Title: Design / Installation Information for TigerScan	**Version:** 1.1

NO APPROVALS REQUIRED

Table of Contents

1 Purpose

This document presents the Solution Development Lifecycle (SDLC) Design / Installation Information for the Team Tiger Email Security Violation Scanner (TigerScan). This document provides specific details about each module of the software and how it contributes to the overall operations of the software.

2 Scope

This document covers the entirety of TigerScan. This includes all modules that make up the software and information on the database that will be accessed by the program.

2.1 Exclusions, Assumptions, and Limitations

It is assumed that terms have been correctly added into the database by Administrators. It is also assumed that the input .TXT files correctly match the email that will be scanned.

3 Solution Design Overview

TigerScan scans an email's contents by taking in and reading the .TXT version of the email. This .TXT document is added into a "scan list" via the GUI. When "Scan All" is selected, the GUI goes through each file that has been added and grades the files by the confidentiality score of each term in the file. The score is then presented to the user through the GUI.

Administrators have the ability to add and remove terms from the database. Administrators are assigned a username and password. After successfully logging in, the administrator can then use the various buttons on the left hand side of the window to add, remove, rename, and reclassify terms. Administrators may also import terms by way of a .CSV document that contains terms and their respective scores, with each value separated by commas. A table on the right hand side of the window displays the terms currently held inside of the database.

4 Technical Architecture

4.1 Hardware Inventory, Specifications and Locations

4.1.1 Servers

N/A

4.1.2 Input / Output Devices

All I/O operations are handled by the user's machine. The operations are performed by the desktop while being presented on the monitor.

4.1.3 Other Devices

N/A

4.1.4 Infrastructure/Application Diagram

N/A

4.1.5 Middleware Hosting

Database will be locally hosted on the user's machine. Although there is a default database created upon the first execution of TigerScan, a database can be saved to any location on the user's machine.

4.2 Interfaces with Other Hardware and External Integration Points

N/A

4.3 Physical Layout

N/A

4.4 Additional Information

N/A

5 Configuration Specification

- TigerScan is contained in a .JAR executable.
- TigerScan can be run on any OS, provided the Java Runtime Environment has previously been installed.

6 Solution Design Specification

6.1 Software Description

This program uses the following modules:

- Database – Contains classified terms and their associated classification scores.
- GUI – Allows the user to add files to be scanned and displays the resulting score.
- Lucene -- Third party software that is used to assist with scanning .TXT documents.
- Scorer -- Tracks and updates the confidentiality score of a term and returns a total grade for a document.
- Content Scanner -- Utilized by the previous four modules in order to allow a user to scan an email and return a confidentiality score for that email.

6.2 Coding Standards

No specific coding standards have been practiced during development of the source code.

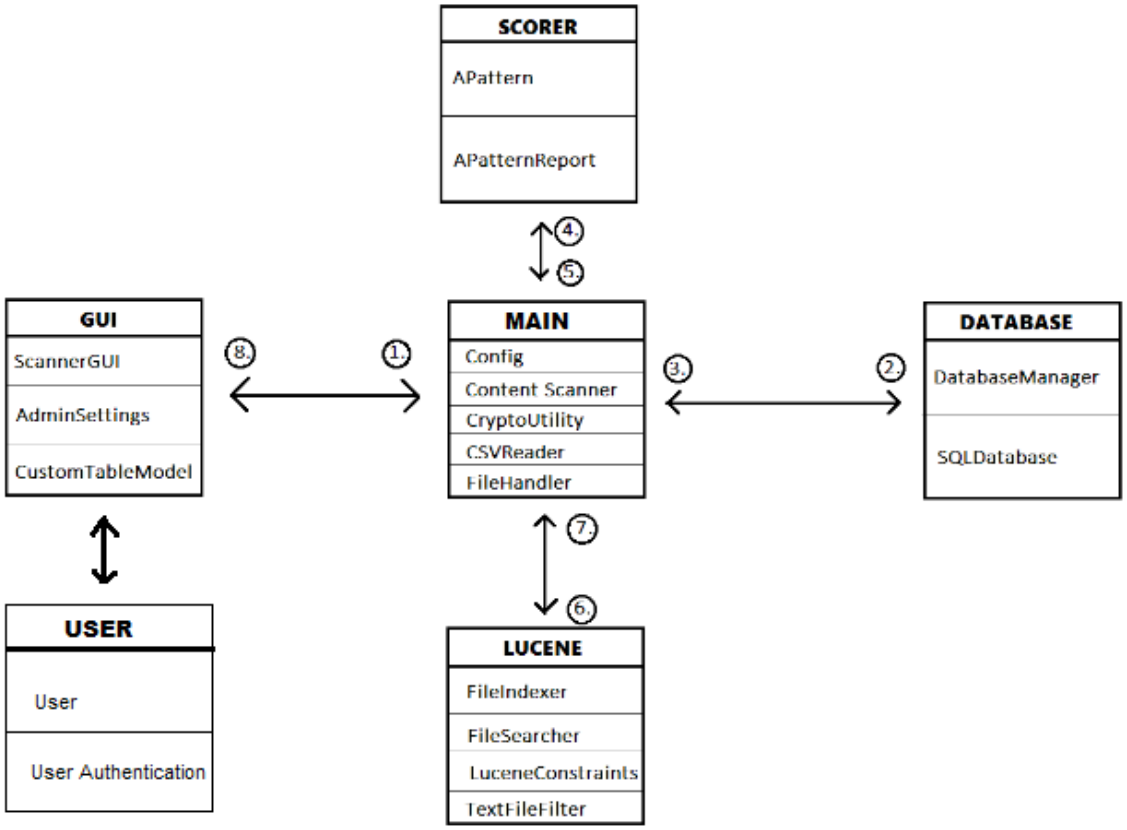
6.3 Solution Data, Information View, and Data Requirements

- Database
  - Default name of the database is “database.db” and is located in a folder called “data”, which will be created in the directory the .JAR executable is held in.
  - The database utilizes SQLite, and so TigerScan does not need external connections in order to communicate with the database.
- Emails
  - Emails that will be scanned must first be saved as a .TXT document on the user’s machine. TigerScan will not be able to scan the email otherwise.
- Result of scan
  - The result of a scan by TigerScan is presented as a value of type double by the GUI.

6.4 Module Description

All modules have been written in Java. Each module is a Java package in the source code.

Below is a data flow diagram that shows the interactions between each module. Each module contains the names of the Java classes (.java files) that make up the module. Numbers 1-8 represent each subsequent step in the operation of TigerScan. (User has not been numbered, as it is not essential to TigerScan's main operation).



Below are tables that detail each Java class in TigerScan. This includes descriptions and I/O parameters of all methods in each class.

DATABASE

DatabaseManager	**Methods**
This class is responsible for maintaining the database by handling the adding and removing of terms.	public DatabaseManager(): *Parameters:* In: N/A
Checks to see if the database contains a certain term.	public boolean hasTerm(String term) *Parameters:* In: ● String term ○ Term that is being searched for. Out: ● boolean value that is set to “true” if the database contains the given term
Adds a term to the database *Error Checking:* Throws a DatabaseAddTermException if the term already exists or if the value is not in the range of 0-100.	public void addTerm(String term, int value) *Parameters:* In: ● String term ○ Term that will be added to the database ● String value ○ Associated confidentiality score for the given term. Out: N/A
Adds multiple terms to the database	public void addTerm(HashMap<String,Integer>

Error Checking: Throws a DatabaseAddTermException if the term already exists or if the value is not in the range of 0-100.	values) *Parameters:* In: ● HashMap<String,Integer> values ○ Map where the "key" is a term and its associated "value" is the numerical value of the term's confidentiality score. Out: N/A
Retrieves the score of a particular term *Error Checking:* Throws a DatabaseNoSuchTermException if the term does not exist in the database.	public int getScore(String term) *Parameters:* In: ● String term ○ Term that the program is searching for Out: ● int value of the given term's confidentiality score.
Removes a term from the database. *Error Checking:* Throws a DatabaseRemoveTermException if the term does not exist	public void removeTerm(String term) *Parameters:* In: ● String term ○ Term that the program is searching for Out: N/A
Removes one or more terms from the database.	public void removeTerm(ArrayList<String> termArray)

Error Checking: Throws a DatabaseRemoveTermException if the term does not exist	*Parameters:* In: ● ArrayList<String> termArray ○ List of terms that will be removed from the database Out: N/A
Removes all terms from the Database.	public void removeAllTerms() *Parameters:* In: N/A Out: N/A
Change/update the score of a term in the database *Error Checking:* Throws a DatabaseAddTermException if the value is not in the range of 0-100. SQLException is also thrown if the database encounters an error.	public void changeScore(String term, int score) *Parameters:* In: ● String term ○ Term that the program is searching for ● int score ○ New confidentiality score that will be associated with the given term. Out: N/A
Gets a HashMap<String, Integer> of terms in the database	public HashMap<String,Integer> getTerms() *Parameters:* In: N/A Out: HashMap that holds every term as a String and every confidentiality value as an Integer.
Gets the number of emails a term has appeared in since it was added	public int getNumbEmailsIn(String term) *Parameters:*

	In: ● String term ○ Term that the program is searching for Out: int value of the number of emails that hold the given term.
Increment the frequency of a term in the database	public void incrementNumbEmailsIn(String term) *Parameters:* In: ● String term ○ Term that the program is searching for Out: N/A
Gets the number of emails a term has not appeared in since it was added	public int getNumbEmailsNotIn(String term) *Parameters:* In: ● String term ○ Term that the program is searching for Out: int value of the number of emails the term has not been seen in.
Increment the frequency of a term in the database	public void incrementNumbEmailsNotIn(String term) *Parameters:*

	In: ● String term ○ Term that the program is searching for Out: N/A
Get the average probability of a term	public double getAverageProbability(String term) *Parameters:* In: ● String term ○ Term that the program is searching for Out: double value that represents the average probability of the given term.
Set the average probability of a term	public void setAverageProbability(String term, double prob) *Parameters:* In: ● String term ○ Term that the program is searching for ● double prob ○ New value of the average probability that will be set for the given term. Out: N/A
Get the probability any email is confidential based on a term	public double getProbabilityAny(String term) *Parameters:* In: ● String term

	○ Term that the program is searching for Out: double value that represents the probability of an email is confidential if it contains the given term
Set the probability any email is confidential based on a term	public void setProbabilityAny(String term, double prob) *Parameters:* In: ● String term ○ Term that the program is searching for ● double prob ○ New value of the probability that an email is confidential. Out: N/A
Get the filename of the database *Error Checking:* Throws an SQLException if the database encounters an error.	public String getDatabaseFilename() *Parameters:* In: N/A Out: String value that represents the filename of the database.
Set the filename of the database *Error Checking:* Throws an SQLException if the database encounters an error.	public void setDatabaseFilename(String filename) *Parameters:* In: ● String filename ○ New filename that will be

	associated with the current database. Out: N/A
Calls the SQLDatabase method to initialize the SQL connection to the database file	public void initSQLConnection() *Parameters:* In: N/A Out: N/A
Calls the SQLDatabase method to close SQL connection to the database file	public void closeSQLConnection() *Parameters:* In: N/A Out: N/A

SQLDatabase	**Methods**
Create SQLDatabase object and initialize connection to the database file *Error Checking:* Throws an SQLException if TigerScan is unable to connect to the database.	public SQLDatabase(String databaseFileName): *Parameters:* In: ● String databaseFileName ○ filename of the database that TigerScan will connect to
Initialize connection to the SQL Database *Error Checking:* Throws an SQLException if TigerScan is unable to connect to the database.	public void initConnection() *Parameters:* In: N/A Out: N/A
Closes connection to the SQLDatabase	public void closeConnection()

	Parameters: In: N/A Out: N/A
Ensures that the database file has the proper SQL table for storing terms. If a file is not empty, but the table is invalid or missing, the database is considered corrupted and should be removed for the program to recreate it *Error Checking:* SQlException if unable to init.	public void initTable() *Parameters:* In: N/A Out: N/A
Performs a SQL INSERT operation on the database *Error Checking:* SQLException thrown if unable to add to table.	public void addTerm(String term, int score) *Parameters:* In: ● String term ○ Term that will be inserted into the database ● int score ○ Confidentiality score associated with the given term Out: N/A
Performs a SQL DELETE operation on the database *Error Checking:* SQLException thrown if unable to remove from table.	public void removeTerm(String term) *Parameters:* In: ● String term ○ Term that will be removed from the database Out: N/A
Performs a SQL DELETE operation on the database (removes all terms). Maintains the table and columns (SQLite doesn't support DELETE * or TRUNCATE) *Error Checking:* SQLException thrown if unable to remove terms.	public void removeAll() *Parameters:* In: N/A Out: N/A
Performs a SQL UPDATE operation on the	public void changeScore(String term, int score)

database to change a term's score *Error Checking:* SQLException thrown if unable to change score.	*Parameters:* In: ● String term ○ Term that will be updated ● int score ○ New confidentiality score for the given term Out: N/A
Performs a SQL SELECT operation to query the database for all terms and scores *Error Checking:* SQLException if unable to get terms.	public HashMap<String,Integer> getTerms() *Parameters:* In: N/A Out: HashMap that contains all terms as “keys” and each term’s associated confidentiality scores as “values”
Get the number of emails a term has been found in (since it was added) *Error Checking:* SQLException if unable to get terms.	public int getNumbEmailsIn(String term) *Parameters:* In: ● String term ○ Term that the database will search for Out: int value of how many emails the given term has appeared in.
Get the number of emails a term has not been found in (since the term as added) *Error Checking:* SQLException if unable to get terms.	public int getNumbEmailsNotIn(String term) *Parameters:* In: ● String term ○ Term that the database will search for Out: int value of how many emails the given term has not appeared in.

Increment the number of emails a word was found in *Error Checking:* SQLException if unable to get terms.	public void incrementNumbEmailsIn(String term) *Parameters:* In: ● String term ○ Term that will be incremented Out: N/A
Increment the number of emails a word was found in *Error Checking:* SQLException if unable to get terms.	public void incrementNumbEmailsNotIn(String term) *Parameters:* In: ● String term ○ Term that will be incremented Out: N/A
Get the average probability of a specified term *Error Checking:* SQLException if unable to get terms.	public double getAverageProbability(String term) *Parameters:* In: ● String term ○ Term associated with the probability in question Out: double value that represents the average probability of the given term
Set the new average probability of a specified term after calculation *Error Checking:* SQLException if unable to get terms.	public void setAverageProbability(String term, double prob) *Parameters:* In: ● String term ○ Term that will be updated ● double prob ○ New average probability of the given term

	Out: N/A
Get the probability any email is confidential based on a term *Error Checking:* SQLException if unable to get terms.	public double getProbabilityAny(String term) *Parameters:* In: ● String term ○ Term that affects the probability of having a confidential email Out: double value that represents the probability of an email is confidential if it contains the given term
Set the new probability that an email is confidential *Error Checking:* SQLException if unable to get terms.	public void setProbabilityAny(String term, double prob) *Parameters:* In: ● String term ○ Term that affects confidentiality of an email ● double prob ○ New value of the probability any email is confidential if it holds the given term Out: N/A
Returns the filename of the database	public String getDatabaseFileName() *Parameters:* In: N/A Out: String representation of the database filename
Set the file name of the database and re-initialize	public void setDatabaseFileName(String filename)

connection to it *Error Checking:* SQLException thrown if TigerScan is unable to change the database filename.	*Parameters:* In: ● String filename ○ New filename for the current database Out: N/A

DBHash	**Methods**
This class hosts the hashCode method that will be used by the database.	No constructor.
This will add salt to a String and return the hashed version.	public static int hashCode(String term) *Parameters:* In: ● String term ○ term to be hashed Out: Hashed version of given term in String form.

GUI

ScannerGUI	**Methods**
This class is the main piece of the software's View. It is called by Main.	public ScannerGUI(ContentScanner scanner, DatabaseManager db) *Parameters:* In: ● ContentScanner scanner ○ Instantiation of the ContentScanner class that will allow the user to scan an email ● DatabaseManager db ○ Instantiation of the

	DatabaseManager class that will allow TigerScan to interact with the database.
Initializes the UI	private void initializeUI() *Parameters:* In: N/A Out: N/A
Creates the GUI's main panel	private JPanel mainPanel() *Parameters:* In: N/A Out: JPanel that serves as the GUI's main panel.
Creates the GUI's north panel	private JPanel northPanel() *Parameters:* In: N/A Out: JPanel that serves as the GUI's north panel.
Creates the GUI's center panel	private JPanel centerPanel() *Parameters:* In: N/A Out: JPanel that serves as the GUI's center panel.
Creates the GUI's south panel	private JPanel southPanel() *Parameters:* In: N/A Out: JPanel that serves as the GUI's south panel.
Retrieves the list of files that need to be scanned	public ArrayList<String> getFilesToScan()

	Parameters: In: N/A Out: ArrayList that holds the String representations of each file to be scanned
Creates the AdminSettings window	public void createAdminDialog(DatabaseManager db) *Parameters:* In: ● DatabaseManager db ○ This is the currently used DatabaseManager. By passing this, AdminSettings will be able to access the same database as the ScannerGUI. Out: N/A

AdminSettings	**Methods**
This class is responsible for creating the AdminSettings panel. It is only called by the ScannerGUI.	public AdminSettings(DatabaseManager db) *Parameters:* In: ● DatabaseManager db ○ DatabaseManager that is currently being used by the ScannerGUI

CustomTableModel	**Methods**
This class is the model for the AdminSettings' table. This is only called within AdminSettings.	public CustomTableModel(HashMap<String, Integer> hashMap) *Parameters:* In: ● HashMap<String,Integer> hashMap

	○ HashMap that contains the values of the table
Retrieves the number of rows in the table.	public int getRowCount() *Parameters:* In: N/A Out: int value of the number of rows in the table
Retrieves the number of columns in the table.	public int getColumnCount() *Parameters:* In: N/A Out: int value of the number of columns in the table
Retrieves the name of a column.	public String getColumnName(int col) *Parameters:* In: ● int col ○ Number of the column in question Out: String value of the name of the given column number
Retrieves the value at a cell.	public Object getValueAt(int row, int col) *Parameters:* In: ● int row ○ row that the cell is located ● int col ○ column that the cell is located Out: The value that is currently held in the specified cell.

Sets the value of a cell in the table	public void setValueAt(Object value, int row, int col) *Parameters:* In: ● Object value ○ Value of any data type that will be inserted into the given cell ● int row ○ row that the cell is held in ● int col ○ column the the cell is held in Out: N/A
Clears all cells in the table	public void clearAllCells() *Parameters:* In: N/A Out: N/A
Gets a row's index	public int getSelectedRowIndex() *Parameters:* In: N/A Out: int value of a row's index
Sets a row's index	public void setSelectedRowIndex(int row) *Parameters:* In: ● int row ○ row number of selection Out: N/A

LoginDialog	**Methods**

This dialog box is used as an interface for the user to log into the system through	public LoginDialog(JPanel relativePanel, URL icon_url, Runnable action) *Parameters:* In: ● JPanel relativePanel ○ JPanel that the dialog will be positioned around ● URL icon_url ○ URL of the icon that is associated with this dialog. ● Runnable action ○ Used to run the AdminSettings panel Out: N/A
Return the user created by the dialog	public User getUser() *Parameters:* In: N/A Out: User object of the logged-in user

NewUserDialog	**Methods**
This dialog box is intended to be used when the users file was not present and the default administrator credentials have been used for the first time	public NewUserDialog(JPanel relativePanel, URL icon_url) *Parameters:* In: ● JPanel relativePanel ○ JPanel that the dialog will be positioned around ● URL icon_url ○ URL of the icon that is associated with this dialog. Out: N/A

Return the user created by the dialog	public User getUser() *Parameters:* In: N/A Out: User object of the logged-in user

ShowLogDialog	**Methods**
This dialog box serves the purpose of displaying the event log associated with the program	public ShowLogDialog(JPanel relativePanel, URL icon_url) *Parameters:* In: ● JPanel relativePanel ○ JPanel that the dialog will be positioned around ● URL icon_url ○ URL of the icon that is associated with this dialog. Out: N/A

LUCENE

FileIndexer	**Methods**
This class takes text files and creates indexes from them that can be searched *Error Checking:* IOException thrown if the IndexWriter encounters an error.	public FileIndexer(String indexDirectoryPath) *Parameters:* In: ● String indexDirectoryPath ○ Path of the directory that holds the text files.
Method that creates a document that can be analyzed	public Document addDoc(File f) *Parameters:*

Error Checking: IOException thrown if the Document cannot be created.	In: ● File f ○ File that will be converted Out: Document conversion of the given file
Closes the IndexWrtier	public void closeIndex() *Parameters:* In: N/A Out: N/A
Creates a Document then adds the Document to the IndexWriter *Error Checking:* IOException thrown if the IndexWriter cannot use the Document.	public void indexFile(File f) *Parameters:* In: ● File f ○ File that will be converted and sent to IndexWriter Out: N/A
Creates the index using a file path of where the data is located and a file filter *Error Checking:* IOException thrown if the IndexWriter encounters an error.	public int createIndex(ArrayList<String> fileNames, FileFilter filter) *Parameters:* In: ● ArrayList<String> fileNames ○ List of the files that will be indexed. ● FileFilter filter ○ Used to filter any files for a specific file extension. Out: int value of the index

FileSearcher	**Methods**
This class searches on the Index Directory created by FileIndexer *Error Checking:* IOException thrown if IndexReader is unable to open the given path.	public FileSearcher(String indexDirPath) *Parameters:* In: ● String indexDirPath ○ Path of the index directory
Performs a search on the specified query through all documents in the index. *Error Checking:* IOException thrown if IndexSearcher is unable to search. ParseException thrown if the QueryParser encounters an error with the search query.	public TopDocs search(String searchQuery) *Parameters:* In: ● String searchQuery ○ Files will be scanned for this query and added to the output TopDocs object Out: TopDocs object that holds information on number of hits and which files contain the search query
Grabs documents that had hits for the search query. *Error Checking:* IOException thrown if the IndexSearcher cannot find the Document.	public Document getDocument(int docID) *Parameters:* In: ● int docID ○ ID number of the document Out: Document object of the file that had the query in its data.
Stems the given term down to its root word.	public String stemTerm (String term) *Parameters:* In: ● String term ○ Term that will be shortened to a root word

	Out: String representation of the given term's root word.

LuceneConstraints	Methods
This class holds constant values to be used for Fields in the FileIndexer class	N/A

TextFileFilter	Methods
This class filters only files that are .txt files.	public boolean accept(File pathname) *Parameters:* In: ● File pathname ○ File that will be checked against the filter Out: boolean value that is "true" if the file has the correct extension.

SCORER

APattern	Methods
This class will handle analysis of the information to determine how likely it is that an email is confidential. This is a modification of Bayes Spam Filtering	public APattern() *Parameters:* In: N/A
Report that a word with confidentially rating p has	public void addWord(String word, double weight,

been added. *Error Checking:* APatternException is thrown if the email has already been scanned.	double aC, int numberOfEmailsWordIsIn, int numberOfEmailsWordIsNotIn, double pConf) *Parameters:* In: ● String word ○ word to be added to the database ● double weight ○ value determined by the user that will affect the confidentiality score ● double aC ○ the average probability that emails with the given word are confidential ● int numberOfEmailsWordIsIn ○ Number of emails that the word has appeared in ● int numberOfEmailsWordIsNotIn ○ Number of emails that the word has not appeared in ● double pConf ○ Probability that any given email is confidential Out: N/A
This will calculate the total probability that an email is confidential *Error Checking:* APatternException is thrown if the email has already been scanned.	public APatternReport calculateProbability() *Parameters:* In: N/A Out: Returns a report of the current probability that an email is confidential

APatternReport	**Methods**
This class is used by APattern to send results from the email evaluation back to the calling class	public APatternReport(double pConfidentialOfThisEmail) *Parameters:*

	In: ● double pConfidentialOfThisEmail ○ the probability that the current email is confidential
This is used by the APattern class to add a word with its new values. The Class that receives this instance should get these words and adjust their values accordingly *Error Checking:* APatternException is thrown if APatternReport has been locked.	public void addWordAndSetValues(String word, double averagePConfidential, double pC) *Parameters:* In: ● String word ○ Word to be added ● double averagePConfidential ○ New value for the average probability of emails containing the word are confidential ● double pC ○ New value for the probability that any given email is confidential Out: N/A
Locks the program from using APatternReport	public void lock() *Parameters:* In: N/A Out: N/A
Get the confidentiality score of this email	public double getConfidentialityScoreOfThisEmail() *Parameters:* In: N/A Out: Returns the confidentiality score of the email in question in a double value.
Get the String word at index i.	public String getWord(int i)

	Parameters: In: ● int i ○ Index number Out: Returns the String at the given index
Get the number of words	public int getNumbWords() *Parameters:* In: N/A Out: int value that holds the number of words
Get the average probability that an email which is confidential has the word at index i.	public double getAverageProbabilityConfidential(int i) *Parameters:* In: ● int i ○ Index number of the word Out: double value that represents the average probability of confidentiality of emails with the term at index i.
Get the probability that any email is confidential (word per word basis)	public double getProbabilityConfidentialPerWord(int i) *Parameters:* In: ● int i ○ Index number Out: double representation of the probability that any given email is confidential

MAIN

Config	**Methods**
Configuration medium to interface with a config file that contains the current database file name and total number of e-mails the program has scanned *Error Checking:* IOException thrown if the BufferedWriter encounters an error.	private static void updateConfigFile() *Parameters:* In: N/A Out: N/A
Initialize the config attributes, reading from the config file or creating one and utilizing default values	public static void initConfig() *Parameters:* In: N/A Out: N/A
Called when an e-mail is scanned, this method will simply increment the number of e-mails scanned *Error Checking:* IOException thrown if updateConfigFile() encounters an error.	public static void emailScanned() *Parameters:* In: N/A Out: N/A
return total number of e-mails scanned	public static int getEmailsScanned() *Parameters:* In: N/A Out: N/A
Sets a new value for the database file name *Error Checking:* IOException thrown if updateConfigFile() encounters an error.	public static void setDatabaseFilename(String filename) *Parameters:* In: ● String filename

	○ New filename for the database Out: N/A
return String current database file name	public static String getDatabaseFilename() *Parameters:* In: N/A Out: String object that holds the value for the filename of the database.

ContentScanner	**Methods**
This class handles the scanning of a given file's text.	public ContentScanner(DatabaseManager db) *Parameters:* In: ● DatabaseManager db ○ This allows ContentScanner to interact with terms that have been added to the database
Scans every file that has been added into the GUI's list	public HashMap<String,Double> scanFiles(ArrayList<String> importedFileNames) *Parameters:* In: ● ArrayList<String> importedFileNames ○ Filenames of files that are to be scanned.

	Out: HashMap that holds the flagged terms as "keys" and their respective confidentiality scores as "values"
Searches a file for a specific search query. *Error Checking:* IOException thrown if FileSearcher encounters an error. ParseException thrown if the search query is unable to be parsed with.	private void search(String searchQuery) *Parameters:* In: ● String searchQuery ○ Query that will be searched for in a file Out: N/A
Creates an index for a list of files *Error Checking:* IOException thrown if the the FileIndexer is unable to create the index.	private void createIndex(ArrayList<String> filenames) *Parameters:* In: ● ArrayList<String> filenames ○ List containing the filenames of files that are to be indexed. Out: N/A

CryptoUtility	**Methods**
Cryptography utility designed for the AES encryption and decryption	No constructor.
Encrypt a String with AES cipher	public static String encryptString(String s) *Parameters:* In:

	● String s ○ String that is to be encrypted Out: String value that holds the encrypted version of the given String
Decrypt a String with AES cipher	public static String decryptString(String s) *Parameters:* In: ● String s ○ String to be decrypted Out: Decrypted version of the given String in a String value.

CSVReader	**Methods**
Reads a given CSV file for terms and confidentiality scores	No constructor.
Retrieves each line from a file	public static ArrayList<String> getLinesFromFile(String filename) *Parameters:* In: ● String filename ○ Filename of the file to be read. Out: ArrayList of the lines of the file. In other words, each line is held as one String object in the list.
Separate a line by commas	public static String[] getContentFromLine(String line) *Parameters:* In: ● String line

	○ Line that is to be split by commas. Out: Array of values that have been read from the line. Each time a comma is detected, the array ignores the comma and creates a new object of type String for the next value.

FileHandler	**Methods**
This class currently handles reading in and writing out text files. It also has read/write methods for use of the Database that includes encryption	No constructor.
This will get the contents of a file as a String *Error Checking:* FileNotFoundException thrown if the method cannot find the file with the given filename.	public static String getStringFromFile(String fileName) *Parameters:* In: ● String fileName ○ Filename of the file that will be handled by the FileHandler Out: Retrieves the contents of the file as a String
This will write a String to a specified file	public static void writeStringToFile(String contents, String fileName) *Parameters:* In: ● String contents ○ String representation of contents that are to be written ● String fileName ○ Filename of the file that the contents will be written to.

	Out: N/A

ConsoleLauncher	**Methods**
This class is used with the terminal (non-GUI) version of TigerScan.	public ConsoleLauncher(String[] args, ContentScanner scanner, String version) *Parameters:* In: ● String[] args ○ Arguments provided by the user ● ContentScanner scanner ○ Performs file scanning (see entry in MAIN module) ● String version ○ Version of TigerScan Out: N/A
Performs the basic TigerScan function of scanning files. Returns “true” when complete.	private boolean scanFiles(ArrayList<String> filenames) *Parameters:* In: ● ArrayList<String> filenames ○ List that contains the filenames of each file that is to be scanned Out: Boolean value set to “true” when the scan successfully completes.

Event Log	**Methods**
This class is meant to write information to a log file for runtime documentation	No constructor.
Write one or more lines of text to the log file	private static void write(String[] lines)

	Parameters: In: ● String[] lines ○ Array of lines that will be sequentially written to the log. Out: N/A
Clear the log file	public static void clear() *Parameters:* In: N/A Out: N/A
Write to the log file to reflect that the program has scanned a file	public static void writeScanned(ArrayList<String> scannedFiles) *Parameters:* In: ● ArrayList<String> scannedFiles ○ List of scanned files, represented by their String filenames Out: N/A
Write to the log file a filename and its associated score.	public static void writeScannedScore(String filepath, double score) *Parameters:* In: ● String filepath ○ String representation of the file's location. ● double score ○ Score associated with the given filepath Out: N/A
Write to the log file to reflect that a term has been	public static void writeTermAdded(String

added to the database	dbFilename) *Parameters:* In: ● String dbFilename ○ Filename of the database Out: N/A
Write to the log file to reflect that a term has been renamed from the database	public static void writeTermRenamed(String dbFilename) *Parameters:* In: ● String dbFilename ○ Filename of the database Out: N/A
Write to the log file to reflect that a term has been removed from the database	public static void writeTermRemoved(String dbFilename) *Parameters:* In: ● String dbFilename ○ Filename of the database Out: N/A
Write to the log file to reflect that all terms have been removed from the database	public static void writeTermRemovedAll(String dbFilename) *Parameters:* In: ● String dbFilename ○ Filename of the database Out: N/A
Write to the log file to reflect that a term's classification has been changed	public static void writeTermClassificationChanged(String

	dbFilename) *Parameters:* In: ● String dbFilename ○ Filename of the database Out: N/A
Write to the log file to reflect that terms have been imported to the database	public static void writeTermImported(String dbFilename, String importedFilename) *Parameters:* In: ● String dbFilename ○ Filename of the database ● String importedFilename Out: N/A
Write to the log file to reflect that the database has been renamed	public static void writeDatabaseRenamed(String dbFilenameOld, String dbFilenameNew) *Parameters:* In: Out: N/A
Write to the log file to reflect that a new database has been selected	public static void writeDatabaseChanged(String dbFilenameOld, String dbFilenameNew) *Parameters:* In: ● String dbFilenameOld ○ Old filename of the database ● String dbFilenameNew ○ New filename of the database Out: N/A
Indexes files and writes the time elapsed to the	public static void writeFileIndexed(int

log.	numIndexed, long startTime, long endTime) *Parameters:* In: ● int numIndexed ○ int value of the number of indexed files ● long startTime ○ long value of the start of the operation ● long endTime ○ long value of the end of the operation Out: N/A
Searches documents based on a query and outputs the results to the log.	public static void writeDocHits(String searchQuery, TopDocs hits) *Parameters:* In: ● String searchQuery ○ Query for the document search ● TopDocs hits ○ Documents that match the given query Out: N/A
Writes changes to the Admin credentials to the log	public static void writeAdminCredentialsChanged(String username) *Parameters:* In: ● String username ○ Username of the Admin Out: N/A
Returns the filename of the log.	public static String getLogFilename()

	Parameters: In: N/A Out: String that holds the value for the log's filename

AUTH

User	**Methods**
This class stores is an object representation of a user.	public User(String username, boolean admin) *Parameters:* In: ● String username ○ String representation of the user's username ● boolean admin ○ Set to "true" if the user has Admin privileges. Out: N/A
Returns a boolean value that is "true" if the user is an Admin.	public boolean isAdmin() *Parameters:* In: N/A Out: boolean representation of Admin status. "True" if the user is an Admin.

UserAuthentication	**Methods**
This class will handle User authentication to differentiate users from administrators and apply necessary local security restrictions	No constructor

Attempt to log in with supplied credentials and return matching User object.	public static User login(String username, char[] passwordAttempt) *Parameters:* In: ● String username ○ String representation of the user's username ● char[] passwordAttempt ○ Given password with each letter as a value in a char[] array. Immediately converted to a String. Out: User object that contains data of newly logged-in user (upon success).
Create a new administrator with submitted credentials and record it in the users file	public static User newDefaultAdmin(String username, char[] pass1, char[] pass2) *Parameters:* In: ● String username ○ String representation of the user's username ● char[] pass1 ○ Given password with each letter as a value in a char[] array. Immediately converted to a String. ● char[] pass2 ○ Confirmation of first password Out: User object that contains data of newly logged-in user (upon success).
Verify that the information entered is that of the system's default administrator.	public static boolean verifyDefaultAdmin(String username, char[] passwordAttempt) *Parameters:* In:

	● String username ○ String representation of the user's username ● char[] passwordAttempt ○ Given password with each letter as a value in a char[] array. Immediately converted to a String. Out: boolean value that is "true" if the given user credentials match the default administrator's credentials.
Check the password attempt during login process against the saved password and return true if it matches.	private static boolean passwordMatch(String passwordAttempt, String password) *Parameters:* In: ● String passwordAttempt ○ Given password during log-in. ● String password ○ Password on file for user Out: boolean value that is "true" if inputs are identical.
Check the String the user tries to use to create a new password to ensure it matches security requirements	private static boolean validPassword(String password) *Parameters:* In: ● String password ○ Given password Out: boolean value that is "true" if given password matches security requirements.
Return a String that has been salted to enhance security of authentication	private static String saltedString(String original) *Parameters:* In:

	● String original ○ String representation of the original string Out: Salted version of the given String.

7 Roles and Responsibilities

There are no additional roles specific to this document.

8 Terms and Definitions

Term or Acronym	Definition
SQL	Structured Query Language
SQLite	Version of SQL that does not require a program to create a connection to an external database. Rather, SQLite is embedded within the compiled program.
CSV	Comma Separated Values
Java package	Collection of Java classes
JAR	Java Archive

9 Supporting References

There are no supporting references specific to this document.

10 Revision History

Version	Version Date	Revisions
1.0		Initial release.
1.1	12/13/16	Added AUTH module, changed CONTENT SCANNER to MAIN. Added information on new methods for other modules.