AN INFRASTRUCTURE FOR DATABASE TAMPER DETECTION AND FORENSIC ANALYSIS

By

MELINDA JOY MALMGREN

_____

A Thesis Submitted to The Honors College

In Partial Fulfillment of the Bachelors degree
With Honors in

Computer Science

THE UNIVERSITY OF ARIZONA

M A Y 2 0 0 7

Approved by:

_____

Dr. Richard T. Snodgrass
Department of Computer Science

STATEMENT BY AUTHOR

This thesis has been submitted in partial fulfillment of requirements for a degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Signed: _____

Abstract

The need for secure data storage has become a necessity of our time. Medical records, financial records, and legal information are all in need of secure storage. There have been several methods proposed for how to secure and tamperproof a database. The use of append-only transaction time databases provided the ability to know the exact time in which data was changed.

This formed the foundation for a new approach wherein the DBMS computes a cryptographically strong one-way hash function for each tuple inserted and then notarizes it using a notarization service. This made it possible to check the consistency of the data by comparing it to the values stored with the notarization service.

In continuation with this method, algorithms were designed to further analyze an intrusion of a database. This thesis examines the implementation of these methods and algorithms, including how to store all relevant information and hash codes pertaining to each database being monitored and how to analyze and display the results of this stored information.

By utilizing a central security master database as part of an enterprise architecture for auditing, and by providing role-specific GUIs, it is possible to efficiently manage the auditing of databases across an enterprise.

TABLE OF CONTENTS

1. Need for Tamper Detection

Secure data storage is an everyday requirement for public businesses, government agencies and many institutions. For many organizations, if data were to be maliciously changed, whether by an outsider or by an inside intruder, it could cause severe consequences for the company. Possibly even for their clients as well.

There are many reasons why someone might want to tamper with data. For example, an unsatisfied student who receives a “D” in his calculus class, in which he needed at least a “B”, could be highly tempted to try to dishonestly change his grade to a “B” in the school’s database. This would be an example of someone who would have to hack into the system from the outside, unless of course the student somehow had access to the database containing the grade.

A similar example, wherein the intruder is an insider rather than someone hacking in from the outside, could be that of an employee at a large company who is trying to meet his sales requirements for a fiscal year. He might attempt to change the dates of transactions to make it appear that they had transpired within the previous fiscal year when, in reality, they had not.

Another example wherein an insider might want to corrupt the database could be at a doctor’s office. As part of the federal Health Insurance Portability and Accountability Act (HIPPA) [1], medical providers are responsible for auditing their interactions with patients, vendors and other medical providers to ensure patient privacy. Because of this, doctors are not allowed to release a patient’s medical information to an insurance company, or any other company, without written permission from the patient. Assuming that the doctor uses a database to track not only all the medical and prescription records, but also the records of who has signed an information release form, it could be highly possible that someone within the office may want to alter the data in the database. If information about a patient was released to an insurance company before the release form was entered into the database, the doctor could potentially be sued for releasing information prior to consent. In this situation it would not be surprising if the doctor (or one of his assistants) tried to either change the timestamp of when the release form was entered or to remove the record showing that they released information about the patient.

These three examples provide just a few of the many reasons why someone might want to tamper with a database. These fraudulent acts can be punishable by law and result in severe consequences if the perpetrator is caught.

It is obvious to see that secure data storage is a huge necessity in everyday life. Would you want your social security information being changed, or your checking account's balance modified, or your medical information released without your consent? The purpose of this project is to explore a new method of protecting databases from these situations and provide a way to monitor a database for such intrusions.

By utilizing a central security master database as part of an enterprise architecture for auditing, as well as role-specific GUIs, it is possible to efficiently manage the auditing of databases across an enterprise.

2. Background and Related Work

This project is a direct continuation of a concept that was created many years ago by Christian Collberg, Richard T. Snodgrass, and Shilong Stanley Yao. There have been multiple papers published describing the ideas and goals this project is based on. The first of which is *Tamper Detection in Audit Logs* [2], discussed in Section 2.1. In Section 2.2, we discuss another paper, *Forensic Analysis of Database Tampering* [3], which expands on the concepts introduced in the first paper. In these two sections we will summarize these ideas, first looking at how to detect tampering within a database and then secondly, how to analyze such tampering. In Section 3 we will discuss how this specific project has assisted these ideas.

2.1 Tamper Detection in Audit Logs

Mechanisms were proposed within a database management system (DBMS), based on cryptographically strong one-way hash functions, which prevent an intruder, including an auditor or an employee or even an unknown bug within the DBMS itself, from silently corrupting the audit log [2]. It was proposed that the DBMS transparently store the audit log as a transaction-time database, so that it is available to the application if needed. The DBMS should also store a small amount of addition information in the database to enable a separate *audit log validator* (to be referred to simply as the *validator* from here on*)* to examine the database along with this extra information and state conclusively whether the audit log has been compromised. It was also proposed that the DBMS periodically send a short document (a hash value) to an off-site *digital notarization service*, to bind when changes were made to a database.

One important thing to note about this approach is that a transaction-time database is an *append-only database*. Modifications never remove information from the database; instead they only add to it. All past versions of the data are retained and can be reconstructed from the information stored in the database.

On each modification of a tuple, the DBMS obtains a timestamp, computes a cryptographically strong one-way hash function of the (new) data in the tuple and timestamp, and sends that hash value to the notarization service, obtaining a notary ID.

However, the authors also noted that notarizing each tuple as it was modified would be quite an expensive operation and hence proposed to instead hash all the tuples modified by a single transaction to compute a single hash value. Instead of storing each notary ID with the given tuple, the IDs are instead stored in a separate *Notarization History Table*. In fact, it is most likely that notarization would only need to occur once per day at which time all changes in the previous twenty-four hours could be hashed and notarized. A tool, called the *notarizer*, can perform these operations on a regular schedule.

Finally, the validator periodically scans the audited tables, computing the hash values on a per-transaction basis and then finally sending these hash values to the digital notarization service along with the ID stored in the Notarization History Table. The validator will then be able to report if the current data is inconsistent with the audit log.

Let's assume an intruder gains access to the database. If they change either the data or a timestamp, the hash value that will be computed will be inconsistent with the notarized ID. Even if the intruder gained access to the hash function itself they could not store a newly computed hash value because it would be inconsistent with the one that was notarized.

One might wonder how all of the above changes would affect the performance of a database. The authors discovered that the auditing overhead was between 9% and 16% in all the experiments they ran. This is a small price to pay for the protection of highly critical data. In summary, the authors present a feasible way of detecting tampering within a database.

2.2 Analysis of Database Tampering

The idea described above provides a way to detect tampering within a database. The question then arises of what to do once tampering has been detected. This issue was discussed in a paper by Pavlou and Snodgrass [3]. They state that "Forensic analysis is needed to ascertain *when* the intrusion occurred, *what* data was altered, and ultimately, *who* is the intruder" [3]. They built their implementations on the ideas discussed in the previous section.

The authors begin by defining some terms, a few of which will be useful to reiterate here. A *corruption event* (*CE)* is any event that corrupts the data and compromises the database. This can happen in many forms, be it an actual intrusion or merely a hardware failure. A *notarization* event (*NE*) is the notarization of a hash value by the digital notarization service and occurs every time the notarizer is run. A *validation event* (*VE*) occurs every time the validator is run. Normally these validation events are scheduled and happen at constant time intervals, but they can be random as well. Tampering is first detected when a validation event has failed.

Forensic analysis involves both *temporal detection*, the determination of time, and *spatial detection*, the determination of where in the database the data was altered. The authors created a c*orruption diagram* to graphically represent corruption event(s) in terms of the temporal-spatial dimensions of a database. Figure 1 illustrates a simple corruption event by using a corruption diagram. A corruption diagram is a graphical representation of corruption event(s) in terms of the temporal-spatial dimensions of a database.

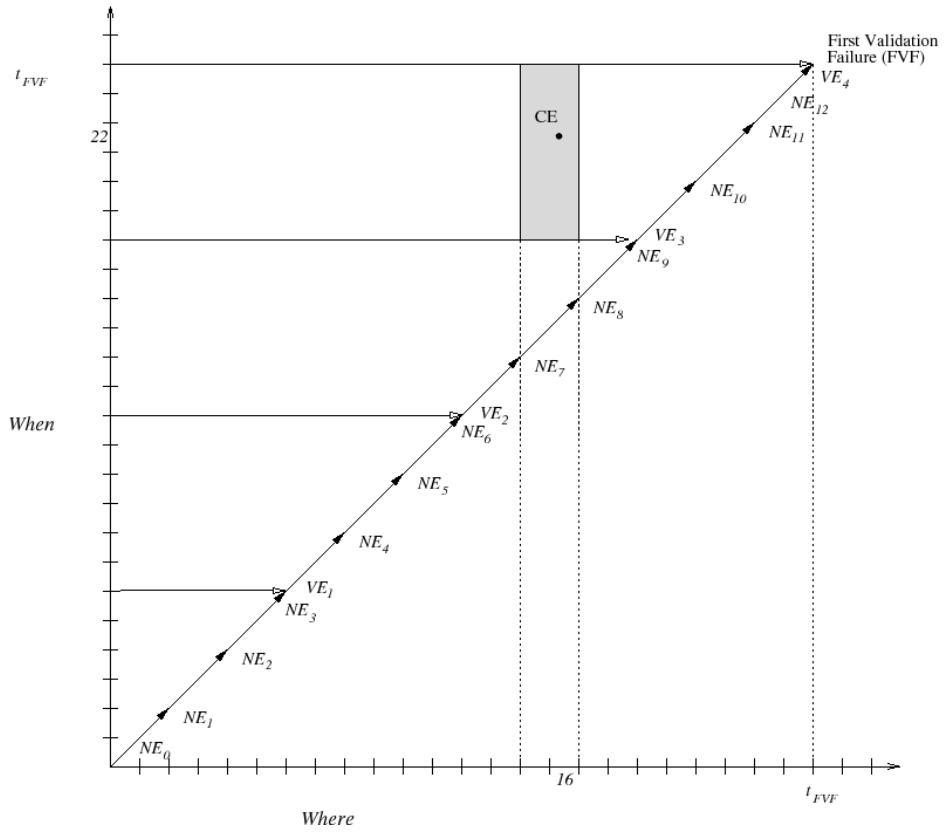

Figure 1 - Corruption Diagram

In a corruption diagram the x-axis represents the transaction time of the data. In other words, this axis represents when the data was stored in the database. This axis is labeled "*Where*" because a tuple is designated by the time of the transaction that inserted that tuple. The y-axis represents the actual-time of the database and is called the "*When*". The 45-degree line is the *action axis* since all the action occurs on this line. Any point on this line thus indicates a transaction committing at a particular transaction time (x-axis) that happened at a clock time (y-axis).

Notarization events are denoted along the *action axis* by the points labeled with "*NE*". Validation events are similarly denoted along the axis by the points labeled with "*VE*". Since the validator can only validate data that has already been notarized it is pointless to have validation events that are unaligned with notarization events. Thus the validation interval should not be less than the notarization interval and the validation interval should be a multiple of the notarization interval. For this reason, all validation events on the diagram are aligned with notarization events. From looking at the diagram we can tell that all the validation events except for the last one succeeded. Due to the fact that the last one failed, we have a corruption diagram. In Figure 1 we can also see the corruption event contained within the corruption region which is denoted by "*CE*". Since this is just an example it is possible to display the corruption event on the graph, however in most cases the exact point of corruption is not known.

Upon the detection of a corruption, the next step in forensic analysis. The authors proposed multiple algorithms that use only the database itself to determine the bounds on a corruption region. For simplicity, I will only describe one of them here, however multiple algorithms do exist and there is the possibility of more in the future. The algorithm that we will look at is the *monochromatic forensic analysis algorithm.* First let's look at how to bound the "when". We know that at the second-to-last validation event that everything was fine. Thus the corruption happened sometime between the last successful validation event and the failed validation event which bounds the "when" on the diagram. Secondly, the "where" can be determined by individually rehashing all the tuples contained within each notarization event and checking them against the notarized hash values. If any of these values do not match up, we know within which notarization

event the corruption occurred and we then have a bound on the “where”. Upon performing this analysis we get the diagram shown in Figure 1.

We should note that this example deals with *data-only* corruption events and not with a corruption that occurs because of timestamps of tuples being changed. The monochromatic algorithm can also deal with such corruptions and there are many other algorithms that are able to create smaller and smaller corruption regions. These are explained elsewhere [3]. It should be noted that while each algorithm is more complex than the last and requires more main-memory processing, each successive algorithm adds additional precision that more than counterbalances the extra work.

In summary, the authors first devised a clean way to visually display a corruption to an analyst using a corruption diagram. They then proceeded to present multiple ways of forensically analyzing such corruptions to more accurately determine where and when they happened. It is expected that more algorithms will be created in the future that will allow even more precise bounds than before.

3. Overview

This project implements and builds on the concepts introduced in Section 2. We designed an auditing system to ensure that multiple databases can be protected from intrusion and corruption.

3.1 Purpose

The purpose of this project was to aid in the implementation of the concepts described in Section 2. The notarizer, validator and forensic analysis application were all implemented by other students. The aim of this project was to design a master database to store all information involved in auditing and forensically analyzing a database and to create three user-specific applications for allowing users to interact with the system.

3.2 Implementation

The notarizer, validator, and forensic analysis application must all be stored and executed on a highly secure machine in order to ensure that they themselves are not tampered with. If any of these applications were not well protected and hence were tampered with, a complete failure of the auditing process could result. In the same manner, the master database must also be kept on a highly secure machine as it stores all of the information produced by the notarizer and validator. Whatever machine(s) these three items are run on should also be stored in a secure room with limited access. This is depicted by the large box in Figure 2.

The Individual Databases shown in Figure 2 represent the databases that are being audited. The Forensic Analysis Validator is the application described above in Section 2. The Digital Notarization Service represents any third-party company that can notarize a digital document. The three applications along the side; the Security Master Application, the Database Monitor Application and the Forensic Analysis Application represent the three applications that were created for this project, one for each role described in Section 3.4. Each application is described in detail in Section 5.

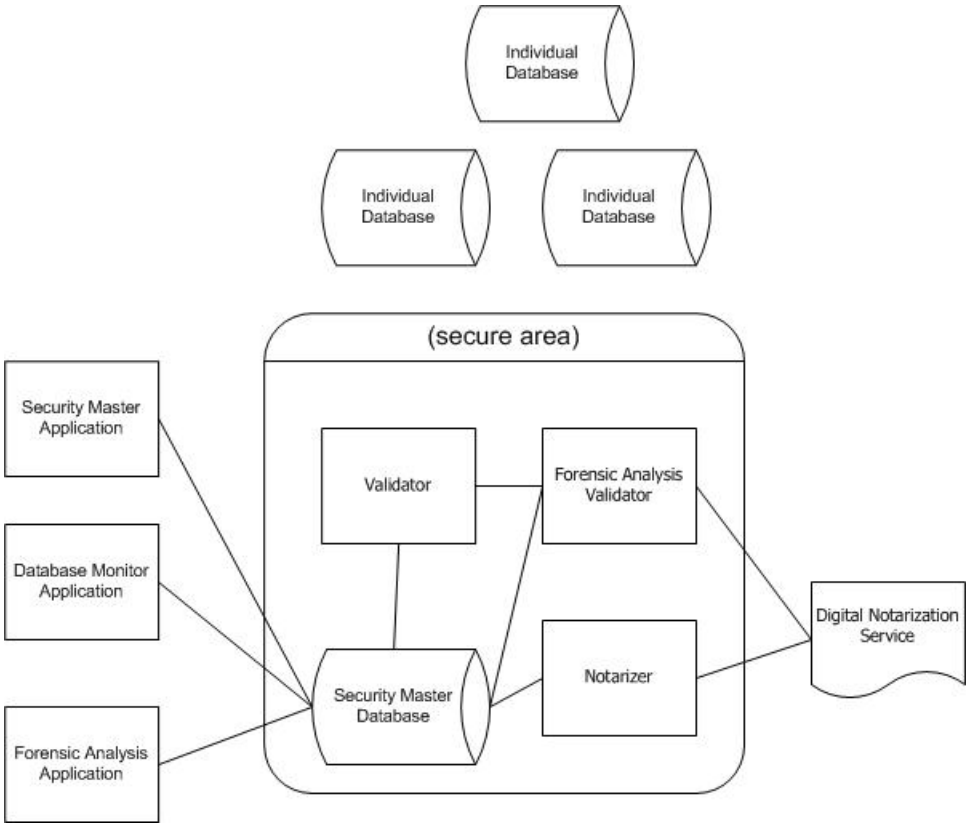

Figure 2 - Auditing System

The Security Master Database is the central element of this system as it interacts with all components except for the individual databases themselves and the digital notarization service. It is the one tool that ties together all the other tools in the system.

3.3 User Roles

There are three roles within the system. The first is that of the *Chief Security Officer* (*CSO*). The CSO is a role that will most likely only be taken on by one employee. This employee should be a highly reliable person who can be readily available to deal with problems that arise in the system. The CSO will be in charge of maintaining the system as a whole. They alone will have the ability to add other employees to the system and to assign each employee to their specific tasks. The CSO role is also the only role that has the ability to add an individual database that the system should monitor. Most

importantly, the CSO is in charge of setting the overall system settings such as maximum notarization and validation intervals that all databases must abide by.

The next role is that of the *Database Administrator* (*DBA*). Generally speaking, a DBA is responsible for the general management and design of one or more individual databases. In our situation a DBA is also in charge of selecting the configuration settings for auditing their database(s). These settings include such things as which algorithm to use, how often to notarize, and how often to validate. There can be many employees in the system that have this DBA role.

The last role is that of the *Crime Scene Investigator* (*CSI*). The CSI is responsible for investigating tamper detections once they have occurred. Each CSI is also assigned to specific databases. Like the DBA role, there will most likely be many employees in the system that are given this role.

3.4 Sequence of Events

A normal sequence of events would be as follows: the notarizer periodically hashes the data in an individual database and notarizes this value with the digital notarization service. At the same time, the validator will periodically be checking the data in this database against the values stored with the digital notarization service. At some point, the validator detects tampering and calls the Forensic Analysis Validator. The Forensic Analysis Validator then performs forensic analysis on this database and stores the corruption region in the Security Master Database. Upon detecting the tampering the validator will also email the DBA and CSI of this database to inform them that tampering has been detected. Using their appropriate applications, the DBA and CSI will then be able to view all known information about the tampering, including a corruption diagram.

4. Security Master Database

The *Security Master database (SMDB)* is the backbone of our infrastructure for tamper detection and analysis. Since it is only a database, it is obviously not notarizing and validating databases or attempting to portray a tamper detection to the user, but without the SMDB none of these tasks would be possible. Without the SMDB a notarization event would just be sent into oblivion, rendering the event entirely useless and unusable in the future. Without the SMDB there would not be any status or records of past events to display to an analyst.

The purpose of the SMDB is to be a single point of interaction for all of the above tasks. It will be the only place that the notarizer and validator write to and the only place that the forensic analysis application draws from. It will be the one secure place where the CSO and all DBAs can monitor their databases.

Since this part of our infrastructure is so critical, our tables must be designed intuitively and logically. Seeing as this is a project that is the first of its kind and still in implementation we need a design that is easily expandable as new algorithms or concepts are introduced. Also, it was also very important that we not store redundant data across multiple tables.

4.1 Conceptual Schema

We started our design of the SMDB by creating an Entity-Relationship diagram [4] as shown in Figure 3. The top leftmost entity type in Figure 3 is the CONFIGURATION entity type. This entity type is specialized to record either an individual database or the overall settings of the system. The condition for this specialization is the composite Source attribute, which consists of the Name and Path. Both of these attributes are set to the string "`Overall`" when an OVERALL entity type is being recorded; otherwise an INDIVIDUAL DATABASE entity type is being recorded.

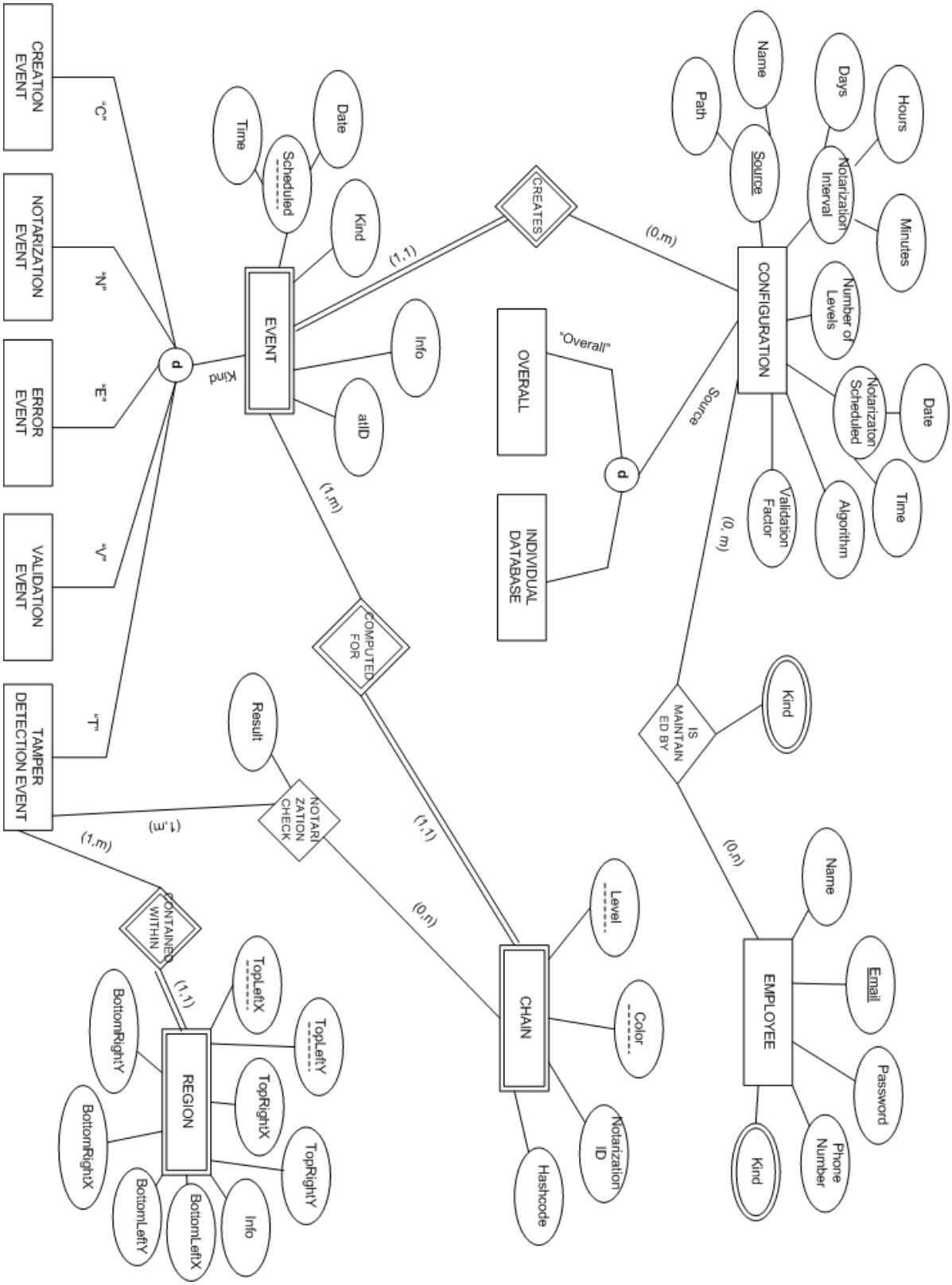

Figure 3 - Entity Relationship Diagram of SMDB

The composite Source attribute is the primary key of the CONFIGURATION entity type. The Notarization Interval, Validation Factor, Number of Levels, Algorithm, and Notarization Scheduled attributes all record specific settings for notarizing and validating the given database. In the case of the OVERALL entity type these attributes record the maximum notarization interval and validation interval and the minimum algorithm and number of levels allowed for each database throughout the system.

The INDIVIDUAL DATABASE entity type has a many-to-many relationship type with the EMPLOYEE entity type because each employee may be associated with more than one database and each database has multiple roles that different employees could be assigned to. At any given time, there can be only one DBA and only one CSI assigned to each database. It is possible for a database to have only one or the other or neither assigned to it. Each relationship between an INDIVIDUAL DATABASE and an EMPLOYEE is denoted by a Kind, either 'C' for CSI or 'D' for DBA. This attribute is multi-valued because an employee may be both the DBA and the CSI for a given database.

The EMPLOYEE entity type also contains a multi-valued Kind attribute to represent the role(s) an employee may take on. In addition to the two listed above, an employee may also have the role of CSO, which is denoted by an 'A'. Instead of linking the employee to an individual database via the IsMaintainedBy relationship, as is done for a DBA or CSI role, the employee is linked to OVERALL to take on the role of the CSO. The Name, Email and Phone Number attributes of the EMPLOYEE entity type all represent basic information needed to contact an employee. The Password attribute provides a way to ensure that an employee is who they say they are. The Email attribute is the primary key of this entity type because no two employees should be allowed to use the same email address.

The next entity type, Event, has a one-to-many relationship type with the INDIVIDUAL DATABASE entity type. Each database can create multiple events but each event is linked to only one database. The ActualDate attribute represents the *valid-time* in which the event *occurred*. The Info attribute allows for anything unusual about an event to be noted.

Similar to the CONFIGURATION entity type, the EVENT entity type is also specialized into other entity types. The Kind attribute is the condition for this specialization and can take on one of five values as seen in the diagram. The first kind of event is the CREATION EVENT, which represents when a database was first created. This information provides a starting point for both the notarizer and the validator when performing operations on a given database.

Both the NOTARIZATION EVENT entity type and the VALIDATION EVENT entity type are fairly self-explanatory in their use. The NOTARIZATION EVENT entity type also contains an atID attribute for scheduling the next run of the notarizer. Probably the greatest confusion comes from differentiating between the ERROR EVENT and the TAMPER DETECTION EVENT. A TAMPER DETECTION EVENT is the direct result of a *failed* validation event. Forensic analysis is then started by performing notarization checks. These checks are represented by the one-to-many relationship type between the TAMPER DETECTION EVENT entity type and the CHAIN entity type, which is discussed below. When the result of one of these checks is a failure, a TAMPER DETECTION EVENT is created. On the other hand, an ERROR EVENT is something that is not directly tied into another event. An example would be if the notarizer or validator attempted to access a specific database but could not locate it or was denied access. In these situations an ERROR EVENT would be created with a detailed description of the problem in the Info attribute.

The REGION entity type has a one-to-many relationship type with the TAMPER DETECTION EVENT entity type because a tamper detection may be associated with multiple corruption regions, but each of those regions is associated with just the one detected tampering. There are eight attributes that represent four X, Y coordinate pairs on a graph. When a region is a rectangle all eight attributes are needed. In the cases where a region is a triangle only six of the eight attributes need to be used, in which case BottomRightX = BottomLeftX and BottomRightY = BottomLeftY. The REGION entity type is used mainly for visually displaying a tamper detection to the user via a graph.

Finally, we have the CHAIN entity type. It is linked to the EVENT entity type because validation events can be broken down into multiple chains and notarization events can also create a single black chain. The other event types do not create chains.

The CHAIN entity type is linked to the TAMPER DETECTION EVENT entity type because chains are only validated once tampering occurs. This entity type also has a one-to-many relationship type with the EVENT entity type because each chain is associated with a particular validation event and each validation event may have multiple chains. In some algorithms chains can be different colors, and there can also be multiple chains of a single color within an event. The Color and Level attributes are used to store this information. The Hashcode attribute stores the hashed value of all the tuples within the chain. The NotarizationID attribute stores the value returned to the notarizer by the digital notarization service.

One of our main goals when designing the CHAIN entity type was making it easily expandable for uses with future algorithms. By setting up the EVENT and the CHAIN entity types the way that we did, it will be easy to accommodate new algorithms as they are designed.

4.2 Conceptual Schema with Time

One important consideration is which entity types are valid-time representations and which entity types are simply transaction-time representations. The difference is that *valid-time entity types* represent when a fact in the database is valid in reality while *transaction-time* tells us only when certain facts were stored in the database. In this section we annotate various entity and relationship types with their temporal behavior.

The EVENT entity type is a valid-time event entity type because it represents when exactly an event occurred in reality. It is the only entity type in Figure 3 that is an *event* entity type. The reason it is an *event* entity type is because it stores information that happened at a specific time and not information that is valid over a period of time. The CHAIN entity type is also a valid-time entity type as each chain is limited by a specific real time interval; however, it is a *state* entity type because it represents data that is valid over a period of time.

The CONFIGURATION entity type is a transaction-time state entity type because it only records when the information was stored in the database and not when the information became valid. This entity type is a state entity type since the information it

records is a state of configuration that is valid over a certain time period. The EMPLOYEE entity type and the IsMaintainedBy relationship types are both valid-time representations because they represent when an employee was active in the system as well as when they were active in certain roles. Both of these entity types are state entity types as they record information that is valid over a period of time.

4.3 Mapping to the Logical Schema

After creating the conceptual schema as seen in Figure 3, we then mapped it into the logical schema shown in Figure 4 and Figure 5. There were many things to consider including how to deal with time and how to best break the entity-relationship pairs into tables. Those issues involving time are discussed in the next section. First, we will look at how we mapped the entities to tables and attributes to columns. In some cases the primary keys of the tables may not be obvious until we discuss how time affects the table.

Starting with the CONFIGURATION, OVERALL, INDIVIDUAL DATABASE trio, we mapped them into a single table, the `Configuration` table, which represents both the OVERALL and INDIVIDUAL DATABASE entity types. We did this because neither the OVERALL nor the INDIVIDUAL DATABASE entity types had any of their own attributes and they inherited all of the attributes from the CONFIGURATION entity type. The Name and Path attributes continued to make up the primary key. These two columns are set to "`Overall`" when a tuple is representing the OVERALL entity type; otherwise the tuple represents an INDIVIDUAL DATABASE entity type.

In a similar manner, we mapped the CREATION EVENT, NOTARIZATION EVENT, ERROR EVENT, VALIDATION EVENT and TAMPER DETECTION EVENT entity types into a single `Event` table. The only column that is not used by all the events it the atID column and since it was only one column we did not feel that it would be a waste of space in the table. When considering the primary key for the `Event` table, a surrogate key, EventID, was used to simplify the table. This also makes it much simpler for other tables to link with the `Event` table since they only need to link with the one field instead of multiple fields. The CHAIN entity type was mapped to its own `Chain`

table. To simplify the primary key once again, we added a surrogate key, ChainID, to serve as the primary key of the table.

Configuration **Table**

Name	char(35)
Path	char(35)
NotIntDays	tinyint(4)
NotIntHrs	tinyint(4)
NotIntMins	tinyint(4)
ValidationFactor	tinyint(4)
Algorithm	varchar(35)
NumLevels	tinyint(4)
NotarizationScheduledDate	timestamp

Employee **Table**

Email	char(35)
Name	char(35)
Password	varchar(35)
PhoneNumber	varchar(35)
isCSI	enum(‘Y’, ‘N’)
isCSO	enum(‘Y’, ‘N’)
isDBA	enum(‘Y’, ‘N’)

IsMaintainedBy **Table**

Name	varchar(35)
Path	varchar(55)
Email	varchar(35)
Kind	enum('A','C','D')

Event **Table**

EventID	int(11) auto_increment
Name	varchar(35)
Path	varchar(55)
Kind	enum('C','E','F','N','V')
Info	varchar(500)
atID	int(11)
ScheduledDate	timestamp

Figure 4 - Logical Schema Part 1

The NotarizationCheck relationship type is one of two relationship types that were mapped to actual tables. The reason that we did this is because the NotarizationCheck relationship type is a many-to-many relationship type between the TAMPER DETECTION EVENT and CHAIN entity types. We keep track of which tamper detection event and which chain is involved with the relationship with the FailureID and ChainID columns. The FailureID column is actually a foreign key to the `Event` table's EventID. The ChainID is also a foreign key to the ChainID of the Chain table. The only other column in the table is the Result column that stores 'P' for a passed check and 'F' for a failed check. Since this is a many-to-many relationship type, the ChainID and FailureID columns make up the primary key of this table.

The REGION entity type required some special consideration when we mapped it to the logical schema. This is because in certain algorithms a region is bounded by notarization and validation events and in other algorithms a region is not bounded by specific events. In order to make the REGION entity type work the way we wanted it to we decided to map it to two tables instead of one. In both tables the primary key consists of the EventID column along with the TopLeftX and TopLeftY columns since the top left coordinates of a region should always be unique within a specific event.

The first table is the `SynchronizedRegion` table which holds regions for those algorithms that are bounded by specific events. Since each coordinate is an actual event, each of the eight coordinate points is a foreign key to the `Event` table's EventID. We also have an EventID column within the `SynchronizedRegion` table that links the region as a whole to a specific tamper detection event. This column is once again a foreign key to the `Event` table's EventID.

The second table is the `UnsynchronizedRegion` table which also contains the foreign EventID column but instead of all the coordinate points being events they are simply timestamps. We added columns for the millisecond values of each coordinate as well since the validator stores times down to the exact millisecond. The two top-left millisecond columns are also part of the primary key.

The EMPLOYEE entity type was mapped directly into a table with columns for each of its attributes. In the case of the multi-valued Kind attribute three distinct columns were made instead. The isCSI, isCSO and isDBA columns can each be either 'Y' for Yes

Chain **Table**

ChainID	int(11) auto_increment
EventID	int(11)
Level	tinyint(4)
Color	varchar(15)
StartMilliseconds	int(11)
StopMilliseconds	int(11)
Hashcode	char(40)
NotarizationID	char(40)

NotarizationCheck **Table**

ChainID	int(11)
FailureID	int(11)
Result	enum('P','F')

SynchronizedRegion **Table**

EventID	int(11)
TopLeftX	int(11)
TopLeftY	int(11)
TopRightX	int(11)
TopRightY	int(11)
BottomLeftX	int(11)
BottomLeftY	int(11)
BottomRightX	int(11)
BottomRightY	int(11)

UnsynchronizedRegion **Table**

EventID	int(11)
TopLeftX	timestamp
TopLeftY	timestamp
TopRightX	timestamp
TopRightY	timestamp
BottomLeftX	timestamp
BottomLeftY	timestamp
BottomRightX	timestamp
BottomRightY	Timestamp
TopLeftXMillisecond	int(11)
TopLeftYMillisecond	int(11)
TopRightXMillisecond	int(11)
TopRightYMillisecond	int(11)
BottomLeftXMillisecond	int(11)
BottomLeftYMillisecond	int(11)
BottomRightXMillisecond	int(11)
BottomRightYMillisecond	int(11)

Figure 5 - Logical Schema Part 2

or ‘N’ for No. Since an employee can be any combination of these three roles it was necessary to have a column representing each role. The Email attribute still worked as the primary key for this table.

The IsMaintainedBy relationship type is the second of the two relationship types in our diagram that got mapped to its own table. Once again this is because it is a many-to-many relationship between its two entities. The IsMaintainedBy table contains columns for the foreign keys of both entities that it relates; Name and Path from the CONFIGURATION table and Email from the EMPLOYEE table. These three columns make up the primary key of this table. In addition it contains a Kind column which denotes if the assignment is for a CSI or for a DBA.

4.4 Mapping to the Logical Schema with Time

One thing that was not discussed in the previous section was how each of the tables deals with time. We will now proceed to explain which columns were added to each table to deal with time. The complete logical schema can be found in Appendix A. When mapping to the logical schema we also added a column to the Configuration table that became a part of the primary key, the StoredDate column. This column is an auto-generated timestamp of when the tuple was inserted into the table. This addition to the primary key was necessary so that multiple configurations could be stored for a single database allowing the history of past settings to be recorded. This makes the table *stepwise constant without gaps*. The reason there are not any gaps is because we are only storing one date for each tuple and that is the creation date of the tuple. Since each database has a tuple inserted into this table when it is first entered into the system and because there is no stop date for a tuple there can never be a period of time wherein a database does not have a configuration. This is what makes the CONFIGURATION table a transaction time table. The only other column in this table that deals with time is the NotarizationScheduled column, which is a user-defined time.

The Event table is the most complex table in the database as far as time is concerned. Since this table is full of events we had to have a column that tells us when

the event occurred. However, we encounter problems when the notarizer does not run on perfectly scheduled intervals. Let's say for example, the notarizer is supposed to run on a certain database every night at 12:00 AM, but for some reason it does not run until 12:03 AM one night. When hashing the tuples in the database we must know exactly when the notarizer stopped hashing the last time it ran. Since tuples can be timestamped down to a specific millisecond, we have to store the exact millisecond the notarizer stops at. So in our `Event` table we added a column ScheduledDate which represents when the event was supposed to happen, and we have the ActualDate and ActualMillisecond columns to store when the event actually occurs. A notarization event is the only event type that uses the scheduled date and time; the rest of the events only require the actual date and time. The reason that the scheduled date is not just stored instead of the actual date for a notarization event is because it is important to know when the notarizer was supposed to run versus when it actually ran. Also the scheduled date and time are useful for graphing purposes, so that events are displayed evenly across the graph even if the events are off by a few minutes. Since we are storing when the event occurred in real time this table is indeed a valid time table. The primary key of the table was already the surrogate key, EventID, so we did not have to change the primary key of this table.

When adding time to the CHAIN table we simply added the StartDate and StopDate of each chain. And since each chain involves hashing we store the time down to the exact millisecond; thus, we also have the columns StartMilliseconds and StopMilliseconds. Since these times are real world times this is a valid time table. As with the `Event` table, there was already a surrogate key, ChainID, serving as the primary key of this table, and that did not have to be changed.

The `Employee` table required the addition of StartDate and StopDate columns in order to record when an employee was active in the system. Without these two columns there would be no record of when an employee was first added to the system or if an employee had any gaps in their career. Since an employee could potentially be listed in this table twice if there are gaps in their career we had to add StartDate to the primary key of this table along with Email. Because we are recording real times, this table is also a valid time table.

The last table that required modifications to accommodate time issues was the `IsMaintainedBy` table. Since the roles of each employee in relation to different databases could change, we also added a StartDate and StopDate column to this table. The primary key of this table was also expanded to include the StartDate along with Name and Path from the CONFIGURATION table and Email from the EMPLOYEE table. Once again this table is recording when an employee had a certain role in real time so this table is also a valid time table.

The `NotarizationCheck`, `SynchronizedRegion` and `UnsynchronizedRegion` tables did not need any modifications when considering time.

4.5 Conclusion

As you can see, the SMDB holds all the information necessary for this infrastructure and does so in a well organized manner. It was also designed to be adaptable and fully expandable. Future algorithms and advancements to this infrastructure can be supported when using this design as a starting point.

5. The Applications

Three applications were created, one for each role in the system. They provide a way for employees to interact with and view the state of system. While some of the applications contain similar features, all were designed to serve the unique role of either a CSO, a DBA, or a CSI. The features of all three applications are described in this section while design and implementation issues are discussed in Section 6.

5.1 Security Master Application

The *SMA (Security Master Application)* is the main instrument for interacting with the system as a whole. In most situations there will be only one employee, the CSO, who has the ability to login to this application. It provides the highest level of manipulation within the system. The purpose of the SMA is to provide control of the system to the CSO as well as to alert the CSO of any problems in the system.

5.1.1 Control Features

The initial screen of the SMA is shown in Figure 6. The application provides the CSO with many features for manipulating the system. The first of which would be the feature that allows the CSO to assign a DBA and a CSI to each database or to un-assign any DBA or CSI from a database. This is done by using the *Assign DBA* or *Assign CSI* buttons as seen in Figure 6. Another feature is the ability to set the minimum notarization interval and the maximum validation interval for all databases in the system. These options can be seen at the bottom of Figure 6. When these setting are modified all databases in the system are required to meet these maximums and minimums. DBAs will not be allowed to change individual database settings to anything outside of these bounds.

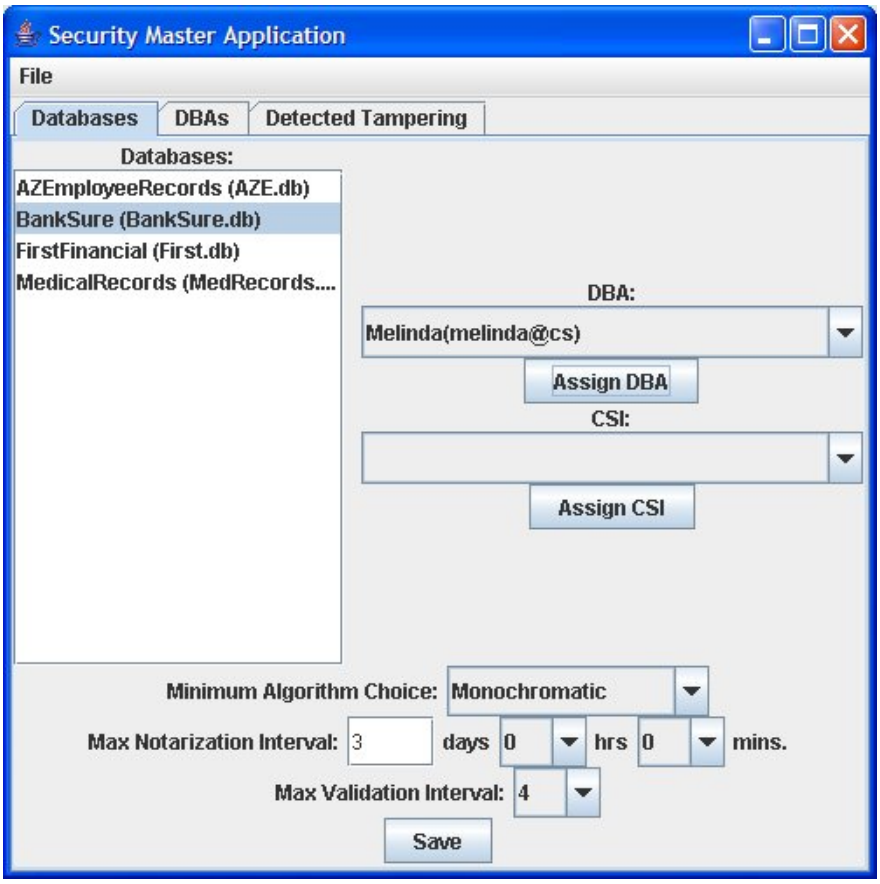

Figure 6 - Security Master Application (Databases tab)

The SMA also allows the CSO to add databases and employees to the system. These options can be found under the *File* menu as *Add Database* or *Add Employee*. The dialog for adding an employee can be seen in Figure 7.

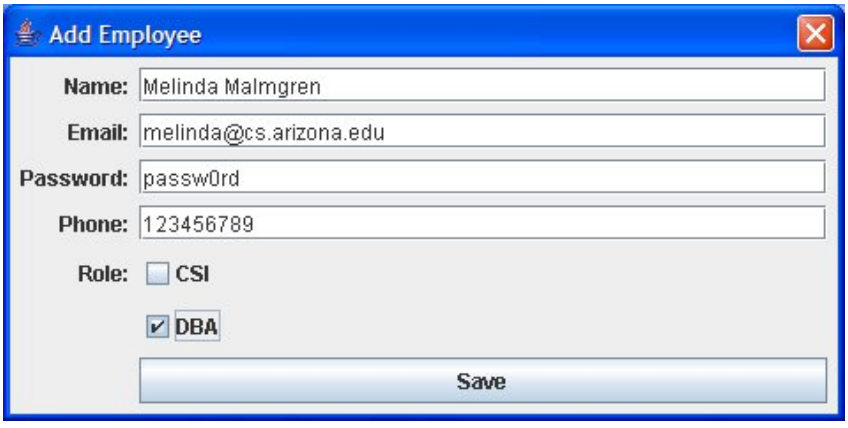

Figure 7 - Add Employee Dialog

5.1.2 Viewing Features

In addition to providing features for controlling the system, the SMA also includes many features for viewing the state of the system. For example, as shown in Figure 8, by clicking on the *Detected Tampering* tab the CSO is able to view of a summary of all tamper detections in the system. This list gives a high level view of each problem including which database it relates to and provides the date and time of the occurrence as well as any special information that is known. In future expansions of the application the list could be expanded to allow the user to click on a description and be able to view a more detailed description and in some cases, a diagram depicting the issue.

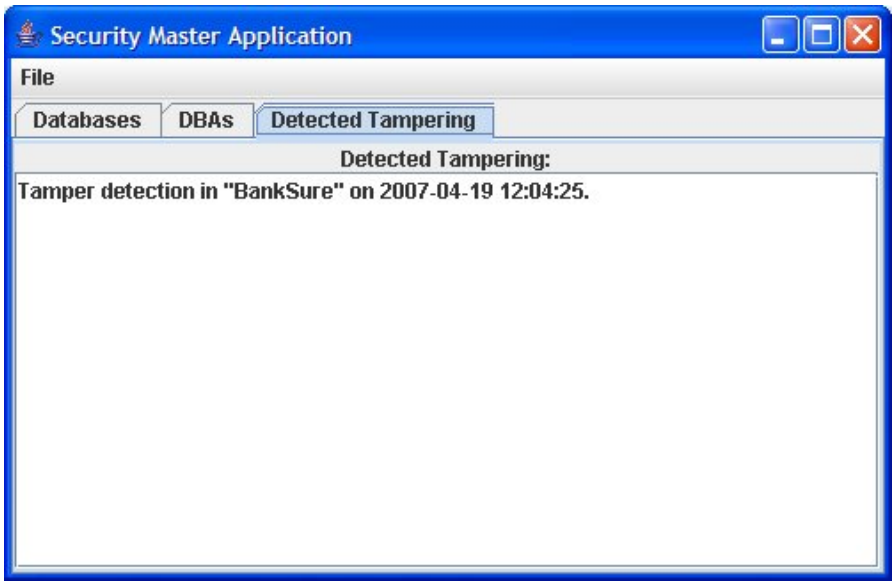

Figure 8 - Security Master Application (Detected Tampering tab)

Another viewing feature is the DBA tab which provides a list of all DBAs currently in the system. By clicking on any of the names the CSO is able to view a list of all the databases that DBA is currently assigned to. This can be seen in Figure 9. Another viewing feature is also a control feature; the ability to select a database in the *Databases* tab and see which DBA and which CSI (if any) are assigned to that database.

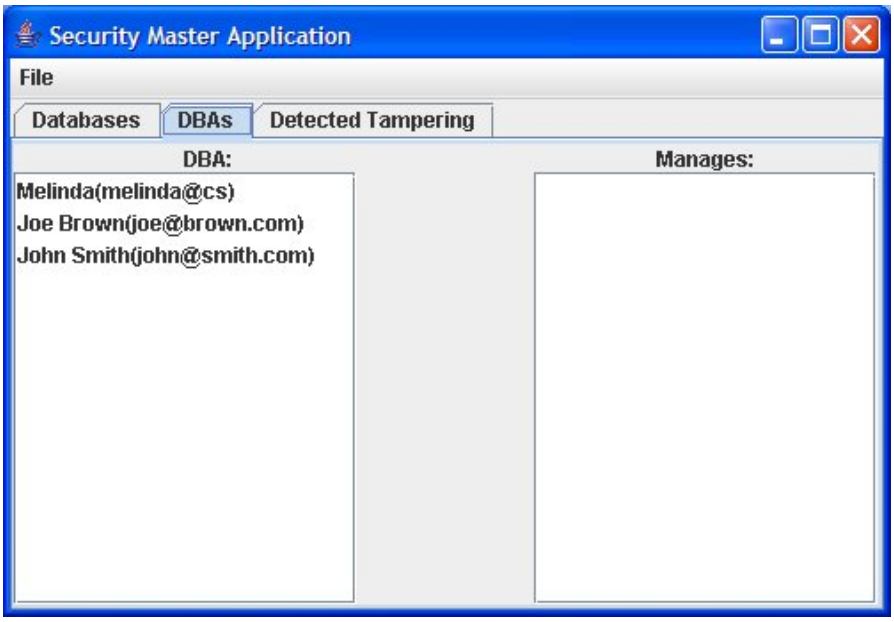

Figure 9 - Security Master Application (DBA tab)

5.1.3 Functionality/Requirements

While most of the features of the SMA that were listed above seem simple and straight-forward some of them actually require complex operations in order to implement them. For example, when adding a database to the system the SMA cannot simply trust that the given name and path will point to a valid database. In actuality, it must first check to make sure that a database does exist at the given path with the given name before adding it to the list of databases in the system. It must also create the Creation Event associated with this database. This requires the retrieval of the date this database was first created. This is done by calling a separate application that was written specifically to retrieve the date of the first tuple within a given database. Similarly, when adding an employee to the system the SMA must first confirm that the given email address is not already in use within the system. In either situation, if the data entered by the CSO is invalid a proper error message must be displayed depicting the problem to the CSO.

Another feature that is actually quite complex in implementation is the ability to set the maximum notarization and validation intervals. When the CSO sets these values

the SMA cannot simply store them in the database but instead must also check every database that is currently in the system to make sure that it conforms to the new requirements. If a database does not meet the new requirements it must be modified so that it does. In this situation, any settings of that individual database that are not compliant are changed to the new overall settings. The SMA must also have a way of verifying that it is indeed the CSO who is trying to access it. This is done using a login screen that requires the CSO's email and password before the application will load.

5.1.4 Summary

The SMA provides a clean and easy way to interact with the system. It provides many necessary features for maintaining the system and was written in such a way as to allow for easy expansions in the future. Since the application utilizes tabs to display different features it is easy to expand by simply adding more tabs to the application.

5.2 Database Monitor Application

The *DMA (Database Monitor Application)* is the tool that allows DBAs to configure and monitor the databases that they are assigned to. Each active employee in the system who has the role of a DBA is allowed to login to this application. The purpose of the DMA is to allow each database to be individually configured and monitored.

5.2.1 Features

The main feature of the DMA is the ability to set the notarization interval, validation interval, and algorithm for each database. The start date is also set and refers to when these settings should go into affect. The initial screen of the DMA is shown in Figure 10.

When a DBA logs in, the DMA is automatically populated with a list of all the databases that the DBA is currently assigned to. These can be seen in the dialog box at the top of the application in Figure 10. Upon selecting any of these databases the current configuration settings are displayed on the *Settings* tab and any tamper detections are

displayed on the *Detected Tampering* tab. The *Detected Tampering* tab can be seen in Figure 11. Also, when a database is selected the CSI that is currently assigned to it can be seen next to the drop down box.

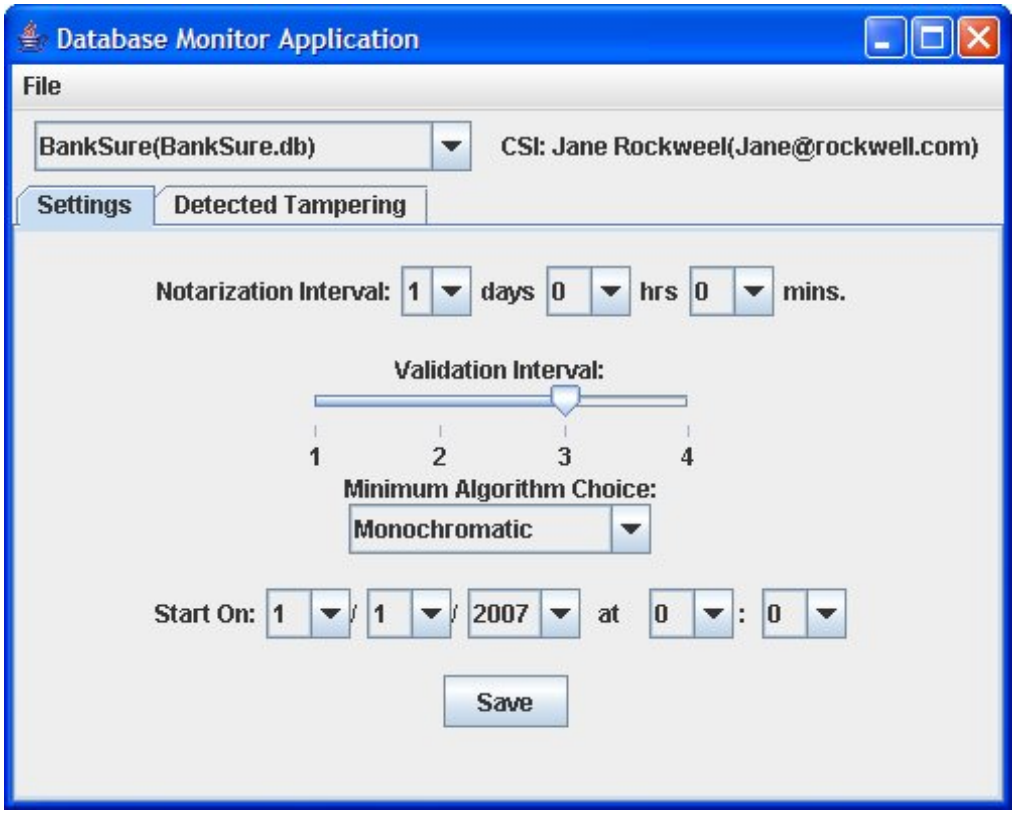

Figure 10 - Database Monitor Application (Settings tab)

5.2.2 Functionality/Requirements

One prerequisite of the DMA is that it must not allow a DBA to set any configuration settings outside the bounds outlined by the current overall requirements. If the current overall settings require that the maximum validation interval not be greater than four, the DMA must not allow a DBA to set any validation intervals to a value greater than four. The DMA does this by only populating the drop downs and slider with values that are in accordance with the current overall requirements. If a CSO later changes the maximum validation interval to six, the DMA slider would then contains values up to six.

The DMA is responsible for a very important behind-the-scenes task; it invokes the notarizer for each database. The first time that a DBA saves the configuration settings for a given database, the notarizer is then scheduled using the "Start On" date given by the DBA. Without this step the notarizer, and subsequently the validator, would never know to run on a given database. It is assumed for this first version of the application that the DBA does not change the settings twice before the notarizer is invoked. Each time the notarizer or the validator runs on a given database, it schedules its next execution time for that database according to the settings stored in the SMDB.

Similarly to the SMA, the DMA must also have a way of verifying that the employee trying to use the application is indeed a valid DBA. This is also done using a login screen that requires the DBA's email and password before the application will load.

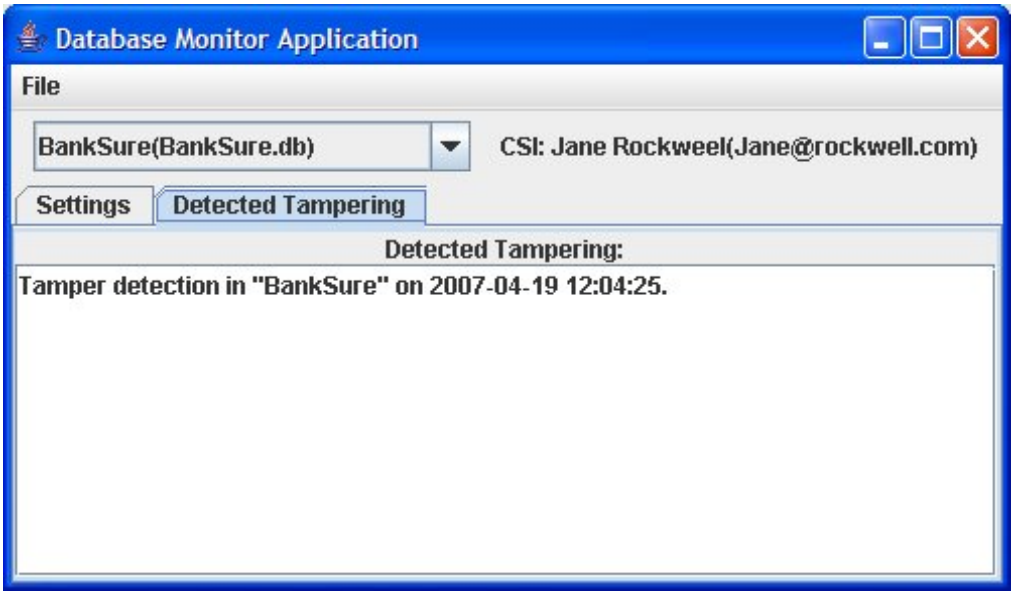

Figure 7 - Database Monitor Application (Tamper Detection tab)

5.2.4 Summary

The DMA provides all the necessary elements for a DBA to interact with the system and maintain their databases. Since it also utilizes tabs like the SMA does it can be easily expanded in future versions to incorporate new features and concepts.

5.3 Forensic Analyzer Application

When tampering has been detected, the CSI assigned to the tampered database should spring into action. This will involve the use of the *FAA (Forensic Analyzer Application).* The FAA is the tool that provides analysts with the known all known information about the intrusion. Each active employee in the system who has the role of a CSI is allowed to use this application.

5.3.1 Features

The FAA is laid out very similarly to the DMA. When an employee logs in they are given a list of all databases in which they currently have been assigned to as the CSI. They can select any of these databases and will be able to see a list of any tamper events in the system for that database. Also, when a database is selected the DBA that is currently assigned to it can be seen next to the drop down box. In the future this application will be able to display a detailed corruption diagram like the one shown in Section 2.3. The initial screen of the FAA is shown in Figure 12.

5.3.2 Functionality/Requirements

Similarly to the SMA and the DMA the FAA must also have a way of verifying that the employee trying to use the application is indeed a valid CSI. This is also done using a login screen that requires the CSI's email and password before the application will load.

5.3.3 Summary

The FAA currently provides a very basic view of tamper detections. There are many future goals for this application which are outlined in Section 6.2. For now, the application provides a means for the CSI to view all tamper detections associated with any databases they are assigned to.

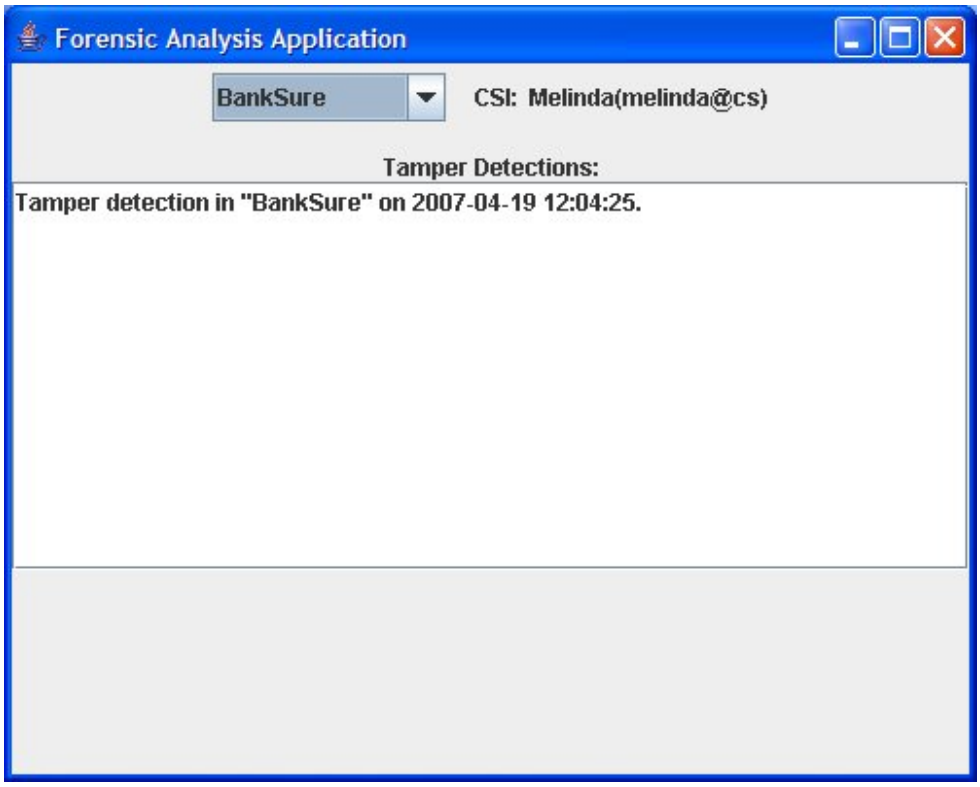

Figure 8 - Forensic Analysis Application

6. Design and Implementation of the Applications

The SMA, DMA, and FAA are all written entirely in Java and utilize both the Model-View-Controller pattern and the Observable-Observer pattern [5]. The class structure can be seen in Figure 13. Instead of having three separate classes to represent the model, view, and controller each application instead has one class, e.g. *CSOView.java*, which represents both the view and the controller and then a separate class, e.g. *CSOModel.java*, to represent the model. The view-controller class contains all of the Java swing components necessary for the graphical representation of the application and also utilizes anonymous inner classes and event listeners to handle user interaction. The model class contains all the methods necessary for performing the operations on the system and is the class that interacts with the SMDB. By using the Model-View/Controller pattern we allow the ability to create additional GUIs to interact with the system.

Each application is started from a login class, e.g. *CSOLoginDialog.java* for, which provides the user with a login dialog. This class first confirms that the information entered by the user is valid in the system and that the user currently has the role necessary to access the application and then invokes the view.

Each of the three applications uses a similar class structure. In the case of the SMA a couple of extra classes were needed for dialog boxes. A basic non-inclusive diagram of the SMA class structure is shown in Figure 13.

As described previously, the CSOLoginDialog class invokes the CSOView. The view displays the application and then interacts with the model to provide all the needed functionality. When adding an employee or database to the system the view invokes a dialog. As shown in Figure 13, the EmployeeDialog class is invoked by the view and then calls the addEmployee() method in CSOModel to store the employee in the SMDB.

The DatabaseConfig class is used by the SMA and the DMA for passing configuration settings between the model and the view. The getConnection() method in the Connector class is used by all three models to obtain a connection to the SMDB. If the location of or any of the details regarding the SMDB ever change, this is the only class that needs to be modified in order to update all three applications.

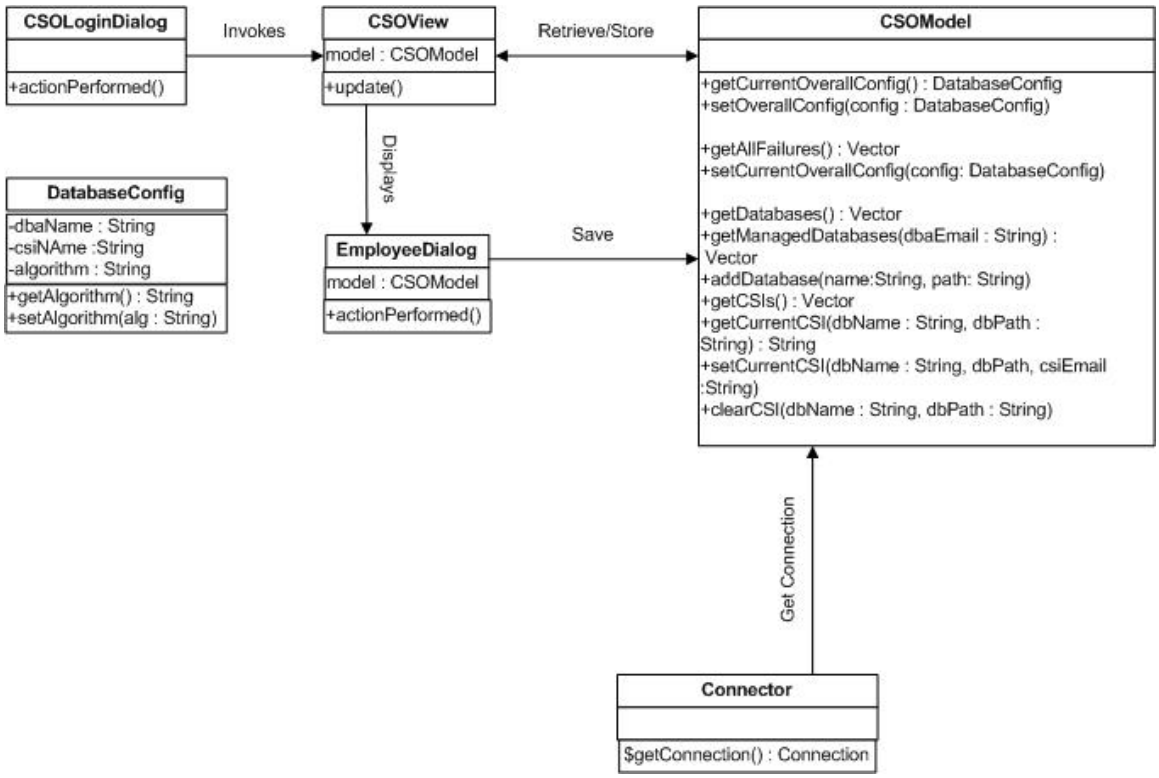

Figure 9 - UML of SMA Classes

7. Conclusion

This project consisted of six team members: three doctoral students, one master's student, a faculty member, and myself, the only undergraduate student. An integrated system was created through weekly team meetings, multiple demonstrations, and team collaboration. The overall concept of this project was described in Sections 2 and 3. My portion of this project was described in detail in the next three sections. Due to the fact that I was designing and implementing central components of the system, namely the SMDB, I was required to fully understand all the pieces of the system and how they were to fit together.

By creating a central database for all of the tools in the system to interact with it made it possible for the notarizer and validator to perform their operations successfully. They can now store their data in this central database as well as use the information stored in it to schedule future executions. The three role-specific applications allow auditing to be started on individual databases and then be maintained.

The necessary tools for auditing a database are in place. It is now possible for doctor's offices, companies, and government agencies to protect their information from threats by implementing this auditing system.

7.1 Results

A robust and well-organized database, the SMDB, was created to maintain the system as a whole and provide the central point of interaction for all tools in the system. This database is easily expandable for future versions of this project.

Three applications were created, one for each role, to allow users to interface with the SMDB. They provide an organized and controlled way for employees to interact with the system. Combined the three applications consist of fourteen different Java classes totaling roughly 3,900 lines of code. Five of these classes are used by all three applications and nine are unique to an individual application. At this point, there are many ideas for additions that can be made to these applications; some ideas are outlined in the next section.

All in all, a large step was made toward securing databases from intrusion and the maintenance of such intrusions. By utilizing a central security master database as part of an enterprise architecture for auditing, as well as role-specific GUIs, it is possible to efficiently manage the auditing of databases across an enterprise. This auditing makes it possible to protect a database from both inside and outside intruders. By using this auditing system, businesses, government agencies, and other institutions can now know that their data is secure and safe from tampering.

7.2 Future Work

At this point the SMDB provides all the necessary tables for implementing the entire system. If future algorithms are invented that require more complex chains or regions, the SMDB can be easily modified to accommodate them. However, the three applications could all be expanded to incorporate new or extended functionality.

Additions to the SMA could include a more robust employee management interface. At this point the only option available in regards to employee records is the ability to add an employee to the system. In the future it would be nice to be able to edit information about an employee and to set the stop date of an employee. The same is true for editing individual database information. Another useful addition would be the ability to view any *error events* that have occurred in the system. At this point the application only displays *tamper detection events*.

As stated before, the DMA assumes at this point that a DBA only sets the configuration settings once before notarization begins on a database. In the future the application should be expanded to incorporate the DBA changing settings multiple times and the issue of how to maintain the scheduling of the notarizer will have to be investigated.

In all three applications, but most specifically the FAA, a detailed corruption diagram would be a very useful addition. It would be nice for a CSO to be able to click on a tamper detection description in the SMA and see a corruption diagram. Also, it would be useful if employees had a way of managing their personal password and contact information.

Bibliography

[1] HIPPA, http://www.cms.hhs.gov/HIPAAGenInfo/, viewed April 20, 2007.

[2] R. T. Snodgrass, S. S. Yao, and C. Collberg, “Tamper Detection in Audit Logs” in *Proceedings of the International Conference on Very Large Databases*, pp. 504-515, Toronto, Canada, September 2004.

[3] K. Pavlou, and R. T. Snodgrass, “Forensic Analysis of Database Tampering”, Proceedings of the *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, Chicago, June 2006.

[4] Elmasri and Navathe, **Fundamentals of Database Systems**, Fifth Edition, Addison Wesley Company, 2003.

[5] Gamma, Helm, Johnson, and Vlissides, **Design Patterns: Elements of Reusable Object-Oriented Software**, First Edition, Addison Wesley Company, 1995.

A - Complete Logical Schema

Chain **Table**

ChainID	int(11) auto_increment
EventID	int(11)
Level	tinyint(4)
Color	varchar(15)
StartMilliseconds	int(11)
StopMilliseconds	int(11)
Hashcode	char(40)
NotarizationID	char(40)
StartDate	timestamp
StopDate	timestamp

Configuration **Table**

Name	char(35)
Path	char(35)
StoredDate	timestamp
NotIntDays	tinyint(4)
NotIntHrs	tinyint(4)
NotIntMins	tinyint(4)
ValidationFactor	tinyint(4)
Algorithm	varchar(35)
NumLevels	tinyint(4)
NotarizationScheduledDate	timestamp

Employee **Table**

Email	char(35)
StartDate	timestamp
Name	char(35)
Password	varchar(35)
PhoneNumber	varchar(35)
isCSI	enum('Y', 'N')
isCSO	enum('Y', 'N')
isDBA	enum('Y', 'N')
StopDate	timestamp

Event **Table**

EventID	int(11) auto_increment
Name	varchar(35)
Path	varchar(55)
Kind	enum('C','E','F','N','V')
Info	varchar(500)
atID	int(11)
ScheduledDate	timestamp
ActualDate	timestamp
ActualMilliseconds	int(11)

IsMaintainedBy **Table**

Name	varchar(35)
Path	varchar(55)
Email	varchar(35)
StartDate	timestamp
Kind	enum('A','C','D')
StopDate	timestamp

NotarizationCheck **Table**

ChainID	int(11)
FailureID	int(11)
Result	enum('P','F')

SynchronizedRegion **Table**

EventID	int(11)
TopLeftX	int(11)
TopLeftY	int(11)
TopRightX	int(11)
TopRightY	int(11)
BottomLeftX	int(11)
BottomLeftY	int(11)
BottomRightX	int(11)
BottomRightY	int(11)

UnsynchronizedRegion **Table**

EventID	int(11)
TopLeftX	timestamp
TopLeftY	timestamp
TopRightX	timestamp
TopRightY	timestamp
BottomLeftX	timestamp
BottomLeftY	timestamp
BottomRightX	timestamp
BottomRightY	Timestamp
TopLeftXMillisecond	int(11)
TopLeftYMillisecond	int(11)
TopRightXMillisecond	int(11)
TopRightYMillisecond	int(11)
BottomLeftXMillisecond	int(11)
BottomLeftYMillisecond	int(11)
BottomRightXMillisecond	int(11)
BottomRightYMillisecond	int(11)