Honours Project Report

CSC4000W

Cry-Help

Tami Maiwashe

Supervised by Dr A.V.D.M. Kayem

	Category	Min	Max	Chosen
1	Requirement Analysis and Design	0	20	10
2	Theoretical Analysis	0	25	0
3	Experiment Design and Execution	0	20	10
4	System Design and Implementation	0	15	15
5	Results, Findings and Conclusion	0	20	15
6	Aim Formulation and Background Work	0	15	10
7	Quality of Report Writing and Presentation	10		10
8	Adherence to Project Proposal and Quality of Deliverables	10		10
9	Overall General Project Evaluation	0	10	0
	Total marks	**80**	**80**	

Department of Computer Science

University of Cape Town

2013

Contents

I Abstract

It recently came to light that more than half of the crime committed in South Africa is not reported to the police. Research has put forward that among the many complex reasons that people have for choosing not to report a crime to the police, one is that people much prefer to feel hidden (and so, in their perception, safe) when reporting a crime. This perception of safety is important, and if that assurance could be granted to community members, the police might see a greater number of people begin to report crime. News that the national police service's website was hacked and the personal details of whistleblowers exposed to the public very likely left the nation fearful and with a negative attitude towards reporting crime.

Access control has been a central topic of research for decades as various approaches to managing authorisations on data are suggested and debated. It is a process employed by an organisation to regulate its staff's access to its data. As much as it can be a powerful tool for managing access restrictions, direct attacks on database servers tend to bypass the access control model in place, finding and exposing a completely readable database.

As such, this project sought to create and evaluate a cryptographic access control scheme suited to the peculiar requirements of a police organisation's access control model. While the solution developed met these requirements, upon evaluation, it was found to perform poorly under the overhead of cryptography.

The report is unfortunately incomplete, and I send my sincerest apologies herewith.

II Acknowledgments

I would like to thank the Department of Computer Science for granting me the privilege of a place in its Honours programme: this has been a challenging but rewarding journey. I would further like to thank my supervisor, Dr Anne Kayem, on that note. This project was a maze of curveballs and confusion, but you kept steady focus throughout, which was encouraging.

Thabo and Nina, thank you for letting me work with you; the nights of slaving were made much more bearable when they were interspersed with the random laughs and new insight from you.

To Mum, Dad, Ofhi, and the family at large (the whole squad coming in December!), your love, prayers, encouragement and support paved this journey with such beauty where it should have been dreadfully scary. I can never, ever thank you enough, and I would not have been where I am today without you.

To all my friends whom my time here at UCT transformed into lifelong brothers and sisters, I saw the verse "A brother is made for adversity" come to life through you. Thank you for believing in me even when I was in my worst form.

Finally, and most certainly not least: God. You are simply amazing. For making all things work together for good at each point in this journey, for giving me ideas for the project when I was utterly distressed and close to resigned, for answering each and every prayer... Thank You. More than anyone else, this is for You.

1 Introduction

1.1. Motivation for the *Cry-Help* project

Earlier this year, the South African Institute for Race Relations released a report that highlighted how it is estimated that in 2011, more than half the crimes committed in the country were not reported to the police.

For many years, researchers have studied the behaviour of crime reporters in order to suggest a framework that they apply to decide whether or not to report a crime to the police. Admittedly, this decision tends to be influenced by people's perception of how responsive and helpful the police are, but Lasley's (1995) study suggests another factor in the crime reporter's decision. An experiment was conducted to measure how the medium of crime reporting affects the reporter's choice, comparing the widely-used telephonic approach with a computer-based approach. The results suggested that the extent to which a person feels anonymous reporting a crime affects their decision to do so positively – the more the crime reporting medium conceals their action, the more likely they are to report a crime.

Clearly, this is in stark contrast to the most popular methods of crime reporting today – phoning the police, visiting the police station or stopping at a Groote Schuur Community Improvement District (GSCID) box, for instance.

If we consider the most popular method of communication in South Africa, we find that the mobile phone has not only penetrated the country rapidly and widely, but its features have also advanced greatly over the years, from touch screens to gesture recognition, and from cameras to audio recording facilities to name but a few. The popularity and capabilities of this technology made it a good candidate for solving the problem of people opting not to report a crime simply because they do not feel safe doing so.

Cry-Help arose as an endeavour to see how best the possibilities that mobile phones offer today could be used in building a better crime-reporting system – one where people can report crime without having to fear being seen doing so. However, any crime-reporting system further needs to provide assurance of security to reporters, that is, the safety of the information that they will send to the police.

As such, this project had two objectives. The first was to build a prototype of a mobile crime reporting system with the following components:

- a mobile application with a user interface that supports discreet crime reporting;
- a way of transferring the crime reports to the police securely;
- a way of keeping the reports safely, protecting them not only from external intruders, but also from unauthorised viewers.

The second was to design the system to perform efficiently, while guaranteeing an acceptable level of security for the reporters and the crime reports. This objective was to be met by the last two components of the project. The measures "good performance" and "acceptable security" are defined and described in the respective reports.

1.2. Project description

This report documents the development of the third component of *Cry-Help* – that is, protecting the crime data from people who aren't authorised to view it, primarily within the police organisation but also in a way that prevents trespassers against the organisation's access restrictions from reading it.

Initially, my solution to this problem involved studying a database encryption scheme called *CryptDB* which allows queries to be performed on encrypted data, thus keeping the data unreadable on the database server at all times and protecting it from intruders. My intention was to see how I could improve the scheme. However, after facing difficulty installing the source code for the scheme and realising that there wouldn't be enough time to translate it from C++ to Java *and* implement an improved version, other options for the project's direction had to be considered.

Keeping *Cry-Help*'s objectives in mind, a decision was taken to design an access control scheme instead to solve the problem at hand. This decision was further supported by how several database encryption schemes that I had considered did not incorporate the access control model of the organisation whose data they aimed to protect. As such, upon querying the database in the context of *Cry-Help*, a police official would likely retrieve data that the organisation does not permit them to see, which would not be ideal.

1.2.1. Research question

The question this component of the project sought to answer is:

Can an access control scheme that allows both hierarchical and team access to data be designed with cryptography employed to further protect the data from outsiders, and how well does this scheme perform compared to a standard non-cryptographic solution?

1.2.2. System overview

The solution I arrived at was an access control scheme dubbed *CrAC*, short for "Cryptographic Access Control".

In addition to this scheme, a simple test bed system was designed as a simulation of a police database system. *CrAC* and an equivalent solution developed using SQL's standard constructs for privilege administration (`GRANT` and `REVOKE` statements) were then embedded therein, and it was used to evaluate the scheme's performance as different parameters were changed.

1.3. Report structure

In the next chapter, some background more specific to my part of the project is provided. In the following chapter, the solution's design is described and justified, and the test bed system's design is looked at briefly. In the fifth chapter, the implementation of both the scheme and the test bed system are reviewed, and thereafter, the experiment setups to test the scheme's performance are documented. In the next chapter, results of the experiments that were run are presented and discussed, leading to the final chapter, where the conclusion of the report is drawn and suggestions for future work on the scheme and systems it would be incorporated in are given.

2 Background

This chapter briefly looks at encryption standards that have been employed in the development of *CrAC*, and particularly at how secure they are.

2.1. Advanced Encryption Standard (AES)

The Advanced Encryption Standard (AES) is a symmetric key encryption algorithm that was recently chosen by the National Institute of Science and Technology (NIST) as the standard for symmetric encryption. It is a block cipher – that is, it divides data into several blocks of bits and enciphers a whole block at a time as opposed to transforming plaintext to ciphertext one bit at a time. It can be used to efficiently encrypt various sizes of texts (unlike, for example, asymmetric encryption algorithms that usually perform poorly when the size of the plaintext grows to more than a kilobyte).

2.2. RSA

RSA is a public-key cryptosystem against which presently no attack stronger than a brute force attack has been found. Like any other asymmetric encryption algorithm, it generates a pair of keys – the public and private keys, the first of which can be, as its name suggests, publicly known, while the latter can and should be kept as its owner's secret. If someone acquires a person's public key, they cannot retrieve the corresponding private key from it unless they discover how to factor large numbers, which is currently a difficult problem to solve. As such, RSA provides a nifty way for communicating parties to authenticate one another's identities and to convey a key under which further communication can be concealed, and all this using what could be public knowledge (the public keys of different people).

2.3. Password-based encryption

Password-based encryption is a technique whereby a string of characters is transformed into a symmetric key that can be used to encrypt and decrypt characters. Its security is generally not well-avouched as it ultimately depends on the user to create a good, unpredictable password, which is uncommon since users themselves naturally want to use a password that is not difficult to remember. However, the large potential benefit of password-based encryption is that the key material lies in what is arguably the most inaccessible vault: the mind of the key's owner. Unfortunately, this is probably just as great a downfall.

3 Design

In this chapter, the details and justifications for design decisions that were taken throughout the development of the solution access control scheme and of the system it was integrated with for performance testing are described. First, a description of what the requirements for the scheme and the functionality of the testing system were and how they were gathered is given, followed by a statement of the scope chosen for the project. Thereafter, the design considerations and choices for the scheme and the system are discussed, and the discussion is divided according to the main required features of the two.

3.1. Requirements elicitation

Since the project was not being developed for a particular customer, requirements gathering was a process followed to acquire an idea of how a police organisation might be structured, especially with respect to various individuals' access to its data. Avoiding the trouble that would likely rise in obtaining ethical clearance to use actual police organisational data, the first design decision taken was to use the Campus Protection Services (CPS) as a reference for developing the solution. Because the organisation's information system and personnel structure is much smaller than the national police force – which the *Cry-Help* project will hopefully be tailored towards in future work, a further design decision taken was to use CPS as basis from which a hypothetical police organisational structure should be extrapolated.

3.1.1. Organisational structure

CPS' personnel mainly comprises three roles: the capturer, the investigator, and the investigations manager. The database administrator was not given much mind in this project as the primary objective was to keep police data safe within the organisation, but future work should consider this role and which rights people filling it should have. The three roles mentioned are arranged in a simple hierarchy, with the capturer having the least access to data, and the investigations manager being able to access and update most, if not all, of the crime-related data.

Drawing from this, the organisational structure decided upon for this project was a role-based hierarchy where the employees higher up in the structure are able to access data that employees with lower rank than them have permission to view. Moreover, from experiential judgment, it was decided to allow teams of employees to share access to certain data (for example, generally when an investigation is opened, a team is assigned to work on it, and the simplest assumption to make about the related access permissions is that the team shares access to investigation-related data). Also, the number of levels in the hierarchy was used as a parameter in testing the efficiency of the solution scheme's efficiency, so CPS' three roles were spread out over n levels (for example, levels nine to thirteen could all comprise capturers, just with some being subordinate to others). The illustration below depicts how CPS' hierarchy was transformed to arrive at the one this project suggested a solution for.

3.1.2. Functions of the system

Looking at CPS' information system, it appears that the main functions it performs are the basic activities one would expect: adding, updating, removing and viewing data. Furthermore, the data they store and work with can be divided into three sets: staff-, incident-, and investigation-related.

Staff-related data is likely the smallest set, simply storing the personal details of staff members. Incident-related data is very broad, recording details that describe various incidents that have been reported to CPS, such as vehicles spotted and suspects' appearance. Lastly, investigation-related data store information such as investigation diaries and the progress of different investigations.

As a result, the testing system was designed to perform "create, read, update, delete" (CRUD) operations on a basic version of these three sets of data, with exceptions where access restrictions would not allow the operation (for example, incidents and investigations should not be deleted; they should rather be archived).

For staff-related data, a subset of personal details was chosen (including simple details, such as one's name and contact details, and importantly, their job position). For incident-related data, another straightforward table was designed with just enough attributes to describe an incident (for example, the place, date and time of the occurrence). Finally, for investigation-related, a table portraying a high-level overview of the organisation's investigations was planned. The entity-relationship diagram below is a model of the data that was chosen to simulate, on a small scale, the information that a police organisation could work with.

3.1.3. Classification of data

The three sets of data chosen required that a solution access control scheme deal with three levels of data privacy: private (or personal), for a group of employees, and public (unclassified).

3.1.4. Access restrictions

Because separation of duty was not strongly present in CPS' structure (for instance, the investigations owner could create, remove, and update any investigation they please without it being verified by anyone else), for this project, access restrictions had to be modelled such that any employee should have to acquire permission to update an investigation that they had created, for instance. The table below *(Table 1)* summarises the access restrictions chosen for which a suitable access control scheme was to be designed.

ROLE	Staff data	Incidents data	Investigations data
Investigations manager	Can view one's own personal details, and a limited number of	Can view all incidents; cannot delete, update or insert incidents	Can view all investigations and can open a new one; Cannot delete or update

	fields for other staff members. The attributes chosen to be kept private were: identity number, cell phone number, and residential address details		any investigation; Can view members of an investigation team; Cannot edit this list in any way though
Investigator	(same as above)	Can view all incidents; cannot delete, update or insert incidents	Can view all investigations that one is a head of, those whose teams one is a part of, or those whose heads are subordinates to oneself; can update investigations that one is a head or member of; Cannot delete or insert an investigation; Can create a record of an investigation's team members and manipulate howsoever one wishes (insert, delete, update or view)
Capturer	(same as above)	Can insert a new incident, can view only those incidents that one captured oneself, those related to investigations whose teams one is in, or those which one's subordinates captured; Cannot update or delete an incident	Can view and update all the investigations of which one is a part; Cannot insert or delete an investigation; Can view members of an investigation team; Cannot edit this list in any way though
Unassigned role	Can only view one's own details	Cannot view any of this data	Cannot view any of this data

Table 1: The access restrictions that the solution was required to implement

3.2. Statement of scope

This project aimed to design, develop and test (for good performance) an access control scheme for a police organisation that employs a role-based hierarchy to describe its personnel structure, and further allows authorised teams of employees to share certain data,

all the while keeping classified data safe from both external and unauthorised internal viewers.

Secondary to this, another aspect of the scope was to design and develop a small simulation of such an organisation's information system in order for the access control scheme's performance to be evaluated. This simulation would perform the CRUD operations on a simple set of data representative of the information the organisation would likely work with.

It is stressed that as much as database security is roughly five-fold, issues such as integrity of data, authentication of people manipulating the data, and non-repudiation were not focussed on. Instead, the solution was designed to control access to the data and to maintain its confidentiality specifically in that regard. However, in a later chapter, ways that the access control scheme could be extended to cater for some of these issues are suggested.

3.3. *CrAC*: overview of design

3.3.1. Dependent keys versus independent keys

As the previous chapter highlighted, most hierarchical access control schemes that have been proposed make use of dependent keys. However, the chapter further described how this approach can have disadvantages. If we think of the dependent keys used in an access control scheme as a chain, breaking the chain in any way – that is, adding a level to the hierarchy or removing one, adding a user to a level or removing them – tends to require a large re-computation of keys, almost like one piece of the broken chain being replaced altogether. Moreover, for the peculiar access control model that a typical police organisation could use (one where team-sharing and hierarchical access are both present), a dependent-keying scheme would not allow sharing exclusive to a group of employees. Consequently, the solution access control scheme was designed using independent keys.

3.3.2. The challenge of protecting persistent data

Furthermore, the scheme was being designed to safeguard persistent data. In past studies, independent keying tends to be used in a context of dynamic access control. Take, for example, a chat application where not only can people communicate individually, but groups can share messages too. Generally, in the group scenario, a session key will be created and shared securely with the group members, and messages passed in the group will then be enciphered using the session key. When the group chat ends though, because the data is non-persistent, the session key will be lost, and if the group is to start another chat, an altogether new session key will be created and used.

In the case of persistent data, this cannot work, because when new keys are generated, that translates to decrypting data with the old encryption keys and re-encrypting it with the new keys. This is not ideal because encryption and decryption are expensive operations that additionally contribute nothing to a system's performance. The scheme thus had to use independent keys in a way that would allow the data stored to be retrieved from their encrypted form over and over again. It also had to do this attempting to minimise the number of encryption/decryption operations in the system.

3.3.3. Integrating the design decisions

In summary, the solution access control scheme works as follows: upon being registered on the system, each user is assigned a set of four personal keys, namely, a password-based key (based on a secret password chosen by the user), a symmetric key, and a public and private key pair. To encrypt personal data, one simply uses their symmetric key. For classified data, however, a little more work is required as one user may have to share different data with different groups of people, and if both sets of data are to be encrypted with the same key, all the people with whom at least one of the sets is shared will indirectly have acquired access to the other sets that they aren't necessarily authorised to view. Resultantly, when a new record of classified data is created, a random eight-digit tag is generated and stored with it, and the fields of the record that need to be protected are encrypted under a new key formed by enciphering the tag using the symmetric key of the record's owner.

The principle of a person owning a record in the database fit the police organisation's context well, because the hierarchical structure persists even when investigations are opened, for instance. As such, it was feasible to assign a record to one person's ownership. This idea received more support from CPS' structure as well, where investigations have a one-to-one relationship with investigators.

Using a random tag to create different keys for different records from one key required that a strong encryption scheme be employed – one that does not allow the encryption key to be discovered even with a large number of plaintext-ciphertext pairs to aid the attack (because if the symmetric key were to be recovered, the user's private data would be left vulnerable to disallowed viewing). Thus far, the Advanced Encryption Standard (AES) provides this guarantee, and so it was chosen as the method of encryption and key derivation.

The need for the private and public keys will be explained in the next section, where the test-bed system's design is presented.

3.4. Test bed system: an overview of the design

The test-bed was designed to be a client-server system where clients could communicate with the server to obtain data from the database, and the server also took the role of intermediary, relaying requests from one client to another and the responses too.

All in all, it performed the following functions:

- *Staff management:* adding and removing a staff member to the system, updating staff members' details, and viewing them
- *Incidents management:* capturing a new incident and viewing the details of a specific incident or all the incidents in the database
- *Investigations management:* opening a new investigation, updating the progress of an investigation, and viewing the details of a particular investigation
- *Hierarchy management:* adding and removing a level from the organisation's hierarchy of job positions

3.4.1. Storing the encryption keys

Designing the system, it became apparent that the first important question to answer was the one of the keys' storage: which of the keys would be kept? Where would they be kept, and how?

Working on the assumption that every employee has a computer in the organisation's network that they are assigned to, it was decided that each user's symmetric key and asymmetric key pair would be stored in a text file on their computer, the symmetric key encrypted under their password-based key, and the private key likewise under their symmetric key.

In the next sub-section, as previously promised, the purpose of the asymmetric keys is explained, but for now it suffices to say that the public keys needed to be stored in a central place that all the users could access (to avoid the steep storage and synchronisation requirements that could come from every user keeping their own record of staff members' public keys). As a result, a list of users' public keys was kept on the server, and that accessibly to all.

3.4.2. Getting an encryption key

Since independent keys were used to implement the access control scheme, consideration had to be given to how an employee would obtain the key needed to encrypt or decrypt data. Moreover, how would the system ensure that only users authorised to view certain content would get the key to decipher that data? This is where the asymmetric keys were used. An easy protocol was designed to authenticate requesters and senders of encryption keys, and it works as follows:

- Person A requests a key from the owner of the record that they wish to view (Person B). This request is encrypted under Person B's public key.
- Person B uses their private key to decrypt the request, and if they permit Person A to view the record, they send the decryption key encrypted under Person A's public key.

Because of public-key cryptography's fundamental ideas – that the private key cannot be determined from the public key and that the private key is a secret that only its owner knows, this straightforward protocol assures the communicating parties of each other's identity.

RSA was the public-key cryptographic standard of choice. Diffie-Hellman seems to be more useful for creating and sharing temporal keys, which clearly does not fit this context.

3.5. Summary

This chapter presented a collection of the important design decisions taken in planning the access control scheme and the testing system it would be implemented in. It began with a description of the hypothetical police organisation that the solution was designed for and of how this organisation's structure was arrived at. Thereafter, the scope of the project's work was given, and emphasis placed on the desired qualities of database security that the project aimed to achieve. Lastly, the designs of the solution access control scheme and the test bed system were outlined, justified against the main features that were required of them. The next chapter details how this design was translated to code.

4 Implementation

This chapter is an account of how the design in Chapter 3 was implemented. The tools and techniques used in doing so are described, as well as the challenges faced that hadn't been anticipated in the design process. The workarounds or solutions to these issues are documented. Reasons for approaches taken or disqualified are also given.

4.1. System development environment

4.1.1. Programming languages

The access control scheme and the test bed were implemented in Java, mainly because it is the programming language I've had the most exposure to. C was a large contender because of how well-optimised a program written in it can be for the machine it runs on, and this would very likely improve the scheme's performance as it relies heavily on cryptography, which requires much computation. However, given the short period of time to develop a solution (moreover considering how the first few were spent on a dead-end path), the steep learning curve for the language far outweighed the potential benefits of using C. Also, since the performance boost offered by C seems to not be significantly large anyway, it was decided that the Java implementation of the scheme could act as a relative measure of how well the scheme performs, as opposed to a definitive indication. This was further supported by the decision to develop a parallel system that used standard SQL `GRANT` and `REVOKE` statements to implement access control in Java as well for comparison.

On that note, SQL was the language of choice to interface the database system for a similar reason: not only is it the most popular language for database manipulation, but it is the one I am most conversant with.

4.1.2. Database management system

MySQL was picked as the database management system (DBMS) for the test bed as it is one of the most widely-used DBMSes worldwide. Furthermore, because the solution was implemented in Java, Java Database Connectivity (JDBC) was used to access and manipulate the database from within the mock application.

4.2. System functionality

The test bed system was implemented as a simple application that offered the functionality listed in the previous chapter (staff, incidents, investigations and hierarchy-management), and two access control schemes were incorporated therein. One just converted the organisation's access restrictions to equivalent SQL statements, while the other – the proposed solution – employed a combination of SQL statements and cryptographic elements to administer access rights.

4.2.1. Scheme A: SQL statements only

The access control scheme developed using SQL `GRANT` and `REVOKE` statements to administer database table privileges achieved all of the requirements outlined in *Table 1*. What follows is a discussion of how it was implemented, and this is divided among the four main tasks of the application, namely, staff, incident, investigation and hierarchy management.

4.2.1.1. Staff management

As required, every staff member with a role assigned to them was able to view a limited profile of all the employees. This was accomplished by granting every user `SELECT` access only to the attributes of the limited profile in the `Staff` table. Each staff member with a role was further permitted to view their full profile. To permit this, upon creation of a user on the system, a `view` associated with their full profile was created too, to which the user was granted a `SELECT` privilege on all the `Staff` attributes. By additionally granting the new user an `UPDATE` privilege on this `view`, the scheme ensured that each staff member could update only their personal details. Staff members without a role were not given any `SELECT` privilege on the `Staff` table at all, but were given `SELECT` and `UPDATE` permissions for their personal profile. As such, the scheme succeeded in guaranteeing that such a staff member could just view and update their own profile and not even a limited profile of anyone else.

4.2.1.2. Incident management

The scheme was successful in allowing only capturers to record new incidents, and furthermore allowing investigators and investigation managers to view incidents (since incidents will always be recorded by capturers, those above them – that is, the investigators and investigations managers – should also always be allowed to view them as per the hierarchical access control requirement for the scheme), while forbidding any updates and deletes by anyone in the hierarchy. Again, this was achieved in a straightforward manner, granting users with those roles the relevant permissions on the `Incidents` table, and simply not granting the `UPDATE` and `DELETE` rights on it to anyone.

Moreover, the access restrictions in *Table 1* required that a capturer be able to read a record of the `Incidents` table only if they captured it, a subordinate to them captured it, or they are in the investigation team for that incident. Before discovering how great a degree of row-level access control SQL's `views` offers, this requirement presented a problem. The cause for the difficulty experienced in implementing it using just `GRANT` statements on the `Incidents` table was two-fold:

- a user can only grant permissions that they themselves possess;
- granting record-level access to someone requires that a view tailored for them be created.

Now, we must bear in mind that the system permits only a capturer to record a new incident and that according to the desired access control model, capturers will have varying viewing rights for the `Incidents` table depending on their rank or what investigation team they are a part of. These viewing rights would have to be administered dynamically: as an incident is captured or as a capturer is added to an investigation team, permission to view the

relevant incident would need to be granted. If every capturer were to grant each of the capturers above them the `SELECT` permission on their `view` of the `Incidents` table, each capturer would likely end up having too large a number of `views` associated with them, and these would be difficult to access and navigate from the mock application. *Illustration 3* is a diagrammatic depiction of this problem.

Using `views` once more though, the issue was swiftly overcome. A `view` on the `Incidents` table was created for each new capturer and the incidents-related restrictions put forward in *Table 1* for a capturer were specified in the `view`'s definition to limit each capturer's reading rights correctly. `Views` were also a very apt solution since they are not static structures, but are instead updated dynamically as the underlying table on which they are based is modified. This proved a superior and more manageable approach to the former one, where the number of views associated with each user would likely have grown at a very fast rate.

4.2.1.3. Investigation management

Encountering a similar problem to the one reviewed in the previous sub-section, enabling dynamic administration of reading rights on different records in the `Investigations` table proved to be challenging at the outset. Again, because investigators own the records in the table, they would either need to be allowed a database-wide permission to `INSERT` rows in all tables at the time that they are created on the system (in order for them to be able to update investigation team members' `views` on the `Investigations` table at some point in the future when they are added to the team, for instance), which would, as aforementioned, be a risky authorisation to grant. Once more, to deal with this issue, a new `view` on the `Investigations` table was created for each investigator and capturer based on the rule the `view`'s owner either have a higher job position than the head of any investigation in question or they be in the investigation team parallel to it.

The other required access restrictions were also successfully implemented. To manage the updates investigators could make to the `Investigations` table (that is, only being able to update investigations that they own), a `view` of the table was created for the investigator as they were added to the system, and this `view` was designed to contain only those investigations which that investigator would be appointed to head. The investigator was then granted an `UPDATE` privilege on that `view`.

4.2.1.4. Hierarchy management

The main requirement here was the when a level was added to or removed from the hierarchy, employees' access restrictions would be updated accordingly. This was achieved by blindly revoking the privileges of the staff on and below the level being added or removed, changing their rank in the `Staff` table, and re-assigning privileges as appropriate to their new positions. This approach was chosen (as opposed to first checking if each employee's authorisations are actually affected by the change) because commonly, different levels in a hierarchy correspond to different job titles (this is just not the case here because CPS' small hierarchy of three roles was spread over many levels), so an employee whose rank is affected by the hierarchical shift would very likely have a new set of privileges.

4.2.2. Scheme B: SQL statements with cryptography – the proposed solution

The preceding section outlined and elaborated the main implementation details of the access control scheme that *CrAC* would be benchmarked against. This section does the same for the solution scheme, again spreading the discussion over the different access control requirements outlined in *Table 1*, as well as how the other design considerations that came about as a consequence of employing cryptography in the scheme were implemented. Most of the `GRANT` and `REVOKE` statements that were used to manage staff members' access rights were used in this scheme too (for example, the `INSERT` and `UPDATE` permissions remained exactly the same). The largest difference between the two schemes was *CrAC*'s departure from `views` as a way of providing record-level access permissions to encryption, so the administration of record-level access is what this section focuses on.

4.2.2.1. Client-server architecture

Since *CrAC* assigns independent keys to users of the system, one's key cannot be used to derive another's in order to acquire access to protected data. As such, the solution scheme required that users be able to request encryption keys of each other, and so it had to be implemented in a client-server architecture, where users could communicate with each other through the server. A peer-to-peer (P2P) architecture would likely have proven a more scalable solution as the server would then not pose the risk of bottlenecking communication. However, the difficulty in administrating such a setup (for example, ensuring that no unauthorised key exchanges occur among peers) outweighed its benefits.

The client-server setup is simple: when a user logs in, a `ServerThread` dedicated to waiting on the user's requests is forked, and it will listen for their instructions until the user enters the exit command. To allow communication between clients, the server keeps two lists of sockets: one for the server's output and input streams from and to each logged-in user (`ActiveNormalConnections`), and the other for its output and input streams from and to each logged-in user's peer requests administrator (`ActiveP2PConnections`) – the description and necessity of which are given after the next section. For now, it is adequate to say that the `ActiveP2PConnections` are what the server relays requests for keys through, while `ActiveNormalConnections` are the channels it uses to communicate responses to the requesters.

A table at the end of the chapter *(Table 2)* lists the requests from users that the server handles with a brief description of how it responds to each.

4.2.2.2. Key management

4.2.2.2.1. User key generation

Upon creating a user on the system, as described in the previous chapter, a set of four keys is generated for the user: a password-based key, a symmetric key, and an asymmetric key pair. The password-based key is generated using MD5 and DES, as recommended by RSA's PKCS5 standard. The symmetric key is an AES key, for the sole reason that AES is currently recommended as the most secure symmetric encryption algorithm. Lastly, RSA was chosen as the public/private key pair generation algorithm for its similar globally-recognised merit.

The password-based key was created with the single purpose of `wrapping` the symmetric key, which in turn `wraps` the private and public key, and in this form, the user's symmetric and asymmetric keys can be communicated to them safely (that is, even if they were to be intercepted by some attacker, the attacker would not be able to make any use of them as they would be in encrypted form). Since the user can regenerate the password-based key on their machine by entering their password, this set of keys can be recovered safely on the client side and stored.

At first, it seemed that the public key could just be sent to the user in its encoded form (an array of bytes) and the key regenerated by the user using those bytes when it was needed. The certificate to be stored in the keystore was generated using Bouncy Castle's `X509v1CertificateBuilder` class, and an attempt was made to use Java's `X509EncodedKeySpec` class on the client side to regenerate a user's public key from its encoded form. However, it became apparent that the key being returned by this process was not correct, as text encrypted under it would still be undecipherable under what was meant to be its corresponding private key.

This approach was then abandoned in favour of Java's `wrapping` function. It is a method in the `Cipher` class that allows one to encrypt a key under another key, and the `unwrap` method can then be called to retrieve the key correctly from the cipher.

4.2.2.2.2. Key storage

Because users need to access one another's public keys for secure communication of requests (this is elaborated on in the next section), users' public keys are stored in a `.keystore` file on the server, which can be modified by way of a `KeyStore` object. Since the `KeyStore` class only allows key pairs to be stored as a certificate and a password-protected private key, self-signed certificates are generated for each new user as they are created on the system, and their private key is protected by their login password. Because authentication was not one of the main objectives of the project, self-signed certificates sufficed as assurance of users' identities to one another.

The keystore is loaded once the server application starts, and its password should ideally only be known by a few people in the hierarchy, if any at all (the project assumed that the database administrator would be the only one with knowledge of the keystore's password). A reason for limiting access to the keystore was so that adding certificates for non-existent staff members would be prevented and so the risk of fake user accounts being used to gain unauthorised access to the organisation's data diverted.

Once the user's set of keys is generated, the set is sent to them as described in the previous section, and because the assumption is that each user has a particular computer assigned to them in the workplace's network, a text file stores this set of keys as strings of bytes. They are stored in their encrypted form, and the initialisation vector used in enciphering the private and public keys is also stored alongside them.

4.2.2.2.3. Generating the encryption key for records

As briefly presented in the previous chapter, the key to encrypt a tuple in the database was created by encrypting a random tag assigned to that tuple under the symmetric key of the

record's owner. This was chosen because AES is secure against known plaintext and known ciphertext attacks, so even with the tags stored unencrypted, knowing the tag and the corresponding encryption key would not be sufficient to recover the symmetric key of the record's owner. Since only the tuple's owner would need to access the tag anyway (the rest of the people with viewing rights for the tuple would get the decryption key from the owner), this scheme could be improved to erase the worry of generating so many plaintext-ciphertext pairs for a single key by encrypting the tag under the owner's public key so that officially only they can access it.

This approach (encrypting a random number and using that as a key) was taken instead of creating an independent key for each record or subsets of records readable by one group of people, because on that path, the number of keys in the system would greatly multiply, and the keys would thus prove difficult to keep safe and share. An instance of how this explosion of keys could occur is shown in *Illustration 4*.

4.2.2.3. Authorisations

Where Scheme A used `views` to provide record-level access to data, *CrAC* achieved it using encryption. Encrypting different data under different keys enabled read authorisations that span a table to be granted at user creation time (so for instance, a capturer could be granted the `SELECT` privilege on the entire `Incidents` table) but still keep parts of it hidden from them. The critical feature to implement was the extra layer of authorisation that would determine whether or not a user can access a certain subset of data in a table that they have viewing rights for.

The authorisation subsystem was implemented based on the idea of data ownership – the idea that each record in the `Incidents` and `Investigations` table has an owner whose permission to read the tuple can be requested by anyone in the hierarchy and this request processed to return either a decline or the key to unlock that data. As briefly described earlier, each user had a separate connection to the server to receive such requests on, and this connection would be enlisted in the Server's `ActiveP2PConnections`. This connection was opened in a different thread to the user's main thread (where the user would be performing the "normal" actions – recording incidents, updating investigations, and so on). The user was tasked with periodically running their `P2PAdminManager` (the separate thread that handled peer-to-peer requests) to attend to authorisation requests.

Some requests required no active consent from the user themselves – the computer would just check whether the requester is either higher in job position than the record's owner or in the investigation team related to the tuple in question. This was the case for most of the requests dealt with (for example, requests for the decryption keys to decipher incidents and investigations). Others required that the tuple's owner explicitly agree to grant the requester what they had asked for. This was the case, for instance, with job position updates and being assigned to an investigation. Here the power of encryption was extended to illustrate how even more rigorous separation of duty could be implemented, diffusing the extent of any one person's authority in the hierarchy. All these instances are described further in sections to follow but here served the purpose of depicting how sharing of encryption keys was managed in the system.

The `P2PAdminManager` was implemented as a thread as it would have been confusing to have all the requests sent to a user channelled into the same stream as the server's main line of communication with the user. Because the requests would not be "synchronised" with the user's personal communication with the server, at some points, the user would very likely receive a peer-to-peer request in the place where they were expecting a response to their own query to the server. The approach taken in the solution is certainly flawed, the largest fault being that in the scenario where a user does not have any pending requests but unknowingly decides to check if they do, the application will hang until a request is received. So for the system to work smoothly, a user is expected to only check for requests when they know there is one to attend to – a very unrealistic expectation.

A different approach that could've been taken to solve the problem is to cater for the possibility of reading a P2P request instead of a response that one is anticipating in one's personal communication with the server. This approach initially seemed daunting because it looked like it would require an `if-else` pair for every message read in from the server. However, if the `BufferedReader` wrapping the user's input stream from the server were to be subclassed and the `readLine`() method modified to differentiate between personal and P2P communications, attending to P2P requests would easily become integrated in the user's interaction with the application without interfering too unexpectedly and yielding incorrect results as would be the case if the `P2PAdminManager` was left free to pop P2P requests on the user's screen as they came in.

4.2.2.4. Staff management

As an obvious choice, the owner of a tuple in the `Staff` table was considered to be the employee whose personal details the tuple recorded. Instead of using a SQL `GRANT SELECT` statement to limit what fields of one's personal profile other users could see, their viewing was restricted by encrypting the sensitive fields under the owner's symmetric key. So if a user attempted to view someone else's personal details, the system would attempt to decrypt the person's profile under the user's symmetric key and, because it would fail to do so, return one of `null` or garble, as the screenshot of the application shows in *Figure 1*. Unfortunately, this method could not differentiate between an employee with a role and one to whom a role hadn't been assigned yet, so these could view other employees' limited profiles alike – a violation of an access restriction stipulated in *Table 1*. This was addressed by not granting "roleless" employees read rights on the `Staff` table, but only on a `view` of the `Staff` table that contains just their record of personal details.

4.2.2.5. Incident management

The main access control requirement with respect to incidents was that any given incident only be viewable to employees higher in position than the incident's capturer or to employees who are a part of the related investigation team. Again, encryption was used to supply legal views of the incidents to each employee. The administration behind the management of investigation teams in the database is described in the next section, but the primary idea of it is that every investigation has an attribute to store a comma-separated list of its team members, and in addition to checking the requester's position in the hierarchy, the investigation's `team` field was referred to in order for the system to decide whether or not a requesting party should be granted the key to decipher the incident's data.

Permission to view a record was granted by way of the decryption key being sent to the requester, enciphered using the requester's public key to ensure that only they can retrieve it.

4.2.2.6. Investigation management

Similar to incident management, investigation management was required to filter `SELECT` rights to only those employees whose rank exceeds the position of the investigation's owner or who form a part of the investigation's team. As outlined briefly above, access granted on the basis of the requester being in the investigation's team was administered with the help of the `team` attribute in the `Investigations` table.

4.2.2.7. Investigation team management

Managing the investigation teams (adding and removing members to and from a team) was an important step in achieving the required access control to investigations and incidents.

At first, the idea was to create a table for each investigation's team upon the investigation being opened (entered in the database). This would store a list of the team members' staff numbers. However, because the incidents' identity numbers – and so the investigation numbers' too – were originally being generated randomly (with no intent really), these tables could not be created as the corresponding investigation was opened because that would require the person opening the investigation, i.e. an investigations manager, to have the `CREATE` privilege for the database. Furthermore, because the investigator in charge of the investigation would ultimately be the one with the responsibility of modifying the investigation team's table, the investigations manager creating this table would need to have this `CREATE` privilege with the `GRANT OPTION`. That way, they would able to grant the privileges necessary for modifying the investigation team tables to the appropriate investigators. This would be a very dangerous assignment of privilege as this would allow investigations managers to create any table they please and dispense rights on it to people, and these could be used as a platform for unauthorised sharing of private organisational data.

The next idea was to pre-initialise these investigation team tables: to create them when the other tables (`Staff`, `Incidents`, etc.) were created and to grant `INSERT` and `DELETE` privileges on them to all investigators managers with the `GRANT OPTION` so as to enable them to endow the same privileges to the investigators assigned to the corresponding investigation. Since each investigation team table's name would match the identity number of the investigation, generating these numbers randomly would not permit this pre-initialisation, so it was decided that the incident/investigation numbers would be determined using SQL's `AUTO INCREMENT` operation.

But because in South Africa, about two million crimes occur in a year, it was estimated that at least a million crimes would be reported to the police per year, and so a million investigation team tables would need to be created annually. Just as the 695th table was created, realising that more than nine-hundred thousand tables still needed to be made, another plan had to be thought of.

The obvious solution then emerged: to have a `team` attribute in the `Investigations` table as briefly discussed in a previous sub-section.

4.2.2.8. Hierarchy management

Hierarchy management was implemented as a task that, again, a DBA could perform as these hierarchical shifts would involve revocation of old privileges and granting new ones suitable to the new job positions assigned.

In *CrAC* however, and specifically in the instance of a staff member being demoted or entirely removed from the hierarchy, this reassigning of privileges also included re-encrypting certain incident and investigation data that the user was no longer allowed access to, and in some cases, first re-assigning the data to new ownership (for example, if the user was heading a investigation and they were fired, data related to that investigation would need to be assigned to another investigator and re-encrypted accordingly). To this effect, the procedure followed to find the incidents and investigations that need to be re-encrypted when a staff member's rank is being lowered is this:

- all the incidents captured by and investigations headed by people who were subordinates to the staff member and whose rank is now higher than theirs are re-encrypted;
- if the staff member was a capturer, any incidents captured by them are re-assigned and re-encrypted accordingly, and likewise in the case of investigations where a staff member was an investigator.

A similar process was followed in the case where a staff member was being removed from the hierarchy entirely. Then they had to be removed from any investigation teams they were in and the related data re-encrypted accordingly, the data owned by their subordinates had to be re-encrypted too, and finally, any data owned by them was re-assigned and re-encrypted.

4.2.2.9. How the server fits in

Below is a table listing the commands that users send to the server to accomplish the tasks described in this section so far *(Table 2)*. Alongside each command is a note on the actions that the server takes in response to the commands.

Command	Description of server's response
`ADD TO KS`	**This command is sent when a new user is being created on the system. It serves as a signal for the server to generate the user's set of symmetric and asymmetric keys, and these are communicated back to the user as described in Section 4.2.2.2.1.**
`CLOSE KS`	This command notifies the server of the user's intention to store the keystore. This generally happens when the database administrator is done adding or removing staff members to or from the system.
`DEL FROM KS`	**This command serves to tell the server to remove a staff member's key entry from the keystore, and this is done when someone is being removed from the hierarchy.**

EXIT	True to its name, this command tells the `ServerThread` attending to a user's requests to stop doing so. The thread shuts down as specified by the program: it closes and removes the user's `ActiveP2PConnection` and `ActiveNormalConnection` from the `Server`'s lists thereof, effectively logging them out of the system.
INITIAL REQUEST FOR INVESTIGATION ENCRYPTION KEY	**This command is sent when an investigations manager is opening a new investigation and needs to encrypt the record under a key derived from the assigned investigator's symmetric key. A parameter sent with this command is the random tag to be stored with the investigation. The server relays the request to this investigator, who then encrypts the tag using his symmetric key and sends it back to the investigations manager through the server.**
IS LEVEL ONE EMPLOYEE ONLINE?	A user typically sends this command when they need to check whether an investigations manager is online to authorise some action. For example, when a level is added to the hierarchy, investigations managers need to be contacted to find out how to re-assign data that belonged to people who, as a result of the change, lost their positions. The server responds with a "yes" or "no", and from there, the client either proceeds to make a further request or checks later for the availability of a Level 1 employee.
LOGIN TO SYSTEM	**This command is sent as part of a user's login process. Since users' passwords are salted, this command tells the server to return the user's password so that their identity can be verified correctly.**
P2P ADMIN CONNECTION	This command is also sent as part of a user's login process. It informs the server that the connection made is one for receiving peer-to-peer requests. Consequently, the server stores its output and input streams to write to and read from this connection in the `Server`'s list of `ActiveP2PConnections`.
REPLY TO MESSAGE	**This command serves to tell the server that the user is responding to a request, and is followed by the staff number of the intended recipient, the number of parts to the message being sent, and the message itself. The `ServerThread` then writes this message to the recipient's `ActiveNormalConnection` as they will be waiting on this response.**
REQUEST FOR INCIDENT ENCRYPTION KEY	A user issues this command when they need the capturer of an incident to send them the key to decrypt the data. The server sends this request to the capturer

	on their `ActiveP2PConnection`.
REQUEST FOR INVESTIGATION ENCRYPTION KEY	**Analogous to the above, this request is channelled to the relevant investigator.**
`REQUEST FOR NEW DATA OWNER`	This command is sent when data is being assigned to new owners (like in the example where an investigator was fired and a new one had to be chosen to lead it).
REQUEST FOR NEW ENCRYPTION KEY	**In the same strain, this command requests a new encryption key for an incident or investigation that needs to be re-encrypted.**
`REQUEST PUBLIC KEY`	Lastly, this command is sent frequently by users in order to encrypt parts of their communication with other users and thereby verify their identities.

Table 2: A list of the commands that the server handles with a brief description of how it responds. They are mostly related to the administration of the users' keys – particularly the generation, assignment and secure communication thereof between users.

4.3. Summary

This chapter detailed how the design described in Chapter 3 was applied to implement the solution scheme, as well as its equivalent minus the cryptographic elements. It explored the challenges encountered and how they were solved. The major emphasis in both schemes was providing record-level access control to tables in the database, and where one achieved that using `views`, the cryptographic solution used encryption instead. The next chapter outlines the experiments conducted to evaluate how well *CrAC* performed against its benchmark.

5 Experiment Design

This chapter is a write-up of the different experiments that were conducted to assess *CrAC*'s performance. The two measures that this project considered to evaluate the scheme's performance were the speed of query execution and the latency introduced by *CrAC* – that is, measuring how long it takes to update the database after some change in the hierarchy has occurred (changes such as, adding a level to the hierarchy or removing a staff member from an investigation team). These measures were considered in a number of scenarios, and the results and discussion thereof follow in the next chapter.

In all the tests, the client was run on a Windows 7 32-bit machine with a clock speed of 2.27 Gigahertz, while the server was run on a *(find out what model the Honours lab machines are)*.

5.1. Experiment 1: Speed of query execution

5.1.1. Purpose

The purpose of this experiment was to determine how quickly a user's queries were executed in the system using *CrAC* (henceforth referred to as System C), and to measure the speed at which the same queries were executed in the system that only used SQL `GRANT` and `REVOKE` statements to deploy the access control model outlined in *Table 1* (henceforth referred to as System S).

5.1.2. Hypothesis

It was hypothesised that query execution would be slower in System C than in System S.

5.1.2.1. Rationale

There were two reasons for thinking this would be the case:

- System C and System S differ only in that the former uses encryption to provide record-level access to data, while the latter uses `views`. The former's approach resulted in many authorisation requests and encryption/decryption operations having to be done to retrieve what the latter could do by searching and returning records from a single `view`. For example, if a user were to request permission to view all the incidents on the system, System C would request the necessary decryption keys from the capturer of each incident, while System S would simply return the entries in that person's `viewableIncidents` view.

- For non-`SELECT` queries, System C and System S were identical except where System C introduced the additional overhead of encrypting certain attributes of the data before storing it.

5.1.3. Experimental design

Eight functions were tested on each system and Java's `System.currentTimeMillis()` was used to record the time at which the method called to execute the query and the time at which the query was completed or the results were done being printed to the console, and by subtracting the former from the latter, an estimate of how long the query took to be executed in that instance was obtained.

The eight functions tested were:

- adding a new staff member to the system;
- updating a staff member's personal details;
- viewing a full personal profile;
- viewing all staff members' limited profiles;
- capturing a new incident;
- viewing all the incidents;
- creating a new investigation;
- updating the details of an investigation.

To test adding records to the database, `StaffManager`'s `loadStaffFromFile`, `IncidentManager`'s `loadIncidentsFromFile` and `InvestigationManager`'s `loadInvestigationsFromFile` methods allow records to be entered from a bar-separated file instead of requiring human input. They were added in batches of a hundred, and the time taken to insert each tuple recorded.

To evaluate the query execution time for updates, updating personal details was chosen. The time taken to update an unencrypted field, an encrypted field, and then both in one transaction were recorded for later comparison.

Lastly, to test how quickly tuples are returned upon being requested for by a user, viewing the staff's limited profiles was timed and compared to viewing that required authorisations from the data owners in order to be able to read the data, for example, viewing incidents.

5.2. Experiment 2: Latency

5.2.1. Purpose

Latency is the amount of time it takes for a system to respond to a request for data with the relevant data. This experiment sought to observe the delay in updating the database tables as a consequence of various changes to the hierarchy of employees. It aimed to acquire this measurement for both Systems C and S for comparison.

5.2.2. Hypothesis

As with the previous experiment, it was hypothesised that System C would not fare as well as System S.

5.2.2.1. Rationale

The hypothesis was motivated by similar reasons to those listed for the previous experiment's hypothesis: where System S would just revoke and grant certain permissions, the extensive use of encryption in *CrAC* would require that much more work be done in System C to update records and ensure that they are no longer accessible to employees whose access rights do not permit them to view data that they had access to before anymore (decryptions and re-encryptions under new keys).

5.2.3. Experimental design

Five functions were run to measure the latency of Systems C and S in different situations, namely:

- demoting an employee;
- adding a level to the hierarchy;
- removing a level from the hierarchy;
- deleting an employee from the system;
- removing an employee from an investigation's team.

Java's `System.currentTimeMillis()` was used in the same way as in Experiment 1 to give a measure of latency: the time when the function began execution to when it ended were recorded.

5.3. Limitations

The experiments conducted were enough only to measure certain aspects of the access control's scheme's performance, so their results served mostly to identify whether or not *CrAC* is a promising solution that should be explored and developed further. To obtain a fuller picture of the scheme's performance, one would need to consider more factors, such as how quickly the system executes queries when a realistic number of people are logged in to it and are sending queries concurrently.

5.4. Summary

This chapter documented the experiments conducted to assess *CrAC*'s performance. The metrics used to measure this were query execution time and latency. *CrAC*, as well as its SQL `GRANT`-and-`REVOKE` parallel, were tested by running a total of eighteen functions on systems that implemented each, and these tests were repeated several times to see if consistent (and so probably reliable) results were obtained. The next chapter presents the results of the experiments, and further provides analysis of the results.

6 Results & Discussion

In this chapter, the results of the experiments outlined in the previous chapter are presented.

6.1. Experiment 1: Speed of query execution

As is to be seen in the graph below, inserts into the database tables generally took twice as long with *CrAC* as they did with the `views`-based access control scheme, proving the hypothesis stated in the previous chapter. This overhead is certainly at least partially introduced by the encryption that occurs on some fields of the data before it is stored on the server. Specifically in the case of adding new staff members to the system, the generation of each user's four-key set also contributes to this overhead.

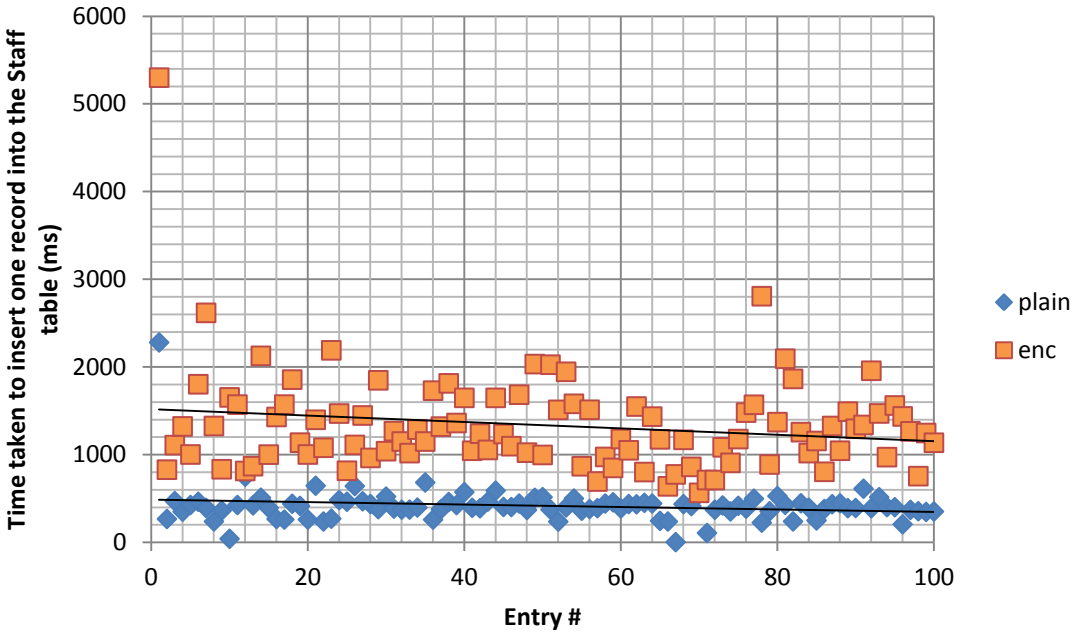

Figure 1. Graph comparing amount of time it took to enter a staff member's details into the database using CrAC and using the GRANT-and-REVOKE access control scheme.

In a similar vein, *CrAC* performed poorly returning records from database tables, taking up to five times as long as the benchmark scheme. This is depicted in the next graph. Again, this comes about as a result of the scheme attempting to decrypt each profile before returning it.

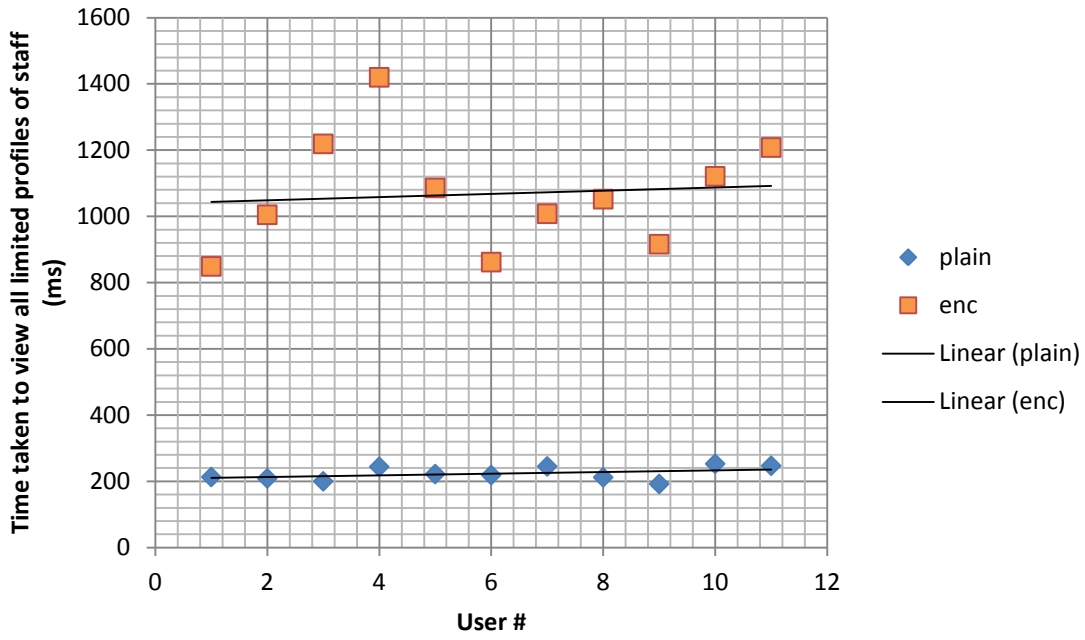

Figure 2. A graph showing how long the system took to return the result to a query requesting to see all the staff's profiles.

6.2. Experiment 2: Latency

It took over fourteen minutes for *CrAC* to adjust users' access restrictions after a level was added to the hierarchy. This was probably because the re-assignment of permissions involved decrypting and re-encrypting some data under a key different to that which now disallowed persons may have used before.

6.3. Summary

This incomplete chapter served to present and analyse the results of the experiments outlined in Chapter 5. Only a few of them were carried out, and the results of even fewer discussed, but they gave a clear enough indication that *CrAC* is not well-suited to the context of *Cry-Help* as it offers very slow execution of queries and affects the system's latency badly when the hierarchical structure is modified somehow.

7 Conclusion & Future Work

7.1. Summary of report

This report aimed to document the design, development and evaluation of a cryptographic access control scheme called *CrAC*. The scheme was designed in an attempt to provide a way of protecting an organisation's data within the organisation. Its primary objective was to facilitate access control, all the while maintaining the data's privacy even from an attacker who could access it directly on the server. The access control model that the solution had to be modelled for was a hierarchical one which further allowed sharing of data in a team of employees.

The solution began by employing a notion of "data ownership": every tuple entered in the database was assigned an owner, and this assigning process was intended to be straightforward. If a capturer recorded an incident, that record was assigned to their ownership; likewise for an investigation headed by a certain investigator, and lastly, for staff members' personal profiles. Working from this assignment of data owners, each person having been assigned a personal secret key and each classified record having had a random (at most) eight-digit tag assigned to it upon entry into the database, the sensitive fields of a given classified record were encrypted under a key derived from the data owner's secret key by enciphering the record's tag with it.

However, by taking this approach to design *CrAC*, many other features that would have to accompany it emerged. These included sharing of encryption keys and re-encrypting data that a former employee once had access to but should no longer be able to view, even if he were to attempt reading it on the server itself.

CrAC's performance was then evaluated against an equivalent access control scheme that didn't use any cryptographic elements. Two aspects of performance were considered to gain an idea of how worthwhile a research direction *CrAC* was, namely: speed of query execution and latency. In terms of these metrics, *CrAC*'s performance was very unsatisfactory, as it would take even up to five times as much time as the non-cryptographic solution to execute the same query on the same underlying data. Its impact on the database's latency was also far less than expected, with it taking over ten minutes to adjust users' access permissions after a level was added to the hierarchy.

7.2. Conclusion

The results of this project suggested that using independent keys as the backbone of a cryptographic access control scheme is not a good idea. At the outset, they were a lucrative prospect as their counterpart – dependent keying – tends to require a change of keys all the way down a hierarchy from the place of impact (for instance, if someone is removed from the hierarchy, the keys of the people beneath them in rank would need to be re-computed and so data encrypted under the old keys would need to be re-encrypted under the newly-assigned keys). However, it soon appeared that as much as re-derivation of personal keys in *CrAC* was not frequent, re-encryption of data was, and since that is an expensive operation, this is a problem, and it is very likely common across all endeavours to protect persistent data cryptographically.

This report concludes that as much as the solution scheme proposed meets the requirements stipulated in *Table 1*, its performance is too poor to be applied in a real police context.

7.3. Future work

One of the most inconvenient features of *CrAC* is how the requests for keys among peers are communicated. A user has to manually keep checking to see whether they have received such a request, and in the case that they have not, the program hangs as it waits on a request to arrive. A way to counteract this would be to develop a graphical user interface as opposed to the command-line interface that was used in this implementation of the access control scheme. By moving to a GUI, popup dialogs could be taken advantage of and used to communicate these requests without interrupting the user's work.

Also, it would be interesting and worthwhile to see how this scheme compares with a dependent key scheme that achieves most, if not all, of what *CrAC* does. If *CrAC* is found to perform worse than a dependent key scheme, then very likely independent keys shouldn't be considered as a means of facilitating row-level access control in databases.