the key can contain. The reverse is also true: since each record is encrypted

independently, the record containing the fewest characters becomes the limiting factor for the minimum key length.

Because of the length of the Contact table's records, the Alzain Program starts out with a longer minimum key length than the Po1 Program in Figures 5.26 through 5.28. This changes around the 400 record mark, though, when the amount of records causes the Po1 Program's key length to match the Alzain Program's key length. After that point, the Po1 Program's security starts to dominate the Alzain Program's security.

The fact that the Alzain Program's security is tied to the smallest record's length is much more evident when examining Figures 5.29 through 5.31. The Po1 Program performs exactly the same in these tests as in the previous three tests, but the Alzain Program's key length changes quite drastically between the two sets of tests. The Alzain Program's minimum key length drops in the latter three figures of this section because it is encrypting nine-digit SSNs instead of full-length records. The Alzain Program still manages to meet the NIST's required 112-bit key length in these tests, but it is clear that the Po1 Program is much more secure when it comes to encrypting lists of at least fifty Social Security Records.

Another observation to be made from both sets of key-length graphs is that the minimum reported key length for both programs increases when more clouds are added to the encryption. This phenomenon is due to the fact that more clouds typically means a longer SSA polynomial, which in turn generates longer SSA keys. In general, adding more clouds to the encryption adds a level of security that goes beyond key length. Allowing

data to be split up across the clouds obscures the true nature of the user's data. It also mandates that an attacker not only know which clouds the user is storing data in, but also be able to overcome the security on each cloud before they can understand what the user has stored there. It is interesting that increasing the number of clouds in the encryption also leads to longer SSA keys in addition to all of the other security benefits that it provides.

It has been shown that the Po1 Program outperforms the Alzain Program in terms of security as long as the table being encrypted contains many records. But how much more security does it provide over the Alzain Program? The usual way to quantify the security of a key of *n* bits is to calculate the average time it would take to randomly guess and verify the key. The timing for this is based off of two encryption rates: one that is able to perform 1 encryption every microsecond, and one that is able to perform 1,000,000 encryptions every microsecond. At these rates, any secret 128-bit key is able to be correctly generated and tested in 5.4×10^{24} or 5.4×10^{18} years, depending on the rate of encryption [Stallings11]. This means that, for every 124 bits that the Po1 Program's minimum key length grows past the Alzain Program's minimum key length, on average it will take somewhere between 5.4×10^{24} and 5.4×10^{18} years longer for the Po1 Program's security to be decrypted than the Alzain Program's.

In addition to the security provided by the Po1 Program's distributed nature and key lengths, the Program's inclusion of fake records also adds an extra layer of security to the encrypted View. Because the fake records are added to the clouds in a pseudorandom fashion, they can be considered a form of cryptographic salt that increases the overall

decryption complexity. Cryptographic salts are rated based upon the average number of bits of randomness added into an encryption algorithm. In this case, the number of bits is determined by the length of the fake records inserted into the cloud. On average and by default, the Po1 Program will insert one fake record into the clouds for every real record inserted. Although the actual amount of security provided by this salting scheme will depend on the datatypes and the number of attributes being encrypted, it can very easily be seen that the Po1 Program's salt adds a large proportion of randomness to the encryption process, which will further increase its security.

Fake records would also add additional security in the form of an increased minimum SSA key length. Every fake record added to the Po1 Program's SSA scheme increases the length of the Secret value being encrypted, and therefore the minimum key length as well. If the Po1 Program inserts one fake record for every real record being encrypted, the SSA key length could potentially double. This, of course, would depend on where the fake records were inserted in relation to the real records, and the random nature of the SSA key generation process. Figure 5.32 gives a projection for this perfect scenario. In it, one fake record is added for each real record inserted into Figure 5.31's 9-Cloud Socials table, resulting in the SSA key length doubling. Extra time would have to be spent moving the fake records to and from the clouds, but the security provided by increasing the minimum SSA key length by 780 bits would more than make up for the increased data transmission time.

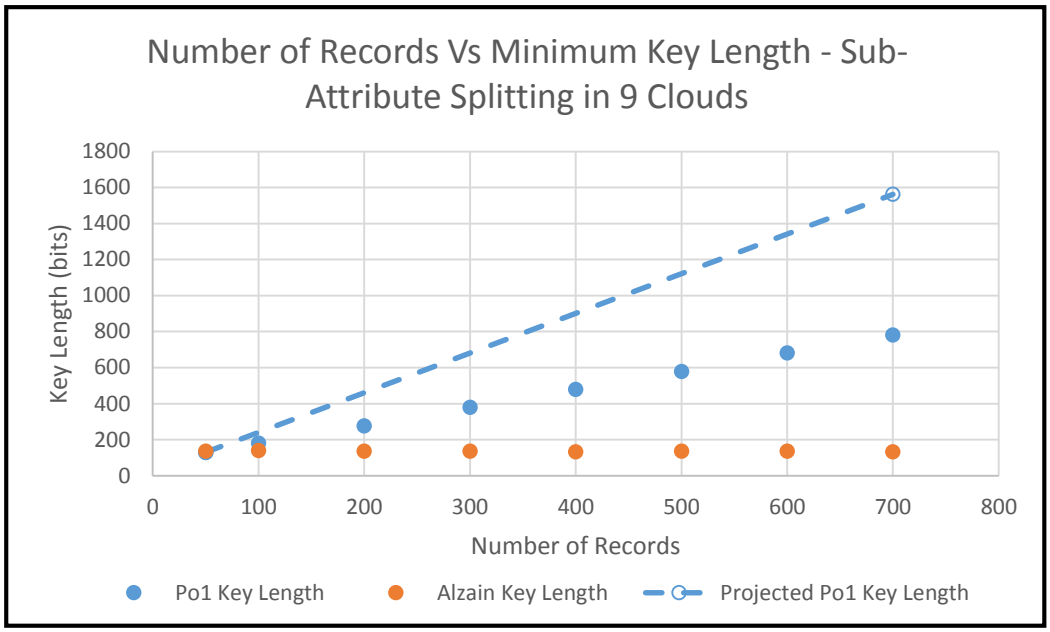


Figure 5.32 – Projection of the Effects of Fake Records on Figure 5.31's Po1 Key Length

Now that the merits of the Po1 Program have been quantified, Chapter 6 will summarize and conclude this thesis.

Chapter 6

CONCLUSION

This chapter provides a conclusion to this thesis by summarizing the results and findings of the previous chapter. Lastly, future works and additions are proposed.

6.1 Summary

Over the course of this thesis, a cloud database security algorithm was proposed and implemented. Unlike other algorithms, the proposed Hoeppner Security Algorithm (HSA) was designed to fragment both real and fake data across the clouds in order to increase the security of cloud databases and obscure the user's metadata footprint. This algorithm was implemented in Java as the "Po1 Program". The Po1 Program's efficiency was rated against the "Alzain Program," an implementation of the algorithm adapted from the paper "MCDB: Using Multi-clouds to Ensure Security in Cloud Computing." The Alzain Program did not provide users with the flexibility to define exactly how their data were split up across the clouds as the Po1 Program did, but it still proved to be a useful comparative methodology.

Many tests were run against the two programs. The first few tests determined how well the algorithms were able to split database records up amongst the clouds. The Po1

Program proved to be more versatile and efficient than the Alzain Program. It allows users to break their records up by attribute, groups of attributes, and by groups of characters within an attribute before placing each group of data into the cloud database of their choice. The Alzain Program, on the other hand, does not provide the user with nearly the same level of flexibility in how their data are distributed, and takes much longer to complete its encryption process. It proved to be more efficient at importing preexisting tables into its encrypted scheme, though, showing that some of the Po1 Program's increased security measures can lead to longer execution times. Another example of this trade-off comes when the Po1 Program creates extra cloud storage space to hold Secret Sharing Algorithm (SSA) encryption keys. The addition of more clouds and tables to the encryption scheme increases the Po1 Program's overall execution time, but also boosts its overall security by increasing the number of clouds that must be compromised for an adversary to decrypt the user's data.

The Po1 Program also explored the costs associated with adding extra clouds to an encryption scheme, and how the ratio of real to fake records affects execution time. In both cases, it was found that adding more clouds and fake records to the HSA is an easy way to increase the database's security without greatly influencing its execution time. Other facets of security were studied, such as how increasing the number of records being encrypted by the Po1 Program increases the amount of time a brute force attack would take. It was concluded that the Po1 Program is most secure when encrypting tables containing many records, while the Alzain Program is most secure when encrypting long records.

In conclusion, the Po1 Program has demonstrated that the HSA is best suited for the encryption of tables containing many records. Security gains are made for each cloud and record (both real and fake) added to the encryption scheme. However, the HSA requires an increased amount of storage space for its SSA keys every time more records are inserted. The HSA also isn't suited for quick table import procedures. Importing preexisting tables is not a typical use case, though, so the Po1 Program's slow performance is not as costly as it appears.

6.2 Future Works

There are many directions for the HSA to be expanded in in the future. The first is big data. All of the tests carried out in this thesis were on tables containing less than 1,000 records. It would be interesting to see if the HSA's efficiency will continue as projected as the number of records it processes is increased vastly.

The next addition that could be made to this thesis is the application of the HSA to groups of tables or full databases. So far, the Po1 Program's focus has been on encrypting and decrypting a single table. This was done intentionally because the table is the building block of the database – if the HSA can be applied to one table, it can be applied to any number of tables. The role key constraints play when encrypting multiple tables could also be studied.

Alternative importing procedures could be explored. The current table import procedure was chosen because it allowed each imported record to be broken down and inserted into

the cloud tables along with fake data in the same way that a normal Insert query is handled. More efficient algorithms for this process could be discovered, which would increase the HSA's overall efficiency.

REFERENCES

Print Publications:

[Alzain11]
AlZain, M., B. Soh, and E. Pardede, "MCDB: Using Multi-Clouds to Ensure Security in Cloud Computing," 2011 IEEE Ninth International Conference on Dependable, Automatic and Secure Computing, pp. 784-791.

[Cormen01]
Cormen, T., C. Leiserson, R. Rivest, and C. Stein, Introduction to Algorithms Second Edition, The MIT Press, Cambridge, 2001.

[Date04]
Date, C. J., An Introduction to Database Systems Eighth Edition, Pearson Addison Wesley, Boston, 2004.

[Delettre11]
Delettre, C., K. Boudaoud, and M. Riveill, "Cloud Computing, Security and Data Concealment," Computers and Communications (ISCC), 2011 IEEE Symposium on, pp. 424-431.

[Hevner79]
Hevner, A. and S. Yao, "Query Processing in Distributed Database Systems," IEEE Transactions on Software Engineering, SE-5, 3 (May 1979), pp. 177-187.

[Jadeja12]
Jadeja, Y. and K. Modi, "Cloud computing - concepts, architecture and challenges," Computing, Electronics and Electrical Technologies (ICCEET), 2012 International Conference on, pp. 877-880.

[Katz15]
Katz, J. and Y. Lindell, Introduction to Modern Cryptography Second Edition, CRC Press Taylor & Francis Group, New York, 2015.

[Lafore03]
Lafore, R., Data Structures & Algorithms in Java Second Edition, SAMS, Indianapolis, 2003.

[Li10]
Li, H., P. Liang, J. Yang, and S. Chen, "Analysis on Cloud-Based Security Vulnerability Assessment," e-Business Engineering (ICEBE), 2010 IEEE 7th International Conference on, pp. 490-494.

[Maryanski80]
Maryanski, F., "Backend Database Systems," ACM Computing Surveys (CSUR), 12, 1 (March 1980), pp. 3-25.

[Shaikh11]
Shaikh, F. and S. Haider, "Security threats in cloud computing," Internet Technology and Secured Transactions (ICITST), 2011 International Conference for, pp. 214-219.

[Shamir79]
Shamir, A., "How to Share a Secret," Communications of the ACM, 22, 11 (November 1979), pp. 612-613.

[Stallings11]
Stallings, W., Cryptography and Network Security Principles and Practice Fifth Edition, Prentice Hall, Indianapolis, 2011.

[Tanwer10]
Tanwer, A., A. Tayal, M. Hussain, and P. Reel, "Thin apps store for smart phones based on private cloud infrastructure," Kaleidoscope: Beyond the Internet? - Innovations for Future Networks and Services, 2010, pp. 1-6.

[Vaquero09]
Vaquero, L., L. Rodero-Merino, J. Caceres, and M. Linder, "A Break in the Clouds: Towards a Cloud Definition," ACM SIGCOMM Computer Communication Review, 39, 1 (January 2009), pp. 50-55.

[Zheng11]
Zheng, L., Y. Hu, and C. Yang, "Design and Research on Private Cloud Computing Architecture to Support Smart Grid," Intelligent Human-Machine Systems and Cybernetics (IHMSC), 2011 International Conference on, 2, pp. 159-161.

VITA

Joseph Hoeppner is an accelerated student at the University of North Florida in the five-year Computer Science program. He will receive both his Bachelor of Science and Master of Science degrees at the completion of the program.

During the school term, Joseph works as a tutor and Teachers Assistant for the School of Computing, holding office hours for students of all courses and levels. He is regarded by students as the best tutor in the School, and is much sought after. Over the summers, Joseph works as an intern for CSX Technology, where he has aided in data mining, software development, and community outreach initiatives. He will be starting as a full-time developer for CSX in 2016.

Joseph Hoeppner has received many accolades while attending the University. He has earned the School of Computing's Academic Excellence and Outstanding Achievement award at both the undergraduate and graduate levels, designed and created the award-winning Android game "Dots" at the School of Computing's Symposium, and has been featured on the University's Dean's List every semester. He is currently a member of five honor societies: the W.E.B DuBois Honor Society, the National Society of Collegiate Scholars, the Pi Mu Epsilon Math Honor Society, the UNF Honor Society, and the Upsilon Pi Epsilon Computing Honor Society, where he served as the Vice President from 2013-2014 and President from 2014-2015.