Design and Verification of Privacy and User Re-authentication Systems

Harini Jagadeesan

M.S. Thesis submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
In
Computer Engineering

Michael S. Hsiao, Chair
Chao Huang
Paul Plassmann

April 30, 2009
Bradley Department of Electrical and Computer Engineering
Virginia Polytechnic Institute and State University
Blacksburg, Virginia

Keywords: user re-authentication, biometrics, security, mouse
keyboard, behavioral, testing, verification, Spec#

Design and Verification of Privacy and User Re-authentication Systems

Harini Jagadeesan

ABSTRACT

In the internet age, privacy and security have become major concerns since an increasing number of transactions are made over an unsecured network. Thus there is a greater chance for private data to be misused. Further, insider attacks can result in loss of valuable data. Hence there arises a strong need for continual, non-intrusive, quick user re-authentication. Previously, a number of studies have been conducted on authentication using behavioral attributes. Currently, few successful re-authentication mechanisms are available since they use either the mouse or the keyboard for re-authentication and target particular applications. However, successful re-authentication is still dependent on a large number of factors such as user excitation level, fatigue and using just the keyboard or the mouse does not mitigate these factors successfully.

Both keyboard and mouse contain valuable, hard-to-duplicate information about the user's behavior. This can be used for analysis and identification of the current user. We propose an application independent system that uses this information for user re-authentication. This system will authenticate the user continually based on his/her behavioral attributes obtained from both the keyboard and mouse operations. This re-authentication system is simple, continual, non-intrusive and easily deployable. To utilize the mouse and keyboard information for re-authentication, we propose a novel heuristic that uses the percentage of mouse-to-keyboard interaction ratio. This heuristic allows us to extract suitable user-behavioral attributes. The extracted data is compared with an already trained database for user re-authentication.

The accuracy of the system is calculated by the number of correct identifications to total number of identifications. At present, the accuracy of the system is around 96% for application based user re-authentication and around 82% for application independent user re-authentication. We perform black box, white box testing and Spec# verification procedures that prove the robustness of the proposed system. On testing POCKET, a privacy protection software for children, it was found that the security of POCKET was inadequate at the user level. Our system enhances POCKET security at the user level and ensures that the child's privacy is protected.

To my family

Acknowledgement

First, I would like to thank Dr.Michael Hsiao, my advisor and committee chair, for his guidance and support in doing this thesis. He has encouraged me all through the course of study and given valuable suggestions in all facets of my research.

Second, I thank Dr.Chao Huang and Dr.Paul Plassmann for serving on my thesis defense committee. I have had the opportunity to take courses under them and get their encouragement and guidance in my endeavors.

Third, I thank my fellow PROACTIVErs for their valuable inputs, support and motivation through the period of study. I also wish to thank all my friends for encouraging me at every step of the research. I further thank all the participants of the study for providing valuable time and data over a continued period of time.

I also thank my family for being there at all times for me. I could not have done this work without their understanding and support.

Last but not the least, I thank God Almighty for giving me a chance to work in this exciting field and complete my work successfully.

Contents

List of Figures

List of Tables

List of Abbreviations

COPPA – Children's Online Privacy Protection Act, 1998

POCKET – Parental Online Consent for Kid's Electronic Transactions

P3P - Platform for Privacy Preferences Project

PIN – Personal Identification Number

SRS – Software Requirements Specification document

STS – Software Testing Specification document

UA – User Agent

BHO – Browser Helper Object

UPPF – User Policy Preference File

MPPF – Merchant Policy Preference File

IE – Internet Explorer

IQ – Interaction Quotient

Chapter 1

Introduction

1.1 Introduction

With the advent of internet, transactions occur over large computer networks and data about a person travels through huge clusters of computers. In the absence of proper security features, it becomes very easy for anyone to hack into the system and access private data. Such private data can be misused by fraudulent entities harming the user. Hence, security of private data is of paramount importance. Security breaches usually result in loss of valuable data and are not known till it is too late. Children are more vulnerable to such attacks since unapproved information can be easily obtained from them by unscrupulous merchant websites. In order to protect children from privacy vulnerabilities, the Children's Online Privacy Protection Act (COPPA) of 1998 specifies certain guidelines to be followed by parents and merchant sites. Specifically, it details the information that needs parental approval before being collected from a child less than 13 years of age [40].

POCKET is a software system that implements COPPA guidelines and comes up with a way to automate parental approval for information collected by the merchants [41]. It creates preference files for children specifying the data that can be collected by merchants from the child with parental consent. Some of the information such as the address of the child, phone number, social security number, etc may not be allowed to be collected from the child. Such information can be de-selected while creating the preference file. Once POCKET is configured with this information, it matches the preference of the client (in this case the child browsing the merchant website) to that of the server (the merchant website). Only if the preferences match, the site is allowed to be viewed by the child. If the preferences do not match, the site is blocked. The client's preferences and the merchant policy files have a format that is consistent with the P3P specifications [42]. Parents can also update the preferences periodically. POCKET requires authentication to use/modify it to ensure that the correct user uses the system.

Authentication refers to the process of confirming the identity of a user for the purpose of granting access to specific information or to do a particular task. Authentication can be done by a number of methods like:

- traditional authentication (involving passwords, personal identification numbers(PINs),etc)
- hardware authentication (involving smart cards, dongles,etc)
- physiological authentication (involving biometric features like iris scanning, fingerprint scanning,etc)
- behavioral authentication (involving some behavioral attributes of the user for identification)

Among these, behavioral authentication is particularly attractive as it is easy to deploy and cost-effective. In particular, keyboard and mouse authentication methods are effective for a number of reasons:

- *commonly used* – keyboard and mouse are the most commonly used input devices. As a result, there is no need for special equipment for user re-authentication. This helps in easy integration into the existing setup.
- *do not require extensive maintenance* – the keyboard and mouse are robust and require minimal maintenance compared to iris scanning, fingerprint scanning, etc
- *cannot be easily hacked* – the behavioral attributes of a person are unique to that person. It is practically impossible to exactly replicate a person's behavior. So, unlike traditional

methods like passwords, PINs, smartcards which can be hacked into if they are lost, behavioral systems are nearly hack-proof.

- *cannot be tampered with* – Hardware authentication methods like smart cards, dongles, etc. can be tampered with to allow an unauthorized user to pose as an authorized user. They can also be stolen from an authorized user. However, behavioral systems cannot be modified or tampered to allow unauthorized entry.
- *lost or stolen* – traditional software and hardware authentication methods require a passkey to verify a user. Anyone having this passkey is considered to be an authentic user. However, when these are lost or stolen from the rightful owner, they can be misused. But this problem is alleviated in behavioral systems.

The only drawback that these systems have is that they must be periodically retrained to reflect the minute behavioral changes of the user. However, with continued usage of the system, this drawback can be addressed by incorporating training as a part of the system. Once the user is authenticated, it is assumed that the user logged into the system ('*logged-in user*') is the only one using the system ('*current-user*') during the session. However, this might not always be the case. Unauthorized users can use the system while in session and the system would not know the difference between the '*logged-in user*' and the '*current-user*'. This can lead to security attacks by insiders. To prevent it, continual non-intrusive user re-authentication is necessary.

User re-authentication adds an extra layer of security to the system. The user's behavior is continually monitored during the session to flag "anomalous" behavior. User re-authentication ensures that the '*current-user*' is the '*logged-in user*'. One of the costliest security threats in computer usage is theft of information by insider attacks [34]. Some of the most common ways of insider attacks is through unauthorized access:

- using an authorized user's password/PIN/pass phrase (passes the initial authentication)
- using an open-and-unattended account (account opened by authorized user and unattended for some time)
- using a forgot-to-log-out account (account opened by authorized user but not logged out after completing the tasks)

These methods, combined with the right level of authorization, can potentially harm the entire system and make the whole network vulnerable to further attacks by disabling the security safeguards.

The main problems addressed in this thesis are:

- Is POCKET effective enough in protecting the privacy of kids? Does it perform in accordance with its requirements?
- Does POCKET need any other system to compliment its operation? What should be the characteristics and functions of such a system?
- How to ensure that the new enhancement to POCKET increases its effectiveness in protecting the privacy of kids online?

1.2 Contributions of the thesis

The first contribution of this thesis is to extensively test POCKET and check if it adheres to all its requirements and expectations. On testing and verifying POCKET, it is seen that the security of the system is adequate. However, it is also found that the security at the user level is not robust (i.e.) the child can be logged into the parent's account and use the internet and the system would not know it. This results in a breach in the privacy protection of the kid and defeats the purpose of POCKET software. This leads to the second question addressed by the thesis. In order to address this drawback of POCKET, a user re-authentication system is proposed. The proposed user re-authentication system should be simple, easy to use and cost effective in deploying it in homes. Also, it is enough if the system can identify between the members of the family.

The proposed system uses behavioral authentication to perform continual non-intrusive re-authentication of the user. Specifically, it uses the keyboard and mouse. They are the most commonly used input devices when using a computer. They are also continually used in every session and do not require any special input. All the operations that are normally done by the user can be subjected to analysis of their behavioral attributes. Further, behavioral attributes have various inherent advantages like cost-effectiveness, easy integration with the existing system, fast deployment, uniqueness of each user, etc. that are added to the system. Analysis of related previous work reveals two drawbacks of systems proposed till now: 1) they perform re-authentication for particular applications, 2) they use only mouse for re-authentication. Our system proposes a user re-authentication method that can be used for all applications and uses both keyboard and mouse information for analysis and identification.

This thesis proposes a new algorithm which performs identification and re-authentication. It consists of two parts: the first part of the algorithm involves the usage of statistical measures,

neural networks and k-Nearest Neighbor algorithm to identify the current user. The second part of algorithm authenticates the user in the system. This algorithm obtains keyboard and mouse data from the user continually, performs identification of the user and authenticates her based on log-in information. It involves a basic training setup, training sessions and testing sessions. This algorithm is also tested and verified to ensure that it satisfies the requirements of user re-authentication system.

1.3 Organization of the thesis

The rest of the thesis is organized as follows:

Chapter 2 – gives a brief overview of the technical details and implementation of POCKET, authentication, re-authentication, behavioral systems and the various heuristics used in the user re-authentication system.

Chapter 3 – explains the software testing procedure of POCKET and tests the software for its robustness and security.

Chapter 4 – explains the design and development of a user re-authentication system using behavioral attributes. Since keyboard and mouse are the most commonly used input devices, they are chosen for their ease of user input, ease of deployment and cost effectiveness of the system. Statistical analysis, feed forward neural network with back propagation and k-Nearest neighbor algorithms with IQ analysis proved to be effective in identifying the user from other users.

Chapter 5 – explains the testing, validation and verification cycle of the proposed user re-authentication system. These procedures are performed to ensure the robustness of the system and to check the system for drawbacks, if any.

Chapter 6 – gives a brief description of related work in the field of authentication and re-authentication using mouse and keyboard attributes. It shows the necessity of this research work.

Chapter 7 – concludes the thesis, gives an overview of the results and discusses future work that can enhance this solution.

Chapter 2

Background

This chapter details the concepts and the theories behind the analysis used in this research work such as POCKET, authentication, re-authentication, behavioral attributes, statistical heuristics, artificial neural networks, k-Nearest neighbor algorithms, Spec#, etc. Each concept has been briefly explained with examples wherever applicable. A discussion on the relevant previous works and their implications on our approach has been included in Chapter 6.

2.1 POCKET

Parental Online Consent for Kid's Electronic Transactions system (POCKET) is a software system that can obtain parental consent for information transactions and protect children's online privacy [41]. It consists of preference files that are created by parents for each child. Preference files specify the data approved by the parent and that can be collected by the merchant website. Only the information that is safe for the child's privacy is selected by the parent. All other information is deselected and cannot be sent to the merchant website. When a website is requested, the preference files of the client (child, in this case) and merchant (vendor) are

matched. If the merchant requests equal or lesser information from the client, the site is allowed to be displayed on the browser. If the merchant requests more information than approved, the site is blocked. Parents can also update the preferences periodically. This feature ensures that the preference files in POCKET can be modified according to the needs of the child.

The P3P framework serves as the base for POCKET while creating preference files for both client and merchant. It includes a few extensions like

- Using a trusted third party (TTP) for interaction between the client and merchant.
- Modifying the merchant preference file to include POCKET compatible data elements, their usage and handling, and
- Automation in personal information exchange between the client and merchant [41].

Using the POCKET user agent, the client is identified as a child and the information approved by the parents can be collected by the merchants automatically. All forms and input pages are eliminated. In its place, if the form contains approved information, the information package is created and automatically sent to the merchant site. Some of the advantages to this approach are:

- The merchant websites cannot collect information that is not specified in the user preference file. It prevents the child from unknowingly giving out optional yet privacy-threatening information.
- The parents can be assured that only information approved by them is given out to the merchants eliminating security risk.
- Unsolicited attacks on the information are avoided by using secure transmission. Further, the merchant websites are modified by changing their P3P policy files to include POCKET-specific policy.

POCKET consists of a user agent (UA) and a browser helper object (BHO). Once setup, the information is exchanged between the client and merchant using a protocol. There are three stages of operation – registration, setup and transaction. The UA operates in both the parent mode (where the preferences are set and modified) and the child mode (where normal activity is done). In the parent mode, the software gives a questionnaire detailing the different fields such as name, address, phone number, etc. that identify the child. The parent can choose the fields and the corresponding answers. These responses are converted to a user preference file (UPPF) and stored in the client computer. In the child mode, the UA installs the BHO and loads the preference file. The BHO, in tandem with the browser, intercepts messages from and to the internet. When a merchant website containing forms is sent to the client (MPPF), the BHO fills in the approved information and sends it back to the merchant. The child does not get to see any forms at all

thereby preventing unsolicited entry of private data by the kid. The preferences can be changed in the parent mode. The UA enters the parent mode after password authentication.

2.2 Biometric Authentication

In Chapter 1, we saw a brief description of authentication, re-authentication and their types. Here, we explain further about biometric authentication. Biometric authentication refers to usage of a person's physical characteristics or attributes to identify and differentiate her from others [43].Biometric systems are preferred because almost all artificial features such as smart cards, passwords, etc. can potentially be forged, stolen or lost. For a secure and 'natural' system, biometric authentication methods are preferred. They require reliable raw data and simple yet powerful extraction methods to perform identification. There are two branches of biometric authentication:

1) Physiological systems – these systems verify a person's identity by their physical characteristics such as fingerprints, iris patterns, facial features, palm aberrations, DNA, etc. Although these attributes are practically unalterable, they are hard to deploy and costly.
2) Behavioral systems – these systems verify a person's identity based on their behavioral patterns for chosen attributes. Some of them include voice quality, keyboard typing behavior, hand movements, etc. Although the accuracy of these systems are dependent on a lot of factors such as stress, excitement level of user, devices used for data collection, etc. they are cheap, can be easily integrated and are effective for a small group of users.

Based on the application, the systems and attributes are chosen. Face recognition can be used in large surveillance applications that require quick and non-intrusive authentication. Speech recognition is preferred in cases where the users interact with systems through predefined sentences. Keyboard and mouse identification systems are used in scenarios where the user continually interacts with the system through keyboard and mouse. A careful analysis of the application is necessary for successful authentication.

Though these systems can virtually eliminate computer fraud, the present systems are still not robust enough for day-to-day use. Continued research in biometric authentication has lead to path breaking results. Specifically, a lot of work has been conducted on face recognition and speech recognition systems. However, these systems require special apparatus for setup and analysis.

They also have problems such as cost, difficulty of deployment, high maintenance, etc. So, recent research concentrates on effective keyboard and mouse authentication systems. They have a number of inherent advantages as discussed in Chapter 1. Specifically, research work has been geared towards inventing systems with a high accuracy and minimal intrusiveness. An ideal system has 100% sensitivity (identifying user A as A), 100% specificity (identifying user B as **not** A), 0% false acceptance rate (wrongly identifying B as A) and 0% false rejection rate (wrongly identifying A as **not** A). From related literature survey in Chapter 6, we see that current systems are far behind the ideal system. Hence, more research in this area is necessary.

2.3 C# statements

Since the system is coded in C#, we briefly explain some of the C# statements used for safe file handling:

- 'try' – it specifies the start of the block of code that the compiler is expected to execute normally. If there is any error, an exception is thrown. This exception is caught by the corresponding 'catch' block. Usually, the 'try' block contains code that interacts with the OS and has chances of throwing exceptions.
- 'catch' – it denotes the block of code which is executed if and only if an exception is thrown while executing the code within the 'try' block. Usually, it contains exception handling code. Sometimes, multiple 'catch' statements are used to handle different types of exceptions. This statement does not exist without the 'try' statement.
- 'finally' – this keyword indicates the block of code which follows the 'try' and 'catch' statements and is always executed irrespective of an exception being thrown. However, adding a 'finally' block after a 'try' and 'catch' pair is optional. If a 'finally' block is not specified, the compiler inserts a blank 'finally' block.

  The example given below shows the usage of 'try', 'catch' and 'finally' block:

```
try{
    StreamReader sr = new StreamReader("hello.txt");
     //do file reading operations
}
catch(Exception e){
  //display error message e
}
finally{
```

Safe_dispose(sr);
}

In this example, if the file handle 'sr' has been assigned values successfully, then the 'try' block is executed, followed by the 'finally' block. Otherwise, the exception is caught at the 'catch' block, an error message is displayed and the 'finally' block is executed.

- 'using' – it denotes the start of a block of code that substitutes 'try' and 'finally' blocks. Internally, the 'using' statement maps to 'try' and 'finally' blocks. It is mainly used in cases where a 'catch' statement is unnecessary. This statement is used to simplify the code. From the preceding example, if the catch statement is not necessary, the remaining code can be replaced by the following code snippet using the 'using' block:

  using(StreamReader sr = new StreamReader("hello.txt")){
  //Do file reading operation here
  }

  In this example, if the file handle 'sr' is not assigned, the 'using' statement ensures that the system safely disposes the file handle and does not throw an exception.

2.4 Heuristics used

The system uses three different heuristics for user-authentication. Statistical methods such as correlation, average difference and algorithms such as neural networks with back propagation method, k-nearest neighbor form the basis of these heuristics for user prediction from the given data.

2.4.1 Statistical analysis

Correlation of two random variables is defined as the measure of similarity and possibility of prediction between the numbers. Simply, two random variables are correlated if knowing something about one variable gives some attribute of the other variable [39]. The random variables can have positive or negative correlation. The main aim of correlation in this system is to capture the level of similarity of the test data with the trained data. Pearson's product-moment correlation coefficient is used in this analysis. The formula for Pearson's product-moment correlation coefficient can be expressed as:

$$\rho = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{X_i - \mu_X}{\sigma_X}\right)\left(\frac{Y_i - \mu_Y}{\sigma_Y}\right)$$

where μ_X and σ_X are the mean and standard deviation of the data set. Mean refers to the average value of the data set while standard deviation gives the average deviation of any sample from the average value. It measures how widely spread the values in a data set is compared to the training data. If many data points are close to the mean, the standard deviation is small; if many data points are far from the mean, then the standard deviation is large. If all data values are equal, then the standard deviation is zero. Pearson product-moment correlation has been chosen because of its applicability to the problem.

Obtain *the input values for all the attributes*
Do
    For *every user u in the database*
        P = Pearson_Correlation(u,p,no_attributes);
    End
  P_{user} *= Max(P);*
Until *end of training session*
Network ready for testing

2.4.2. Artificial neural networks

Artificial neural networks refer to the set of algorithms modeled based on biological neural networks and uses similar strategies to achieve the results. The algorithm creates a set of neurons that are interconnected to each other and produce results based on these interconnections. The advantage of this system is that they can easily adapt to the changes in the input and output and vary their weights to produce good results. Given sufficient data, they can model any non-linear system. They are also very good at pattern recognition.

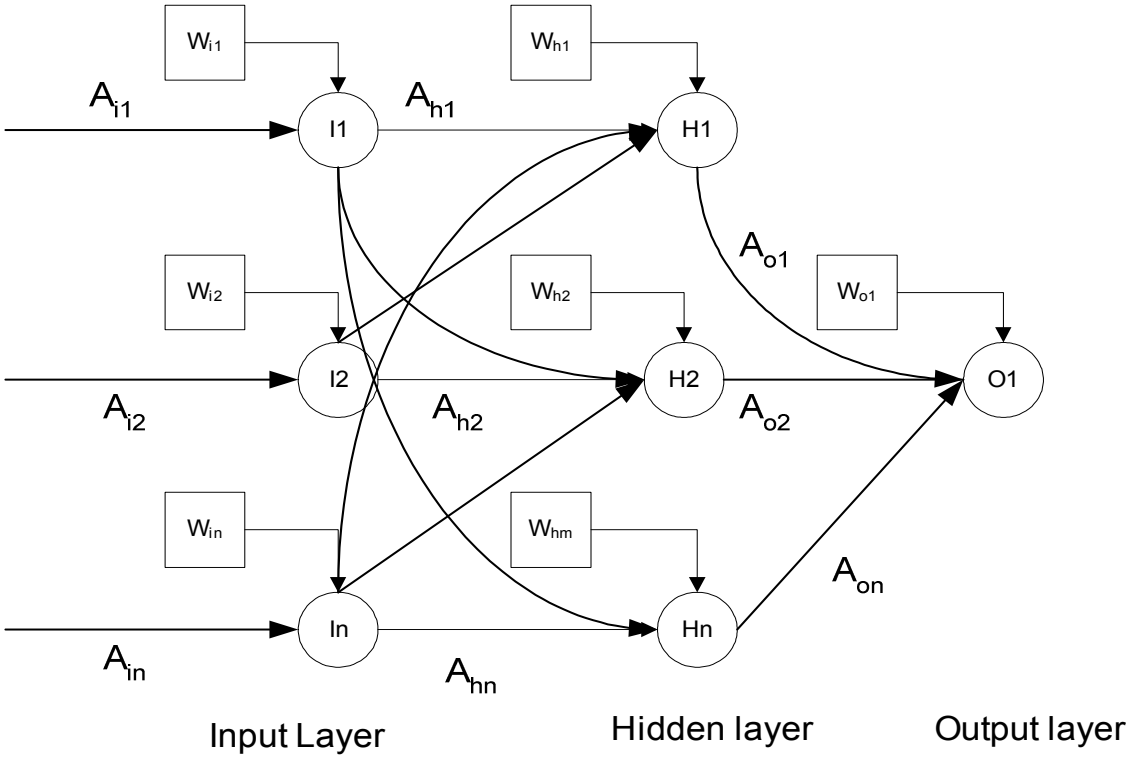

Figure 1: A simple feed forward network

In Figure 1, the feed forward network is shown. The inputs are fed into the neurons in the input layer (I_j). Their weights are labeled W_{ij} and are initially assigned random values. Their activation values are denoted by A_{ij}. They specify the probability of activation of the corresponding neuron. The output of the input neuron layer is fed to the neurons in hidden layer (H_j). Their weights are labeled W_{hj} and are initially assigned random values as well. Their activation values are denoted by A_{hj}. The results from the hidden layer neurons act as inputs to the neuron in output layer. Since we need only one output from the system (if it is the user (denoted by a 1) or not (denoted by a 0)), only one output neuron is shown in the figure. The weights of the output neuron are denoted by W_{oj}.

The back propagation algorithm [44] refers to the propagation of the calculated error from the back of the system (i.e., from the output layer) to the front of the system (i.e., from the input layer). Suppose the calculated output is O_1 while the expected output is O_{C1}. So, the error difference is $e_1 = O_{C1} - O_1$. This difference is fed back to the hidden layer where the weights are varied. This change in weight from W_{hj} to W_{hj1} is given by:

$dw_{hij} = r^* a_i * e_j$,

where r is the learning rate and a_i represents the activation of W_{hj}.

If the set of input patterns form a linearly independent set then arbitrary associations can be learned using the delta rule [45]. Then, the differences are fed back to the input layer whose individual weights are modified in accordance with 'Delta rule'. The 'Delta rule' is used in this system as it has only one hidden layer. The Delta rule is inefficient for any feed forward system having two or more hidden layers.

Since the system tries to find patterns based on the trained data, it follows supervised learning i.e. the system starts with a random set of weights for its neurons, obtains the inputs and calculates the outputs based on the weights and the inputs. The calculated output is compared with the expected output and the error is fed back to the network using back propagation algorithm. As a result, the weights of the neurons are suitably modified to make the calculated output to more closely match the expected output. This procedure is repeated for the given data set. This is known as network training phase. At the end of the training phase, the system consists of optimized weights for the neurons that can more closely predict the actual function's outputs.

A feed forward neural network is used in this system. It is a multiple layer perceptron network with one input layer, one hidden layer and one output layer. At every node, the sum of products of inputs and weights is calculated; if the value is above a threshold, the neuron is fired, else the neuron is deactivated. At the output, the difference between the sample output and calculated output are found and this difference is used to vary the weights of the node. This is known as Delta rule and is implemented through back-propagation. Back propagation helps the system operate in a closed loop feedback system and achieves satisfactory results (i.e.) adequately trained system in a short time. However, the accuracy of the system depends on various factors such as the stopping criterion, the number and accuracy of samples, number of neurons in each layer, etc The algorithm of a simple feed forward neural network with back propagation is given below:

Initialize the network with neurons, neuron connections and random weights
Do
For every sample t in training set
Z = Network(t); feed forward network.
Zt = Sample output from the data
Find (Zt – Z), Δ_H *and* Δ_I
Using Δ_H *and* Δ_I *, modify the weights of each node in the network.*
Until system classifies correctly or above stopping threshold

Network ready for testing
Where

Δ_H → *the difference in sample output and calculated output is fed to the nodes in the hidden layer*

Δ_I → *the difference in the hidden layer results are fed to the nodes in the input layer*

2.4.3 K-Nearest Neighbor algorithm (k-NN)

k-Nearest Neighbor is a simple algorithm which classifies samples based on their proximity to the training samples. It is a typical example of instance-based learning or lazy learning where all the samples are obtained and the computation is done in the end unlike the neural networks where calculation is done after each sample is fed to the network in the training phase. Here, the test sample is compared with all the training samples and k-training samples closest to it are chosen. Their classification is studied and the test sample is assigned the class of the majority of the k-neighbors. As the number of k increases, the accuracy and complexity of the system increases. Usually k is small; it is chosen such that it is odd and not a divisor of total number of samples. In training phase, the samples are obtained and their classification is stored. In testing phase, the test sample's distance from all training samples are found and k closest samples are chosen. The majority of class among the k samples is assigned to the test sample.

Obtain *the input values for all the attributes*
Do
  For *every user u in the database*
    $D_{euclidean}$ = *Euclidean_distance(u,p) where p is the test sample;.*
    $D_{chebyshev}$ = *Chebyshev_distance(u,p)*
    $D_{manhattan}$ = *Manhattan_distance(u,p);*
  End
  E_{user} = *k-Min*($D_{euclidean}$*);*
  C_{user} = *k-Min*($D_{chebyshev}$*);*
  M_{user} = *k-Min*($D_{manhattan}$*) ; where k = Min(3,no_of_users);*
  User = *Max*(E_{user},C_{user},M_{user}*);*
  return(User) ;
Until *end of training session*
Network ready for testing

Consider a simple example of five vectors of red set and five vectors of blue set in Figure 2. Here, we see that the point A is closest to the red cluster and is hence classified as such. Point B is closest to the blue cluster and is classified as blue. However, point C has three points from the red cluster and two points from the blue cluster. Since number of red points close to C > number of blue points close to C, C is classified as red as shown in Figure 3.

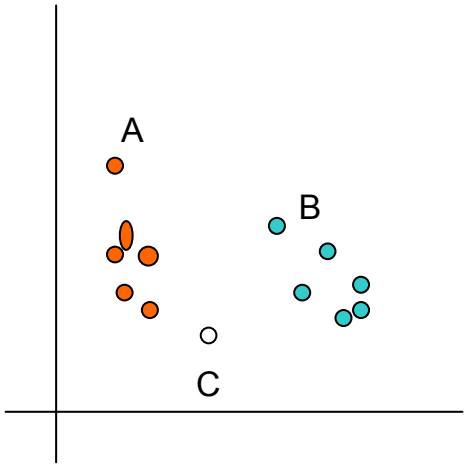

Figure 2: Example showing k-NN algorithm usage

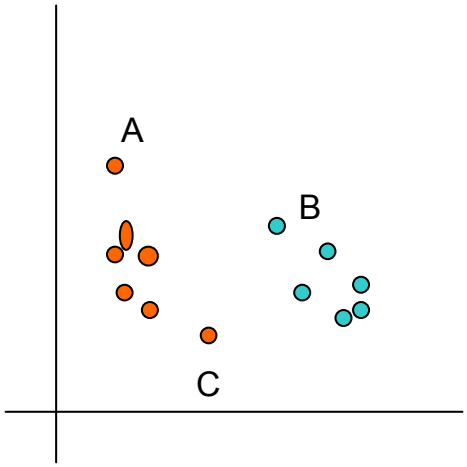

Figure 3: Example with sample C sorted using k-NN algorithm

2.5 Software testing

Software testing and verification procedures form an integral part in a system development cycle. The software requirements specification (SRS) document specifies the functionality, external interfaces, performance, attributes and design constraints of the system. It is the agreed-upon document between the customer and the developer. Once the system has been designed, it is important to test the system and verify that the system performs the expected operations correctly.

Testing ensures that the system "*does what it is supposed to do and does not do what it is not supposed to do*". Testing is an important phase of software development in that it is aimed at evaluating an attribute or capability of a program or system and determining that it meets its

required results [35]. Software testing concentrates on statement coverage and path coverage metrics.

There are many types of software testing: black box testing, white box testing, unit testing, incremental integration testing, functional testing, system testing, beta testing, etc. In this thesis, black box testing and white box testing of POCKET software and the user re-authentication system are detailed. A brief definition of different types of software testing is given below:

- *Black box testing* – refers to the testcases written without knowledge of the internal structure of the system. This is a random testing method which covers some of the most commonly used paths leaving the tester to concentrate on the hard-to-reach paths in other forms of testing. Black box testing also ensures that the system works with a high degree of confidence. The tests are based on user inputs and the corresponding outputs. The paths are counted as follows:
    - 'if-else if-else' statements are considered to be different paths based on the number of conditions in the block.
    - 'while' block is considered to have two paths – condition in 'while' executes to 'true', condition in 'while' executes to 'false'.
    - 'try catch' statement and 'using' statement have two paths – 'try' part is executed without exceptions, 'catch' block is executed due to some exception that is thrown in 'try' block and caught.
    - 'for' loops, 'do while' statements, etc have only one path.
- *White box testing* – refers to the testcases written taking into account the internal structure of the system. This type of testing targets specific cases of system operation. Most of the time, these specific cases are hard-to-reach corner cases that do not occur frequently during system usage. It is also known as Glass box Testing. Internal working of the code should be known for this type of testing. Tests are based on coverage of code statements, branches, paths, conditions. White box testing, by itself, is tedious and requires a lot of time, resources and effort. In combination with black box testing, it gives good results in a shorter time. A complete test suite of white box testing and black box testing will give 100% path coverage.

2.6 Software Verification

Software verification refers to the software discipline that makes sure that software satisfies all the requirements given. It contains both dynamic and static verification. Dynamic verification refers to software testing and has been discussed in detail in Section 2.5. Static verification refers to the process of applying physical techniques to ensure that the requirements are met by the system. It can be split into formal verification and metrics verification. Metrics verification refers to the measurement of some property of the code or its specification. Some of the most popular software metrics are code coverage, path coverage, number of lines of code, bugs per line of code, etc.

Formal verification is the method of proving the correctness of the algorithms of a system in context of certain properties or specifications. It can be used in various algorithms that use logic such as digital circuits, combinational circuits, source code, etc. The abstract math model of the system is created and its properties are proved formally. Formal verification can either be done by model checking or logical inference. Model checking is an exhaustive check of the system's states and their transitions. A number of techniques can be used to reduce the time and resources required for the exhaustive exploration of the states in a system. In logical inference, mathematical reasoning of the abstract math model is done using theorem proving software such as HOL theorem prover, ACL2 theorem prover, etc. Properties of a system that need to be verified are expressed in temporal logic.

In this thesis, in order to verify the C# code, a specification system called Spec# from Microsoft research is used. It consists of Spec# programming language, Spec# compiler and the Spec# static program verifier [38]. Spec# is an extension of C#. It includes non-null types and checked exceptions and the method properties are type-casted using preconditions, post conditions and object invariants. Spec# compiler is integrated into C# environment. The compiler statically enforces non-null types, emits run-time checks for method contracts and invariants, and records the contracts as metadata for consumption by downstream tools. Given a Spec# program, the program verifier (called Boogie) generates the logic verification conditions. Boogie uses Automatic Theorem Prover to analyze verification conditions and prove or disprove their correctness. It uses various tools such as dynamic checking, automatic static verification, etc. for proof of verification.

[38] states that "Spec# is uniquely suited for object oriented programming in that it can specify, verify and maintain invariants. Further, Boogie can handle callbacks, threads, aggregate objects and inter-object relationships and it supports both object and static class invariants. It enables sound, modular reasoning." These features suit the C# language in which the code is written. Hence Spec# is chosen for verification of the code. A simple example of C# code and its corresponding Spec# code with keywords of properties are given below;

C# code:

```
void activate(Neuron n)
{
      for (int j = 0; j < 62; j++)
      {
         n.Excitement += n.Preons[j].Pre.Output * n.Preons[j].Weight;
      }
      n.Output = 1 / (1 + ((-lambda * (n.Excitement + n.Bias))));
}
```

Spec# code:

```
void activate(Neuron n)
        requires n!=null;
        ensures n.Output!=0;
        {
        assume n.Preons!=null;

            for (int j = 0; j < 62; j++)
            invariant j>=0;
            invariant j<=62;
            {
                n.Excitement      +=      n.Preons[j].Pre.Output      *
n.Preons[j].Weight;
            }
            n.Output = 1 / (1 + ((-lambda * (n.Excitement + n.Bias))));
        }
```

The above code consists of four properties that need to be verified statically. The keywords and the properties are listed below:

- 'requires x !=null' – here, the variable x is verified for non-null condition.
- 'ensures' – this keyword denotes a post-condition (i.e.) a condition that has to hold after control quits the function.
- 'invariant' – this keyword denotes that the specified condition must hold true for every run of the loop – in this example, there are two 'invariant' properties that need to be verified.

Chapter 3

Software Testing of POCKET

In this chapter, software testing methodologies used to test POCKET are detailed. In Chapter 2, a brief description of POCKET was given. This chapter concentrates on the testing and validation of POCKET for varied responses. It is subdivided into the 'Motivation' section which describes the rationale behind this work, 'Approach' section which details the architecture and methodology of testing framework and 'Experimental results' which analyze the results followed by 'Summary' section which concludes the chapter with the observations of testing procedure.

3.1 Motivation

Software testing is an integral part of any software development cycle. The main aim of software testing is to ensure that the system works in the specified manner when a certain set of inputs are given. Further, the system must not give unexpected results thereby undermining its dependability. POCKET will be used in homes by parents with the trust that the system can allow/block websites based on the given preferences. In order to make sure that this is true, the system should be tested extensively.

3.2 Approach

From the SRS document, it was observed that, theoretically, the total number of paths that can be taken by POCKET from input level to output level is 9.0048E+43. However, it is also seen that there exist certain parts of code that are executed in every path. Testing the same code repeatedly with similar inputs and external environment variables leads to unnecessary waste of resources. It is advantageous and simpler to test the paths in each function and make sure that the paths conform to the expected behavior. For, every path in the system has to pass through one of the possible paths in any chosen function. Testing the paths of each function with proper inputs ensures that all the combinations of these paths (i.e., the paths in the system) are tested with a high degree of confidence. This testing method, in turn, saves time, effort and resources. Applying this concept reduces the total number of paths to 283918515492. The selection of paths was detailed in Chapter 2.

The testing procedure consists of two stages: black box testing and white box testing. Since POCKET has a user interface part (UA) and a machine executable part (BHO), it is necessary to test both parts. Black box testing involves the selection of different inputs at the user interface. Since these inputs are given by the user, the corresponding paths taken by the software are commonly used. Blackbox testing allows the tester to concentrate on hard-to-reach paths in other forms of testing. Black box testing also helps to assure a high degree of reliability on the operation of the entire system.

3.2.1 Black box testing

The black box testing of POCKET begins with testing the installer and moves on to testing the actual software. In the first step, the installer installs the POCKET User Agent. After installation, the UA section asks for a login name and password. It does two checks, password length being 6 or more characters and password consistency check. Then, the UA shows the preference selection form where all combinations of preferences are possible. Each combination is checked. However, some options are mutually exclusive and cannot be selected together. For example, if the parent elects to give out only the last name of the child and chooses “Only Last Name” option, then the “Only First Name” and “Full Name” options are deselected automatically. This ensures that there is no confusion in selecting conflicting preferences. We also test if the tooltips function as required. The next screen shows the preference information completion form. Here, the selected

preferences have blanks to be filled by the parents. Some consistency checks for Birth Date, Exact age and Age range are done to ensure that the given information corroborates with one another. When an information field is left blank, the system makes sure that it is completed. This procedure is repeated for all combinations possible. After completing the preference selection, the UPPF file is created. Then, in order to install the BHO and start POCKET, the software closes all the open IE windows. Once the BHO is installed and UPPF stored successfully, the UA is enabled and shown as a dialog box on the desktop. It can be moved to the system tray by minimizing it.

Some of the features of POCKET UA are to enable/disable the browser privacy protection, update UPPF file, change the password and close POCKET. Each feature is tested extensively. For example, consider the password change feature. When this feature is selected, a password change dialog box appears (Figure 4). Here, thirteen different paths are possible for black box testing. The paths are listed below:

1) The login name and password do not match.
2) The login name does not exist.
3) The login name is blank.
4) The password is wrong.
5) The password field is blank.
6) The new password is less than 6 characters.
7) The new password is blank.
8) The retyped password field is less than 6 characters.
9) The retyped password field is not consistent with the new password.
10) The retyped password field is blank.
11) Successful change of password
12) Unsuccessful change of password (due to internal error)
13) 'Cancel' button is pressed and control returns to POCKET application window.

Each path is tested with appropriate inputs and the corresponding confirmation/error messages are verified. This analysis is done for every feature and the corresponding inputs are used. POCKET can be minimized to the system tray. So, the system tray options are also tested. For example, the POCKET icon in the system tray contains 'Restore' option. This option is verified to see if the POCKET window gets restored to normal size. Similarly, other options for POCKET icon in the system tray are tested.

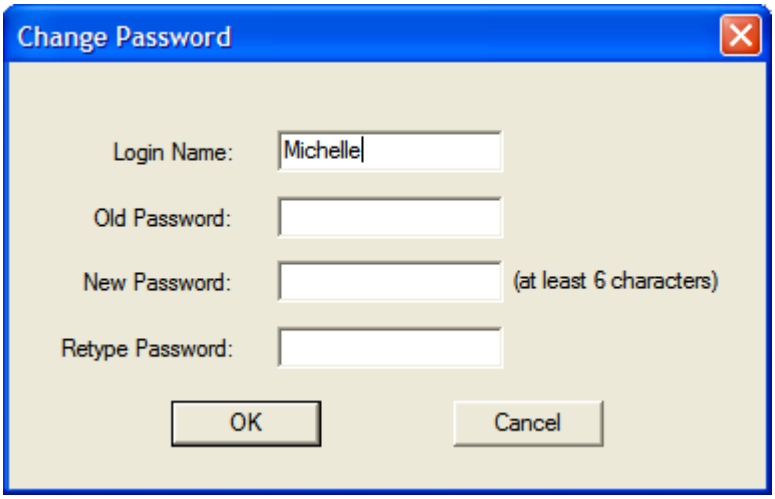

Figure 4: Password change form

Finally, the About Menu and its dialog box results are tested. A short walkthrough of black box testing done for POCKET is detailed in Appendix A.

Repeated runs of the software result in a large number of paths being covered. In our system, the total number of paths covered by black box testing is 283545222346. Total number of paths is obtained by incrementing a counter every time a path is covered. This is done by modifying the code suitably. Black box test results translate to 99.869% path coverage. The results for path coverage are shown in Table 1 listing the number of paths in each class and process. For example, AccountCreationDlg.cpp contains 8 paths of execution. All of them are covered by black box testing. In GraphHandler.cpp, there are 110 paths out of which 108 paths are covered by black box testing and 2 of them are not covered making the coverage of this class to be 98.18%

Table 1 : Path count of Black box testing of POCKET software

Process and Class name	Total number of paths	Number of paths executed	Number of paths yet to be covered
POCKETUserAgent (UA)	**283918515158**	**283545222022**	**373293126**
AccountCreationDlg.cpp	8	8	0
ChangeAccountDlg.cpp	9	9	0
CloseIEDlg.cpp	10	10	0
GraphHandler.cpp	110	108	2
LoginDlg.cpp	7	7	0
PocketUserAgent.cpp	63	36	27
PocketUserAgentDlg.cpp	102	82	20
PPFContDlg.cpp	283448752619	283109013994	339738625
PPFDlg.cpp	40	35	5
PPFile.cpp	469762063	436207625	33554438

PPFUpdateDlg.cpp	11	11	0
SystemTray.cpp	76	62	14
UsrAccount.cpp	15	12	3
XercesString.cpp	19	19	0
GraphMain.cpp	6	4	2
BHO	**334**	**324**	**10**
BHOPlugin.cpp	8	7	1
EyeOnIE.cpp	195	188	7
GraphHandler.cpp	112	110	2
XercesString.cpp	19	19	0

3.2.2 White box testing

When we analyze the remaining 373293146 paths, we see that they are either constrained by some inputs to the function or are untestable. For example, in function

STDMETHODIMP CEyeOnIE::SetSite(IUnknown *pUnkSite)

The IE browser is continuously monitored for page reloads. In the code, we see that one of the paths is taken when pUnkSite is null. However, under normal operation of the system, pUnkSite is never null. The 'if' statement is used as a precaution in this function. So, in order to test this path, a unit test case with null input value for pUnkSite should be used. Such hard-to-reach paths are targeted with specific testcases in white box testing stage.

Every test case sets the external environment and provides the specified set of input combinations to the system. The output from the system is caught by test environment and compared with expected output. If the outputs are same, then test is passed. If not, a bug is observed at the path and is analyzed. White box testing increases path coverage to greater than 99.99%. This is shown in Table 2. Consider class PocketUserAgent.cpp with 63 paths of execution. Black box testing (36 paths) followed by white box testing (27 paths) increases the path coverage to 100%.

For example, consider the function below

```
int CEyeOnIE::getVendorpolicyPreference (XercesString name)
{
      for (int i=0; i < vendorList.size(); i++) {
      if (!XMLString::compareString(vendorList[i].policyName,
      name)) {
                  return vendorList[i].policyPreference;
            }
      }
      //control never comes here during normal operation
      return 0;
}
```

Here, the function compares the name of the policy (such as name, address, age, etc.) with the policy given by the vendor (from MPPF file). In black box testing, the 'if' statement is executed to 'true' all the time. So, the return value is never '0'. In white box testing, in order to test the 'else' path, the name input is assigned a value that is not a policy name. As a result, the returned value is '0'. This is compared with the expected value and found to be correct. So, this path is covered by the said test case.

Table 2: Path count of White box testing of POCKET software

Process and function name	Total number of paths	Number of paths already tested	Number of paths uniquely covered by unit testing	Number of paths yet to be covered
POCKETUserAgent (UA)	**283918515158**	**283545222022**	**373293116**	**10**
AccountCreationDlg.cpp	8	8	0	0
ChangeAccountDlg.cpp	9	9	0	0
CloseIEDlg.cpp	10	10	0	0
GraphHandler.cpp	110	108	2	0
LoginDlg.cpp	7	7	0	0
PocketUserAgent.cpp	63	36	27	0
PocketUserAgentDlg.cpp	102	82	10	10
PPFContDlg.cpp	283448752619	283109013994	339738625	0
PPFDlg.cpp	40	35	5	0
PPFile.cpp	469762063	436207625	33554438	0
PPFUpdateDlg.cpp	11	11	0	0
SystemTray.cpp	76	62	14	0
UsrAccount.cpp	15	12	3	0
XercesString.cpp	19	19	0	0
GraphMain.cpp	6	4	2	0
BrowserHelperObject(BHO)	**334**	**324**	**10**	**0**
BHOPlugin.cpp	8	7	1	0
EyeOnIE.cpp	195	188	7	0
GraphHandler.cpp	112	110	2	0
XercesString.cpp	19	19	0	0

From software testing of POCKET, we observe the following results:

- Since POCKET is highly user interactive, many paths are tested by black box testing.
- Hard-to-test paths are covered by white box testing.
- 10 paths were found to be untestable because they were the result of two 'if' statements. The condition of one 'if' statement was the 'else' condition of another 'if' statement making them practically impossible to be in the same path, though theoretically possible.
- 17 paths were found to be redundant due to the coding style and fixed in code design.

- A bug was also found during black box testing and has been fixed.

Integration testing (function testing) was not done for POCKET because of the huge number of paths possible (9.0048E+43) and lack of appropriate resources. Since 100% statement coverage and 100% path coverage are considered adequate enough to deem the system tested, POCKET is considered to be adequately tested for our purpose.

3.3 Summary

In this chapter, we have presented black box testing and white box testing methodologies for POCKET. On analysis of the testing results, we see that POCKET is extensively tested. The security features of POCKET are adequate during data transaction and hence the goal of the software (i.e.) privacy protection of the child is attainable. However, the security features of POCKET are based on the assumption that the user security is inherent. That is, the child only operates the system with the child's login id. But, in real world, this may not be the case. The child may use the system when the parent/guardian is logged in but is temporarily away from the computer. POCKET is usually disabled in parent's account. This, in turn, exposes the child to internet without any protective software. So, the system should be modified to include an identification system that automatically 'senses' the user operating the system. If the system identifies the '*current-user*' as a child, even if the '*logged-in user*' is a parent/guardian, POCKET should be enabled automatically or the user should be logged out of the account for security reasons. Such a system would be complementary to the objective of POCKET and would make POCKET more robust and secure.

Chapter 4

Design of User Authentication and Re-authentication System

This chapter details the design methodologies and development process of behavioral attribute based user authentication and re-authentication system. It is subdivided into four sections. They consist of the 'Motivation' which describes the rationale behind this work, 'Approach' which details the methodology and heuristics used, 'Experimental results' which analyze the results obtained followed by the 'Summary' which gives a brief zest of the chapter based on observations and suggests future enhancements.

4.1 Motivation

In Chapter 3, we described software testing of POCKET. We concluded that the security features of POCKET were adequate at all levels except one. POCKET did not have any mechanism to identify if the '*current-user'* is the same as '*logged-in user*'. This calls for a requirement for user authentication and re-authentication system that works in tandem with POCKET for data security at input level.

In POCKET, user authentication system will add a layer of authentication on top of password so that even if the child knows the password, she cannot change the preference file. The system will require the behavioral attributes of the parent to actually make changes to the preference file, thereby increasing the security of the system.

User re-authentication systems target the scenario where the child uses the computer for a period when the parent/guardian is logged in. The system should be able to detect the change in behavior of '*current-user*' and appropriately allow/block requests. This will ensure that the privacy of the child is not compromised. Some of the attributes of the proposed system are:

- ***Continual:*** User re-authentication is invoked at regular intervals to make sure that the '*current user*' is the same as the '*logged-in user*'. It identifies anomalies in user behavior immediately. This ensures the prevention of passive attacks on the system that would otherwise render it useless.
- ***Non-intrusive:*** This property improves user interaction experience. Prompting for password or PIN periodically would be intrusive and result in reduced user experience. Using other intrusive authentication mechanisms repeatedly would make the system unviable from the user's standpoint. The effort, time and cost of using such systems may be too high compared to the security advantages provided by it. So, the system must be non-intrusive and must perform re-authentication while providing a seamless experience.
- ***Behavioral:*** The keyboard and mouse are the most commonly used input devices in a computer. They are continually used in every session and do not require special input for analysis. All the operations that are normally done by the user can be used to extract behavioral info. This enhances user experience. Further, behavioral attributes have various inherent advantages such as cost-effectiveness, easy integration with existing system, fast deployment, uniqueness of each user, etc. So, the behavioral attributes from the keyboard and mouse are used to distinguish between users in this system.

4.2 Approach

In this chapter, we propose a system that uses three heuristics. These heuristics are based on '*statistical measures*', '*neural networks*' and '*k-nearest neighbor*' algorithm. The result from each of the heuristics is analyzed to converge on the final decision.

User authentication system is a simple authorization software. It gets the username and a sentence from the user and analyzes the results. The results are fed to the environmental system (described below) and the result (authenticated/not authenticated) is displayed to the user. If authenticated, the system allows the user to perform normal tasks. If not authenticated, the system would redirect the user to traditional log-in page.

User re-authentication system consists of two parts, the outer environmental system and the internal analysis engine (Figure 5). The environmental system collects data, performs pre-analysis calculations and sends it to the analysis engine. The analysis engine runs the obtained data across three heuristics and the database previously collected and stored. It returns the most probable user match from the database. The outer environment system compares the result with '*logged-in'* user and determines if the user behavior is anomalous or not. If so, the behavior is flagged. Else, re-authentication process is repeated. Once the number of anomalies crosses a preset threshold, the '*current-user*' is logged out and is asked to log in again. If authenticated, the system allows the user to resume normal operation and continues re-authentication procedures.

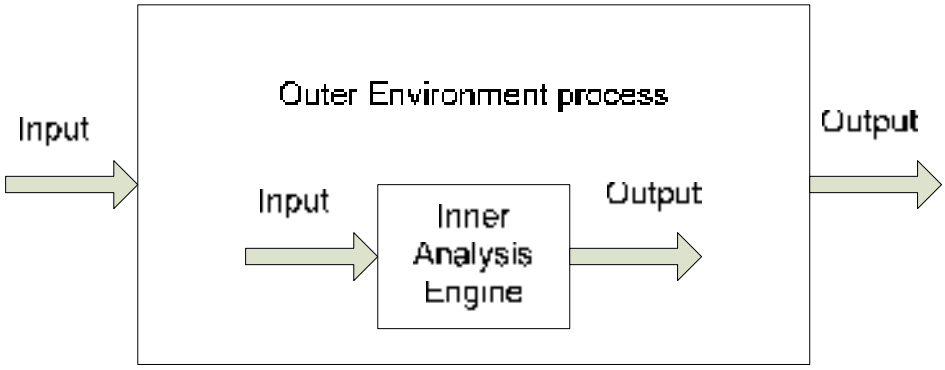

Figure 5: Block diagram of process interaction in user re-authentication system

Some common definitions used in the development of this system are:

- ***Keystroke Digraph Latency*** – It is the time taken between pressing two characters consecutively [19]. For example, in typing the word "Virginia Tech", the time taken by the user between pressing 'v' and 'i' denote the digraph latency for "vi". Similarly, the time between 'i' and 'r' denote the digraph latency for "ir". Among the 728 combinations possible (26 letters and space – the 'space space' combination is not considered 27*27 – 1 = 728), from preliminary research we observe that the frequency reduces drastically beyond 56 pairs. Rounding off, the 50 most frequently occurring alphabet pairs for analysis are chosen.
- ***Interaction Quotient (IQ)*** – It is the percentage of interaction performed by the user with the application, computed by the number of keyboard and mouse interactions collected. The interaction quotient is very high for a game (requires the users to continuously move the mouse or enter input into the keyboard) while it is very low for news websites (the user does not do any interaction other than move the mouse sporadically. The value of IQ is application and task dependent.

- ***Sensitivity*** – It is the percentage of correct identification of '*current-user*' being the '*logged-in user*'.
- ***Specificity –*** It is the percentage of correct identification of '*current-user*' not being the '*logged-in user*'.
- ***False acceptance rate (FAR)*** – It is the percentage of wrong identification of '*current-user*' being the '*logged-in user*'.
- ***False rejection rate (FRR)*** – It is the percentage of wrong identification of '*current-user*' not being the '*logged-in user*'.

The system design consists of two phases, (1) a learning phase and (2) a verification phase. In the learning phase, a simple exercise is given to the user and the user is asked to do it. The keyboard and mouse events are collected and various attributes are discovered from raw data. Using these parameters, a profile for each user is uniquely computed and stored. During the verification stage, once user passes the authentication point, keyboard and mouse events are collected in the background and analyzed. If the parameters of '*current-user'* match with the profile registered for '*logged-in user'*, then re-authentication is successful. Otherwise, the system reports that '*current-user'* is not the same as the '*logged-in user'*. On continued anomaly detection, the system logs the user out. This procedure is followed every few minutes so that re-authentication is continually done and the system knows for sure that the '*current-user'* is the '*logged-in user'*.

4.2.1 Learning phase

The learning phase is conducted during system setup. It consists of two parts: initialization part and training part. The initialization part consists of obtaining the number of users in the system, their usernames, passwords and asking each user to perform simple tasks. Each simple task consists of a set of windows '*forms*'. Each '*form*' collects different data from the user. The corresponding results are stored in the database.

The first '*form*' concentrates only on capturing user's keyboard movements (such as typing speed, other keyboard attributes etc.) by asking her to type her username and a sentence containing all the twenty six characters of English alphabet. Most of the top fifty combinations of characters and space are repeated twice in the sentence to get an average latency for top 50 character pairs. From [15], we see that the name of each user is typed uniquely by the user. Also, [19] shows that keystroke digraph latencies can be used to successfully authenticate the user. Combining these

results, the first '*form*' is used as a simple authenticator that gets the username and a simple sentence. At the time of authentication, this form is given again to user. The user is asked to perform the same exercise. If the results of the exercise match training data, the user is authenticated. Else, the user is asked to perform traditional authentication methods again.

The second '*form*' consists of a series of buttons that have to be pressed in a certain order as guided by the system. This form aims to exclusively capture the mouse movements of user.

The third '*form*' is used to capture both mouse and keyboard movements simultaneously. It consists of two sets of radio buttons with a text box below each of them. The user is required to choose one of the radio buttons in each list and type the corresponding choice in the text box. This is a more complex setting similar to the web pages where both the mouse and keyboard are used. From the results of preliminary study, it was found that the data collected for training was not enough to cover the broad spectrum of a user's behavior. On analysis, it was found that as the IQ increased, the system performed very poorly in detecting the user. Hence, the learning period was extended with varied tasks (from low to high IQ).

The fourth '*form*' was a direct representation of a simple game environment. The user will have to press a moving object repeatedly in short intervals. This gives a good example of a high IQ environment.

The fifth '*form*' is a keyboard-only input where a picture is described in a paragraph. The user has to enter at least 75 characters in the paragraph. This reflects the user's actions in a text-based application.

The sixth '*form*' involves both keyboard and mouse movements. It consists of a picture and requires the user to describe it with frequently used words. The paragraph must at least be 75 characters in length. This is followed by random clicking of buttons based on the system's instructions. This captures both keyboard and mouse movements in the same period of time.

Inputs from the initialization phase serve as training data for applications based on their IQ level. For example, a game application or search application that requires continuous interaction from the user uses the training data that has high mouse movement. A news site which requires very few clicks or a pdf application will use the training data that has very less mouse movement.

The system takes the results from the '*forms*' as initial values and starts the training phase. Here, the user is asked to use an application of his/her choice and perform normal operation. This gives an opportunity for the system to train in a general, less controlled setting. Based on the keyboard and mouse actions of the user, the system calculates the IQ and trains itself with suitable data. The analysis of results is done in a separate thread in the OS so that the user's normal operations are not affected and to ensure seamless user experience. The user uses the system for some time and the system tries to identify the user. Based on user input, the system continuously retrains itself. The training phase takes around 10 observations and consolidates the results of each observation by continual updates of various attributes. At the end of training session, the system identifies the user with a higher accuracy. This training phase improves the accuracy of the system in identifying the user by 15%.

During the training session, the system takes only instances (i.e., observation periods) when either or both the mouse and keyboard were used. Data samples from keyboard and mouse when they are not being used implies that the keyboard and mouse data for that session is zero (i.e.) the IQ for that session is zero. If the training system takes that data sample as part of user's behavior, it leads to erroneous results. The absence of data, in this case, is different from the value zero because it is theoretically possible for a person to have zero as the value for their characteristics. Hence, to safeguard the system from incorrect training inputs, samples with zero IQ are dropped.

At the end of training phase, the results are consolidated and stored in a text file. Then, the user is notified of the end of training session. This is repeated for all users of the system and their respective profiles are stored in the database. This marks the end of the first two phases of the system.

4.2.2 Verification phase

The verification phase begins with the login screen. Once a user logs in (the *'logged-in user'),* testing period begins. Her actions are observed for a given period and the resulting values are computed and checked against all the stored profiles. The profile that is closest to the observed values is termed the '*current*-user'. On comparison with '*logged-in user*', in case of differences, an anomaly is detected and the event is logged. After repeated detection of anomaly, the '*current-user*' is notified that the system detects an anomaly and is logged out.

A high level view of the analysis engine is shown in Figure 6.

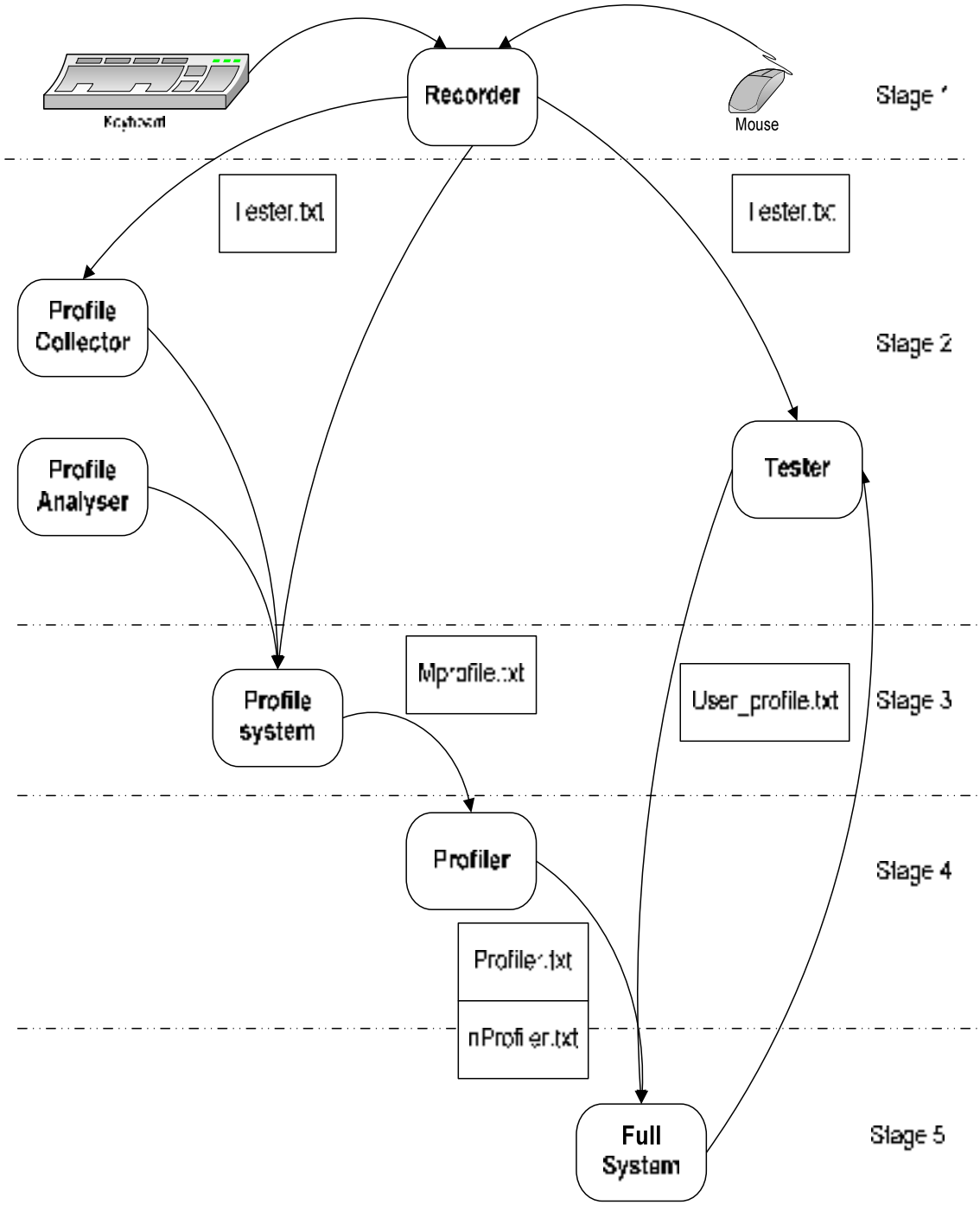

Figure 6: High level diagram of the processes and their interactions in the analysis engine

The system consists of seven processes linked to one another as shown in Figure 6.

Stage 1 – This stage consists of the keyboard logger and mouse logger used to capture raw data from user's actions. This process is called the Recorder and its life is controlled by processes in Stage 2. Output of Recorder process is stored in tester.txt file.

Stage 2 – This stage manages the data collection and basic analysis of collected data. Here, all the analysis functions are separate so as to enable clean integration with other methods in future. The processes are activated based on period of observation and number of intervals per user.

Stage 3 – A single process is present in this stage. It combines the profile collecter.exe and profile analyzer.exe processes into one system and creates a tangible output for each observation of the user. It also calls the Recorder.exe process. It creates Mprofile.txt at the end of the observation period. It is called once for every observation period. Here, the attributes are extracted from the pre-analyzed data.

Previous studies show that the correct selection of attributes plays an important role in the accuracy of the system. In the design of keyboard and mouse based authentication system, ten mouse attributes (general mouse speed and acceleration, mouse speed in eight directions) and 52 keyboard attributes (typing speed and acceleration, 50 most frequently used keyboard digraph latencies) are used for analysis.

Stage 4 – This stage consists of a single process that performs further analysis on the data collected so far and creates the final profile for every user. It activates the processes in stage 3 once for every user. The results from Mprofile.txt are analyzed and a master profile list of all the users is created. This is stored in profiler.txt and is used for comparison by the system during testing phase. This process is called by the main process for every user so that each user's data can be obtained, analyzed and stored in profiler.txt. In such manner, a profiler.txt is created from the '*forms*' based on the IQ of the '*form*'. Further, the data is modified to suit the application of neural networks and k-Nearest neighbor for the system. It consists of the initialization and training phases.

Stage 5 – This stage consists of the main program. This program controls all other processes and ties the initialization, training and testing phases together. It calls the process in stage 4 to perform initialization and training. Then, during normal operation, it calls the tester process periodically to observe the user's actions. The results are analyzed using data from profiler.txt, nprofiler.txt and uprofile.txt. The output is passed on to the outer environment system.

The inner analysis engine applies the data to three heuristics and obtains the results from each of them. The results are compared with each other. If all the heuristics identify the same user, it is sent as the result to the outer environment system. If the heuristics give conflicting results

(possible because of the differences in their approach), then their results are compared with the '*logged-in*' user. Even if one of the heuristics' results is the '*logged-in*' user, the analysis engine returns that heuristic's result as the analysis engine's result. Such an approach leads to a high probability of false-positives and negligent probability of false-negatives. A detailed discussion of the heuristics used in the system follows.

4.2.3 Statistical Analysis

'Statistical analysis' involves the usage of average differences and Pearson product-moment correlation. Average differences between the sample input and collected data are computed and the cumulative weight is calculated. Weights are assigned for the attributes based on their ability to differentiate between users. For example, typing speed has a higher weight compared to the digraph latency of one combination. Also, general mouse speed has a higher weight compared to the mouse speed in a particular direction. This heuristic identifies the user that has the highest cumulative weight and maximum correlation. One of the main reasons to choose statistical analysis as one of the heuristics of identification is that correlation basically finds the percentage of similarity of the input sample with the user profiles in the database. This provides a simple yet effective way to identify the user without intensive calculations. However, this method can fail in certain cases. Suppose there exists a user who moves the mouse with a slower speed on average. But, when she plays a game, she might move the mouse faster and hence not correspond to her usual behavior. In such a case, statistical analysis would match her better with a user who generally moves the mouse fast and hence identify the user incorrectly.

Statistical heuristic_training:

Obtain *the input values for all the attributes*
Do
    For *every user u in the database*
        A = Avg_diff_calc(u,p) where p is the test sample;.
        P = Pearson_Correlation(u,p,no_attributes);
    End
  A_{user} *= Min(A);*
  P_{user} *= Max(P);*
  If(P_{user} *== log-in user)*
        return(P_{user}*);*
  else if (A_{user} *== log-in user)*
        return(A_{user}*);*
  else
      return(P_{user}*);*
Until *end of training session*

Network ready for testing

Statistical heuristic_testing:

Obtain *the input values for all the attributes*
For *every user u in the database*
A = Avg_diff_calc(u,p) where p is the test sample;.
P = Pearson_Correlation(u,p,no_attributes);
End
A_{user} *= Min(A);*
P_{user} *= Max(P);*
If(P_{user} *== log-in user)*
return(P_{user}*);*
else if (A_{user} *== log-in user)*
return(A_{user}*);*
else
return(P_{user}*);*

4.2.4 Neural Networks

Our heuristic uses a partial feed forward neural network with back propagation method. It creates a neural network for every user and trains the network for that particular user. Further, for IQ with only mouse movements, the neural network disables the keyboard inputs and trains only with the mouse attributes. Mouse attributes are disabled for only-keyboard input sessions as well. This ensures that the system can better train for the inputs given. The neural network starts with random weights. The initialization data are given to the neural network and trained for a particular user. As the training data is given, the neural network is continuously retrained to predict the user correctly. When the test data is fed to the system, it is given to all the user networks and the one that has the highest prediction of accuracy is chosen from them. One of the reasons to use neural networks as a heuristic for identification is to obtain the complex system denoted by behavioral attributes as inputs and user as output. As a result, even when the user acts out of his normal behavior (for example, moving the mouse fast when playing a game), neural networks identifies the underlying similarities and matches the user to the corresponding user profile. It is very useful in identifying users in corner cases. A Mersenne Twister has been used to generate the random numbers needed for this heuristic.

Artificial Neural Network heuristic_base_training:

Obtain *the input values for all the attributes*
For *every user u in the database*
Initialize *the network with neurons, neuron connections and random weights*

Do

For *every sample t in training set*

Z = Network(t); *// feed forward network.*

Z_t *= Sample output from the data*

Find $(Z_t - Z)$, Δ_H *and* Δ_I

Using Δ_H *and* Δ_I *, modify the weights of each node in the network.*

End

Until *system classifies correctly or above stopping threshold*

End

Network ready for training session

Where

Δ_H → *the difference in sample output and calculated output is fed to the nodes in the hidden layer*

Δ_I → *the difference in the hidden layer results are fed to the nodes in the input layer*

Artificial Neural Network heuristic__training:

Obtain *the input values for all attributes*

Load *the system with trained weights from base training session*

Do

For *every sample t in training set*

Z = Network(t); feed forward network.

Z_t *= Sample output from the data*

Find $(Z_t - Z)$, Δ_H *and* Δ_I

Using Δ_H *and* Δ_I *, modify the weights of each node in the network.*

End

Until *system classifies correctly or above stopping threshold*

Network ready for testing

Where

Δ_H → *the difference in sample output and calculated output is fed to the nodes in the hidden layer*

Δ_I → *the difference in the hidden layer results are fed to the nodes in the input layer*

Artificial Neural Network heuristic_testing:

Obtain *the input values of all attributes*

Load *the network with trained weights*

For *every user u in the database*

Z = Network(t); feed forward network.

Z_t *= Sample output from the data*

End

Z_u *= Closest_to_1(Z);*

User = $u \rightarrow Z_u$

return(User);

4.2.5. k-Nearest Neighbor algorithm

The k-Nearest neighbor algorithm treats the set of attributes as dimensions and the set of samples as set of points satisfying certain relationships such as rotation, translation, etc. k-NN uses Euclidean distance, Chebyshev distance and Manhattan distance for analysis. Based on the most frequently occurring user in all the three distance calculations, the result is given.

Euclidean distance – the normal distance measurement between two points measured using a ruler or repeated application of Pythagorean theorem. Between any two points P and Q in n-dimensional space, the Euclidean distance is calculated as follows:

$$D_{Euclidean} = \sqrt{\sum_{i=1}^{n} (p_i - q_i)^2}$$

Chebyshev distance – the distance between two points is the greatest of their differences in all dimensions. Between any two points P and Q in n-dimensional space, the Chebyshev distance is calculated as follows:

$$D_{\text{Chebyshev}} = \max_i(|p_i - q_i|) = \lim_{k\to\infty} \left(\sum_{i=1}^{n} |p_i - q_i|^k \right)^{1/k}.$$

Manhattan distance – the distance between two points is the sum of (absolute) differences of their coordinates. Between any two points P and Q in n-dimensional space, the Manhattan distance is calculated as follows:

$$D_{Manhattan} = \sum_{i=1}^{n} (|p_i - q_i|)$$

The value of k is the minimum value of 3 (preset) or the largest of total number of users. The user identified the maximum number of times in this set is chosen to be the result. A user's behavior remains consistent over a large period of time. So, when we map the vectors on an n-dimensional space, we see that these points cluster together for each user. Since the k-NN algorithm can easily identify the user based on the test vector's distance from the clusters, it has been chosen for identification.

k-Nearest Neighbor heuristic_training:

Obtain *the input values for all the attributes*
Do
For *every user u in the database*
$D_{euclidean}$ *= Euclidean_distance(u,p) where p is the test sample;.*
$D_{chebyshev}$ *= Chebyshev_distance(u,p)*
$D_{manhattan}$ *= Manhattan_distance(u,p);*
End
E_{user} *= k-Min(*$D_{euclidean}$*);*
C_{user} *= k-Min(*$D_{chebyshev}$*);*
Muser = k-Min($D_{manhattan}$*) ; where k = Min(3,no_of_users);*
User = Max(E_{user},C_{user},M_{user}*);*
return(User) ;
Until *end of training session*
Network ready for testing

k-Nearest Neighbor heuristic_testing:

Obtain *the input values for all the attributes*
For *every user u in the database*
$D_{euclidean}$ *= Euclidean_distance(u,p) where p is the test sample;.*
$D_{chebyshev}$ *= Chebyshev_distance(u,p)*
$D_{manhattan}$ *= Manhattan_distance(u,p);*
End
E_{user} *= k-Min(*$D_{euclidean}$*);*
C_{user} *= k-Min(*$D_{chebyshev}$*);*
Muser = k-Min($D_{manhattan}$*) ; where k = Min(3,no_of_users);*
User = Max(E_{user},C_{user},M_{user}*);*
return(User) ;

Statistical analysis and k-NN algorithm cover the normal behavior of a user input while neural network covers excited/abnormal behavior of the user. All three heuristics are important in reaching a good accuracy. The outer environment system obtains the result from the inner analysis engine and compares it with the '*logged-in user*'. If it is the same, then the user has been re-authenticated. If it is different, the anomalous behavior is noted. The re-authentication begins again. If the number of anomalies crosses a set threshold, then the system reports the anomaly and requires the user to perform authentication again by traditional methods. A brief explanation of the algorithm used is given below:

Algorithm:

Initialize *the system*
Do
Add *user u to system*
Get the user's basic info with training experiment;
Extract_data();

```
  Compute_IQ();
  ANN_basic_training();
  profile=0;
  /*Training session begin*/
  Do
     Get data from user for specified time
     Compute_IQ();
     Statistical_heuristic_training;
     ANN_training();
     k-NN_training();
     report result;
     if correct_id
          profile++;
    else
         profile;
  Until profile==10
   /*Training session ends*/
Until all users are added
/*Testing begins*/
Do
   Log in user u
  Get data from user
  Compute_IQ();
  Statistical_heuristic_testing();
  ANN_testing();
  k-NN_testing();
  report result;
  if  correct_id
        continue;
  else
       flag++;
While flag<=threshold
if flag>threshold
     report security_breach
else
    continue_testing
end
```

Figure 7 and Figure 8 give a succinct view of the algorithm's operation:

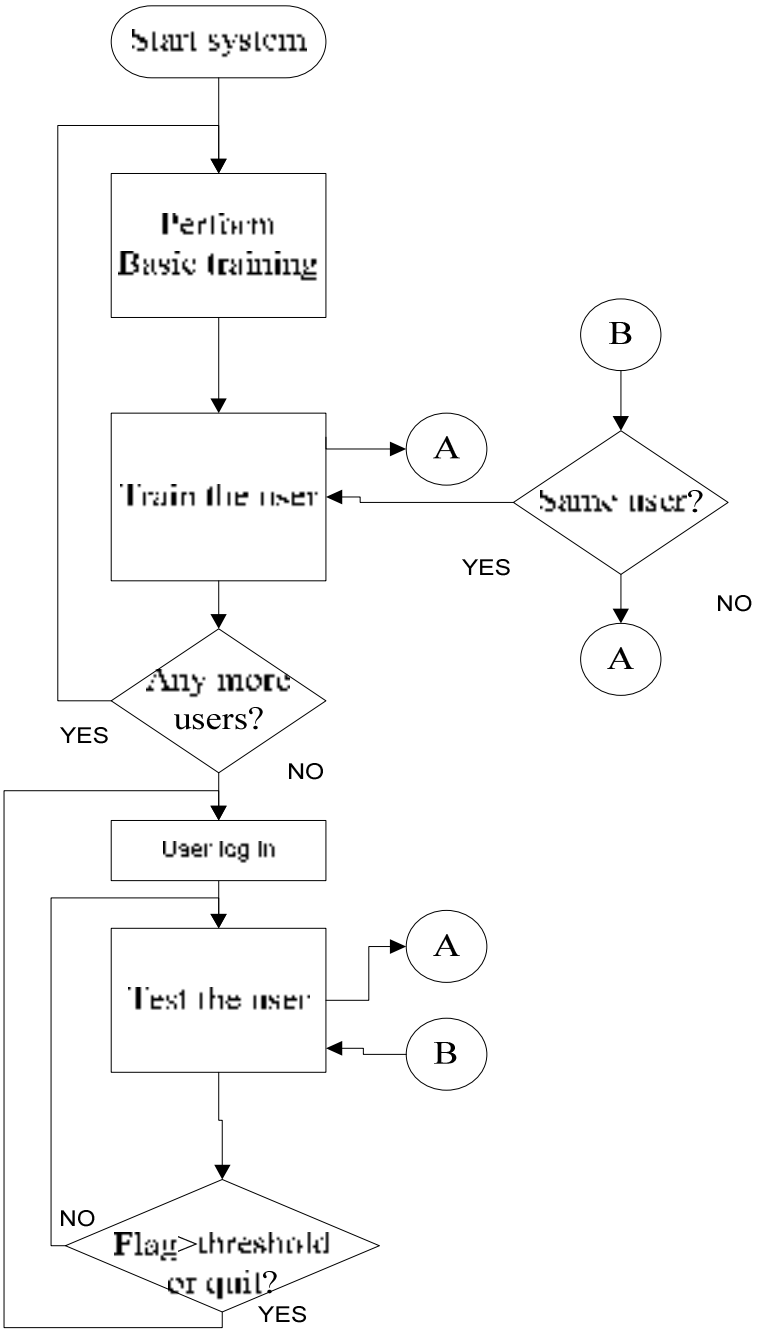

Figure 7: Flowchart explaining the algorithm

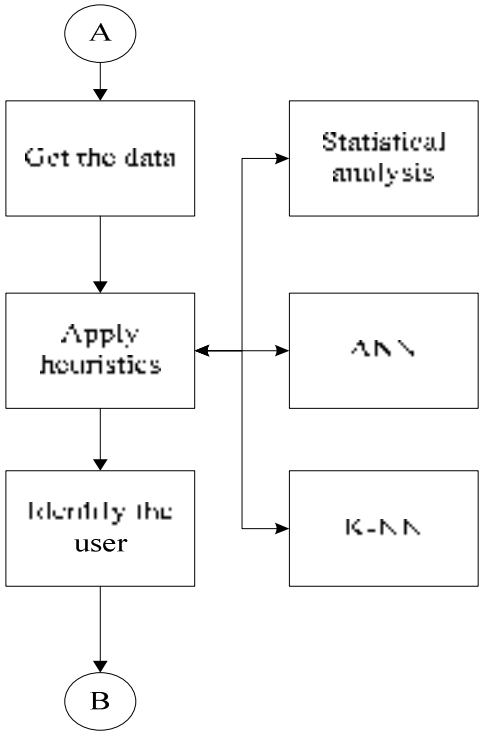

Figure 8: Flowchart 2 - explains the process of identification

4.3 Experimental Results

In this section, we present the results of user authentication and re-authentication system and corresponding observations. It is divided into three regions:

- User authentication
- Application based user re-authentication
- Application independent user re-authentication/General user re-authentication

4.3.1 User authentication results

The experiment conducted as part of preliminary study. It consists of 20 users with 40 samples obtained from each user. The users chosen were proficient with computer usage, worked for more than 5 hours per day on the computer and used a variety of applications on the computer. Each sample was validated with the data provided at training.

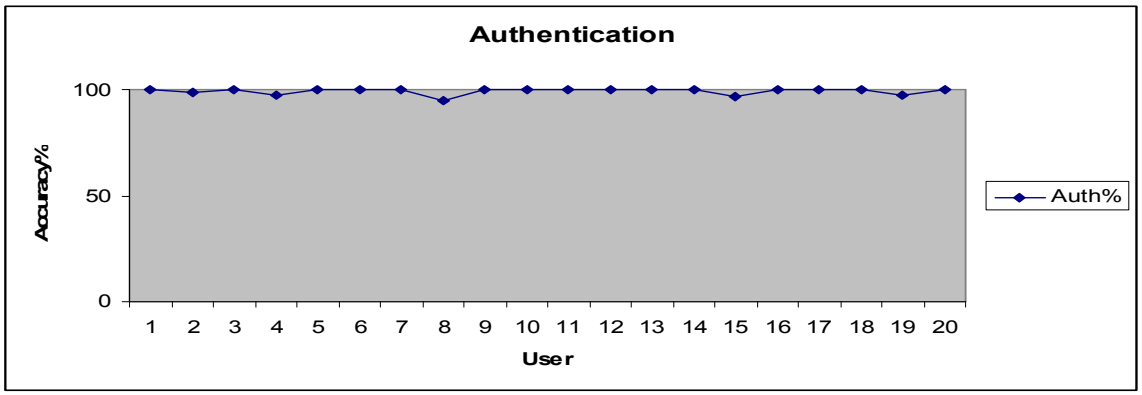

Figure 9: Graph showing the accuracy Vs the number of users in the authentication system

On the average, the authentication system has 99.28% accuracy. From Figure 9, we see that behavioral metrics can be used as effective authentication systems.

4.3.2 User re-authentication based on applications:

In this experiment, 5 users were chosen at random from the 20 users in the above pool. A group of applications ranging from the notepad to a simple game was selected. The users were asked to perform two sub-experiments. The training part in these two experiments was 1) general training in the first case and 2) application based training in the second case. For both, the testing part was application based. This procedure was repeated for all applications. Then the test data was validated using the general training data and the application-based training data. The results are shown in Table 3.

Table 3: Application based testing with general training

		2-users	3-users	4-users	5-users	Average
Application1	Notepad	100	100	100	100	100
Application2	Microsoft Word	100	98.33	85	79	90.5825
Application3	MS Outlook	86	69.33	62.5	58.33	69.04
Application4	Game	99.39	99.39	98.91	98.28	98.9925

Figure 10 shows the accuracy in identifying the user for each application with increasing number of users in the database.

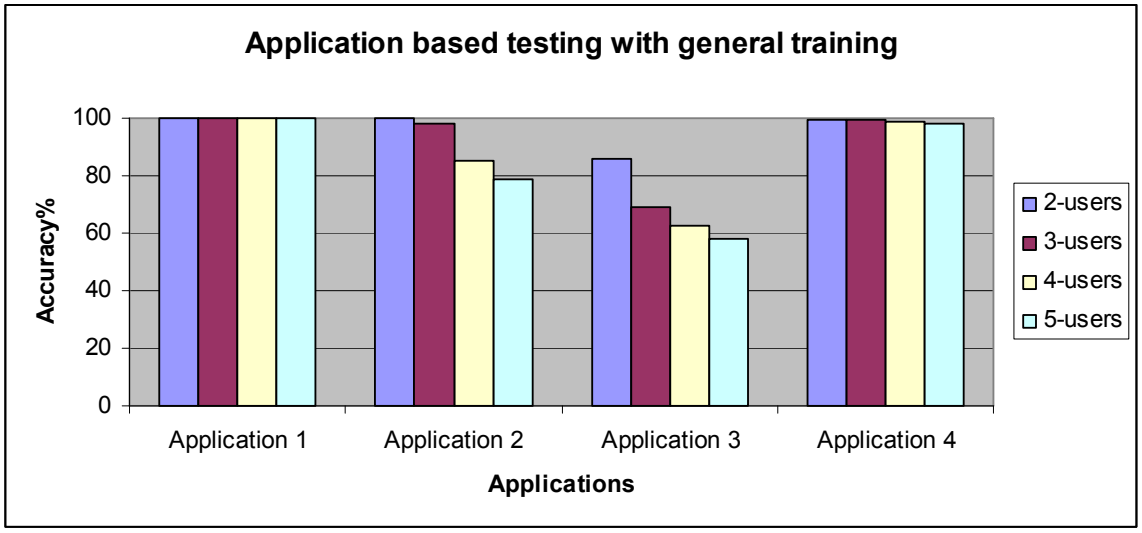

Figure 10: Graph showing the accuracy of system Vs the applications and the number of users for application based testing with general training

From Figure 10, we observe the following results:

- If the application's tasks closely match the general training tasks, the accuracy of the system is higher. For example, here, the general training was obtained from the notepad (random keyboard input) for the first three applications and '*form'* 4 (game environment – random mouse movement) for the fourth application. The applications used were the notepad, word, email client and a game. We see that the accuracy is very high (100% for notepad and 99% for the game) when the application activities closely match that of the training application activities. The accuracy drops as varied activities are introduced into the system. In application 2 (Word), though the activities of the application were similar to that of the training data (notepad), there were also certain other activities (such as clicking on tables, icons, etc.) that were absent in training. So, the IQ had a different mix of mouse and keyboard movements. Similarly, for application 3 (Outlook Express – an email client), the IQ had a higher percentage of mouse and a lower percentage of keyboard. So, the test data significantly deviated from the training data. This is evident in the lower average accuracy rates of 90.5% and 69%.
- As the number of users in the system increases, the accuracy decreases. This is intuitive. The more the number of users, the system has more choice and the user-regions can overlap with one another making the identification harder and resulting in higher false positives and false negatives. However, by the way the heuristics are designed, the probability of obtaining false positives is higher than false negatives.

By this analysis, we predict that if the system is trained with data from the respective applications before testing, the accuracy would be better. This leads to the next experiment. Here, the system was trained with the particular application's data and then testing was done. It resulted in Table 4 and Figure 11. Figure 11 shows the distribution of accuracy for applications with increasing number of users in the database.

Table 4: Application based testing and training

	2-users	3-users	4-users	5-users	Average
Application 1	100	100	100	100	100
Application 2	96.67	96.67	100	97.33	97.6675
Application 3	91.67	88.43	87.5	86.9	88.625
Application 4	99.39	99.39	98.91	98.28	98.9925

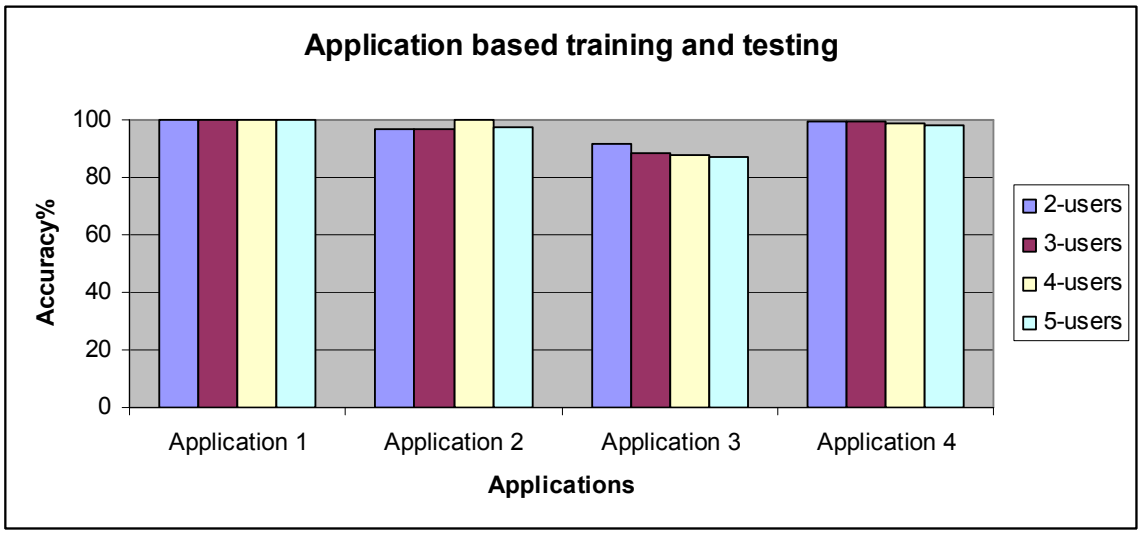

Figure 11: Graph showing the accuracy of system Vs the applications and the number of users for application based testing and training

From Figure 11, we observe the following results:

- This experiment proves the earlier hypothesis that system training with data from the respective applications before testing improves the accuracy markedly. This is evident from average accuracies of applications 2 and 3 (97.66 and 88.6) when compared to the corresponding average accuracies of applications 2 and 3 (90.5 and 69 – an increase of 8% and 28.5% over the earlier experiment).
- Here, we see that the intuitive result is not very clear any more. Though we see slight decrease in the accuracy as the number of users increase, it is not marked. However, this is still a hypothesis in that this experiment is limited in the number of users and the

number of applications used. This result might also be due to the possibility of higher false-positives compared to false-negatives. So, further analysis is necessary to prove or disprove this hypothesis.

- In this experiment, although we see an increase in the accuracy levels for the applications, we see that the accuracy of one of the applications (application 3) is lesser than others. On further analysis, it is seen that the particular application has a higher percentage of mouse movements compared to others. This leads to the hypothesis that the keyboard attributes overshadow the mouse attributes and do not aid in accurate identification. This hypothesis is particularly important for neural networks because the more the attributes from the keyboard, more importance is given to the keyboard characteristics. Similarly for the mouse attributes. This leads to the next experiment.

Next, the system was trained with the particular application's data and then testing was done. However, in this training and testing analysis, the keystroke digraph latencies were not considered. So, only 12 attributes were considered (10 mouse attributes and 2 keyboard attributes) giving more importance to the mouse attributes. It resulted in Table 5 and Figure 12.

Table 5: Application based testing and training without keystroke digraph latencies

	2-users	3-users	4-users	5-users	Average
Application 1	86.67	88.33	90	89.81	88.7025
Application 2	95	88.33	90	88.33	90.415
Application 3	98.58	97.16	87.25	89.1	93.0225
Application 4	93.98	89.55	87.46	86.13	89.28

Figure 12 shows the accuracies for each application for users in the database. The number of users in the database range from 2 to 5 users.

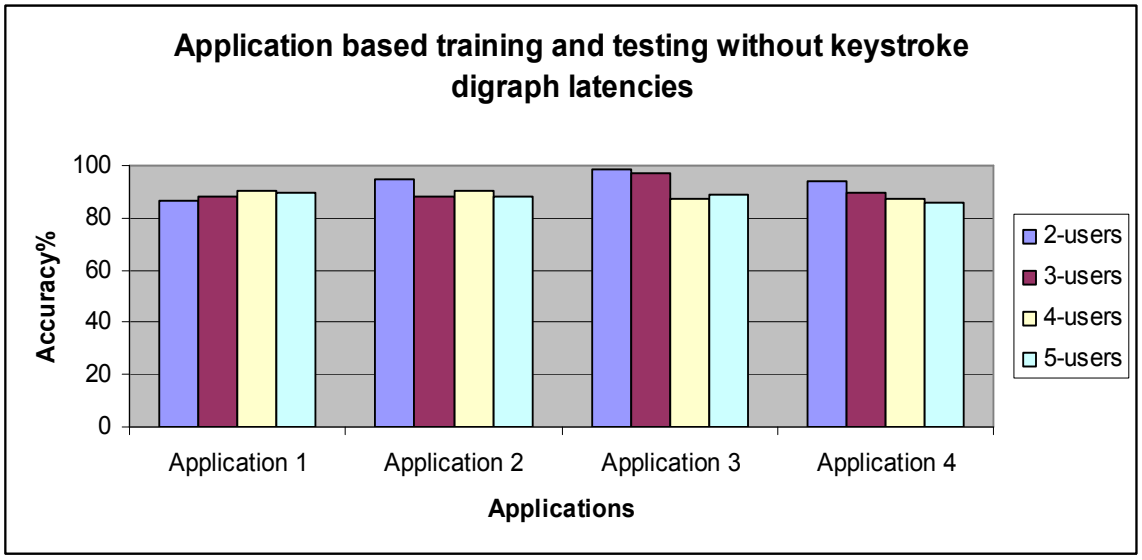

Figure 12: Graph showing the accuracy of system Vs the applications and the number of users for application based testing and training without keystroke digraph latencies

From Figure 12, we observe the following results:

- This experiment proves the earlier hypothesis that removing some keyboard attributes from the training and testing improves the accuracy markedly for applications with higher percentage of mouse movements. This is evident from average accuracies of application 3 (93) when compared to the corresponding average accuracies of applications 3 (88.6 – an increase of 5% over the earlier experiment). But we also see that the average accuracies of applications 1, 2 and 4 have decreased. Applications 1 and 2 have a higher percentage of keyboard movements. Removing some keyboard attributes reduced the effectiveness of the system in determining the user with the keyboard attributes. Though application 4 uses only mouse attributes, its accuracy is affected by lack of some keyboard attributes. This might be due to the interdependencies in the neural network.
- Here, we see that the intuitive result is not clear any more. Though we see decreases in the accuracy, we also see an almost equal number of increases in accuracy. So, this leads to the observation that less number of attributes leads to a reduced dependency on the number of users. Even a small number of users can lead to reduced accuracy while a larger number of users can have an increased accuracy. However, this observation can only be confirmed by further analysis where a large number of applications and users are used in the experiment.

In order to check if the above observation is true, the experiment is repeated for application based testing with general training. The results of the experiment are given in Table 6 and Figure 13 which show the accuracy of applications for users ranging from 2 to 5 users in the database.

Table 6: Application based testing with general training without keystroke digraph latencies

	2-users	3-users	4-users	5-users	Average
Application 1	86.67	88.33	90	89.81	88.7025
Application 2	95	81.66	65	55.67	74.3325
Application 3	90.17	80.42	87.5	83.33	85.355
Application 4	93.98	89.55	87.46	86.13	89.28

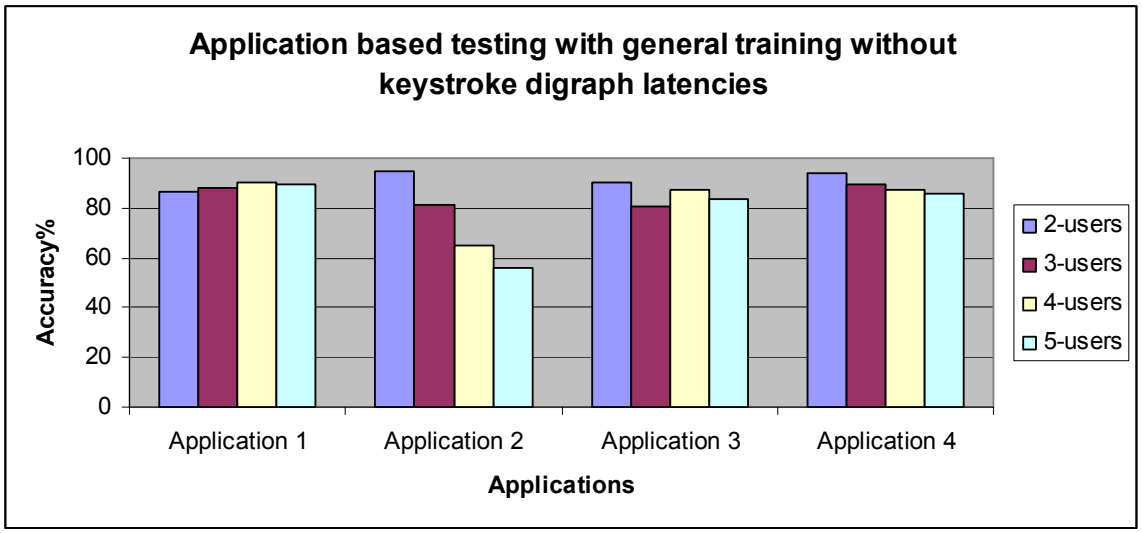

Figure 13: Graph showing the accuracy of system Vs the applications and the number of users for application based testing with general training without keystroke digraph latencies

From Figure 13, we observe the following results:

- This experiment confirms the above observations that
  - For text based applications, keyboard attributes are necessary for a high accuracy.
  - For applications with higher degree of mouse movements, keyboard attributes decrease the accuracy of the system.
- Here, no connection can be derived between the number of users and the accuracy of the system. So, as the number of attributes decreases, the relation between the number of users and the accuracy of the system is eclipsed by other factors.

The usage of heuristics has been shown in Figures 14-17:

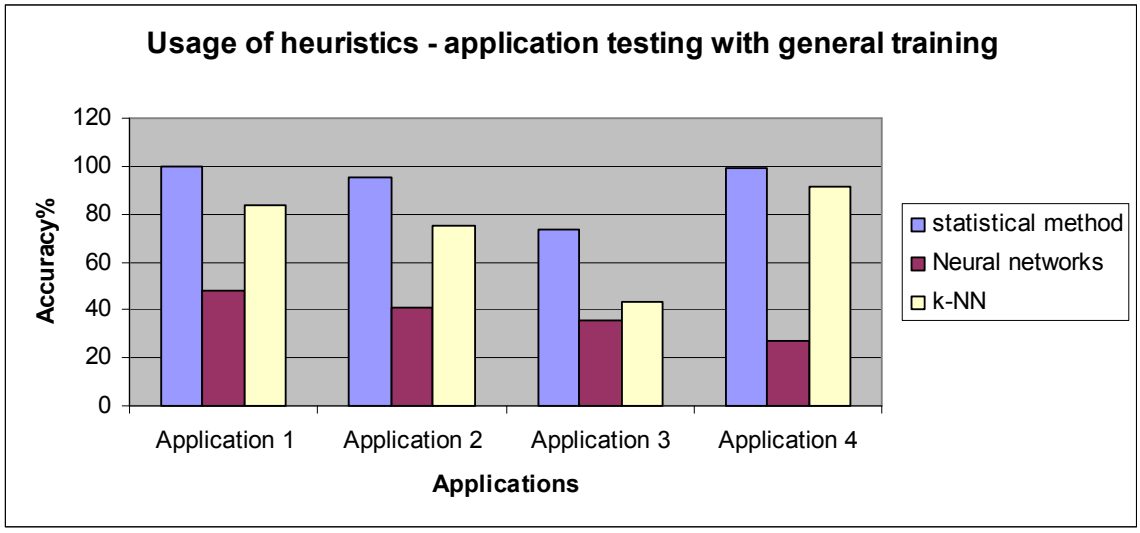

Figure 14: Graph showing the accuracy of system Vs the applications and the usage of heuristics for application based testing with general training

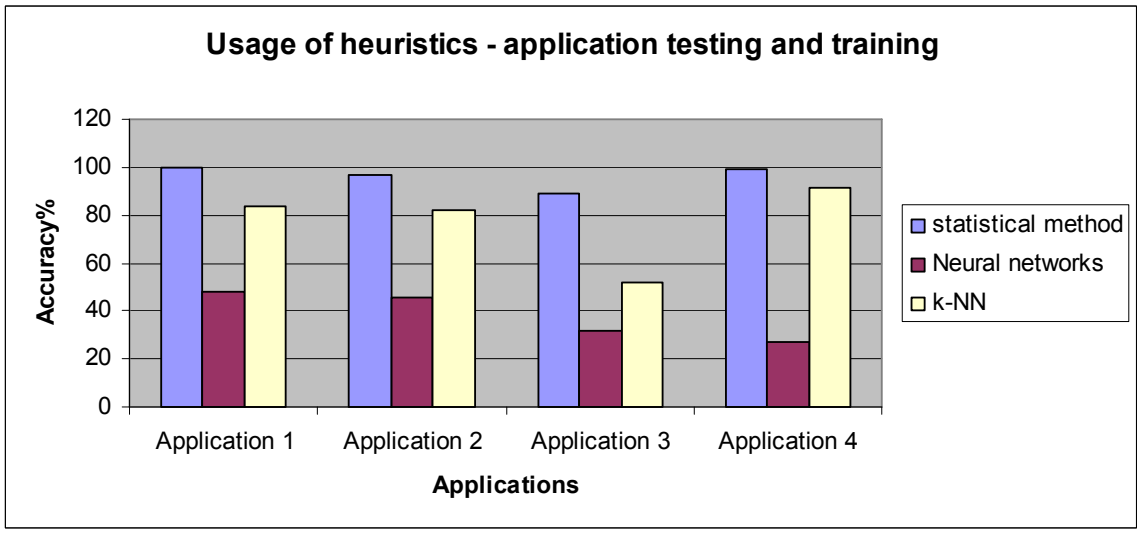

Figure 15: Graph showing the accuracy of system Vs the applications and the usage of heuristics for application based testing and training

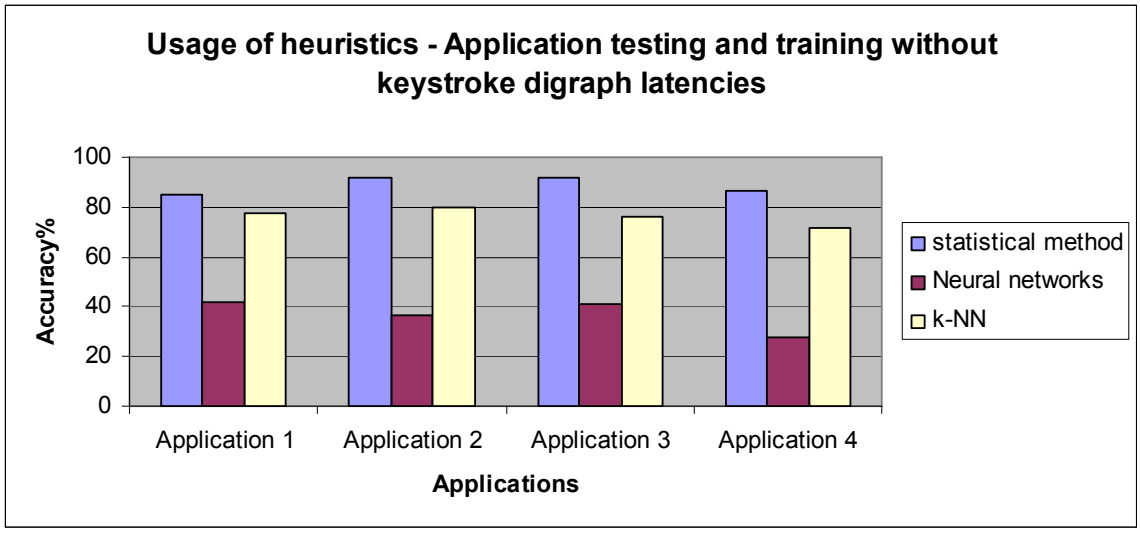

Figure 16: Graph showing the accuracy of system Vs the applications and the usage of heuristics for application based testing and training without keystroke digraph latencies

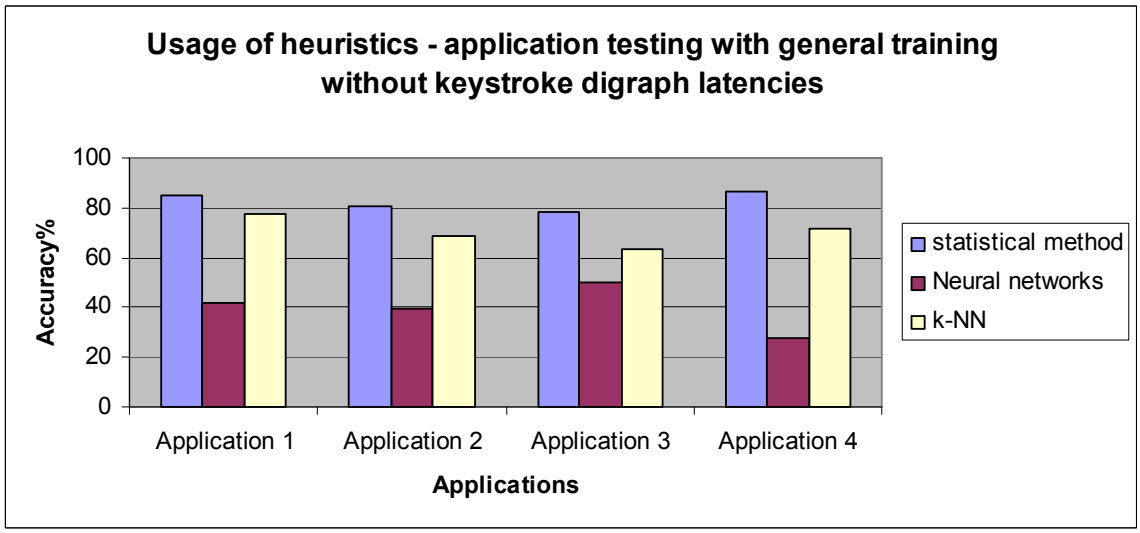

Figure 17: Graph showing the accuracy of system Vs the applications and the usage of heuristics for application based testing with general training without keystroke digraph latencies

In all the cases, we see that the statistical methods identify most of the commonly occurring cases for each user. The neural networks and k-NN identify less number of cases with excited/abnormal behavior of the same user. However, some of these cases are corner cases (not found by any other heuristic) thereby justifying the usage of this heuristic.

4.3.3 Application independent re-authentication/General re-authentication

In this section, an attempt has been made to design an application independent re-authentication system. Here, the same heuristics used for authentication system have been applied. However, the difference lies in training and testing of the system. During training phase, the user is asked to perform normal operations on any application of her choice. The user can even shift between applications and use either or both keyboard and mouse in each session. During testing phase, the user is asked to perform normal operations. Her actions during testing are not constrained to a particular application. On analysis, we see that the results are very good when the user's actions match that in the training phase. For example, in the training phase, if the user has predominantly used mouse-based actions, then the system can better identify the user when she does only-mouse activities compared to mouse-and-keyboard or only-keyboard activities. This is true for other activities too. We also see that the prediction is hardest when both mouse and keyboard are used.

For this experiment, 4 users were selected from the pool of 20 users in the preliminary study. Each user group (2-user, 3-user and 4-user groups) was asked to train and test with normal operations. Thirty samples were taken from each user group (20 samples where the '*current-user*' is the '*logged-in user*' and 10 samples where the '*current-user*' is not the '*logged-in user*'). From the analysis of these sample results, Table 7 is obtained.

Table 7: Application independent User Re-authentication

Number of users in system	Sensitivity	Specificity	False Acceptance Rate (FAR)	False Rejection Rate (FRR)	Accuracy
2-users	100	70	30	0	90
3-users	100	50	50	0	83.33333
4-users	85	50	50	15	73.33333

All the values of Table 7 are expressed in terms of percentages. Accuracy is expressed as the percentage of correct identification instances to total identification samples (30, here). Plotting the results in Table 7 into a graph, we obtain Figure 18.

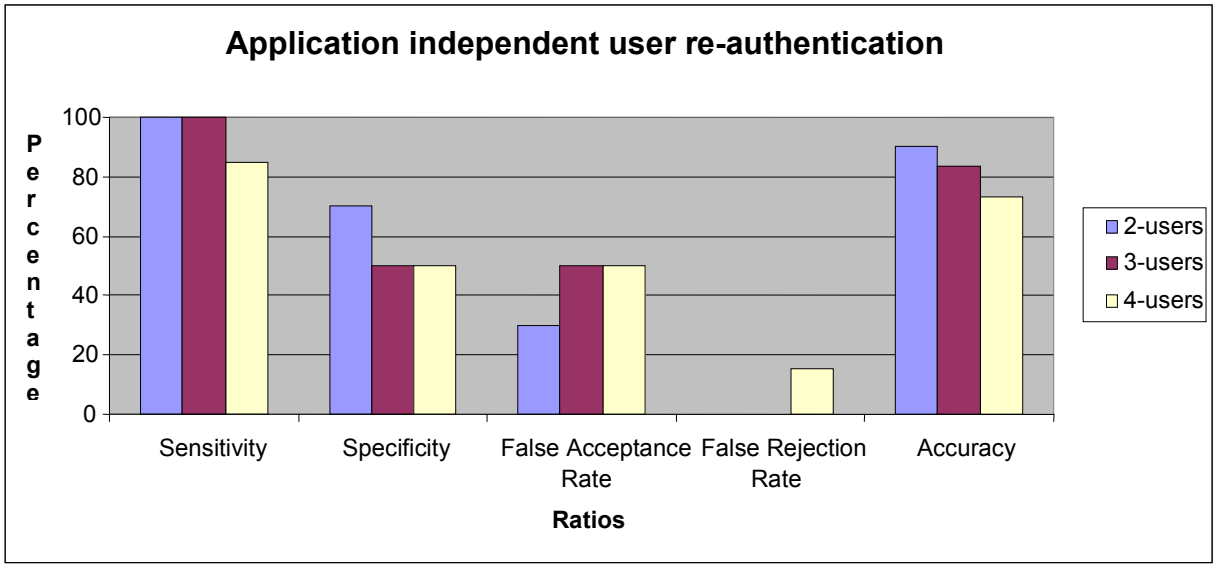

Figure 18: Graph showing ratios Vs percentages for application independent user re-authentication

From Figure 18, we observe the following results:

- The sensitivity and specificity decrease as the number of users increase in the database. Accordingly, accuracy also decreases.
- The FAR and FRR increase as the number of users increase in the system. Though the FRR is acceptable, FAR is high.
- The average accuracy of the system is around 82%.

Experiments with larger user groups and larger sample size per user group may provide a more accurate view of these ratios in the system.

4.4 Summary

In this chapter, we have designed and developed a user re-authentication system using three different heuristics. With the help of experimental results, we have shown that the system gives best results if

- Each application has a training data collected from that application.
- For applications with either high-keyboard-interaction or high-mouse-interaction, corresponding attributes are included in the analysis.
- For applications with both keyboard and mouse interaction, based on the percentage of mouse movements, the attributes are either added or removed from the analysis.
- The overall accuracy of the system for application based user re-authentication is around 96%.

- For application independent user re-authentication system, the training data is chosen based on the IQ of the application used.
- The accuracy is highest for only-mouse activities or only-keyboard activities based on the training data given by the user. It is harder when mouse-and-keyboard activities are given to the system.
- The overall accuracy of the system for application independent user re-authentication is around 82%.

Chapter 5

Software Testing and Verification of User Re-authentication System

In this chapter, we describe the procedures for testing and verification of user re-authentication system software. The ‘Motivation’ section describes the rationale behind this work, ‘Approach’ describes the architecture and methodology used to build the testing framework, ‘Experimental Results’ analyze the results obtained and finally ‘Summary’ concludes the chapter with possible future enhancements.

5.1 Motivation

In Chapter 4, the design methodology and development process of user re-authentication system were discussed. The SRS document of the system is included in Appendix B. In this chapter, validation and static verification for the user re-authentication system were performed using existing industry standard tools such as .NET memory validator [36], .NET performance validator [37] and Boogie Spec# [38] while the dynamic verification (software testing) was performed using test cases.

5.2 Approach

From the analysis shown in the SRS document, we see that, theoretically, the total number of paths that can be taken by the system from input level to output level is 1.227E49. Most parts of the 1.227E49 paths of the system are redundant. So, trying to execute each of them would lead to unnecessary waste of resources. Redundancy arises from the fact that some paths in different functions are mutually exclusive. Also, path selection of certain processes depends heavily on the data input by the user. Applying these constraints and thereafter considering the ways a user can input data into the test environment, the number of paths from the input to the output of the system to be tested is reduced to 2543382. The selection of paths is detailed in Chapter 2.

The path segments are divided based on the functions. The test environment consists of methods to input the user data automatically. We aim to reach 100% path coverage by executing each of these paths at least once. It ensures that any combination of these paths will also be executed correctly with a high degree of confidence.

5.2.1 Black box testing

Tests are based on requirements and functionality. The testing procedure for the user re-authentication system begins with black box testing. This is a random testing method which covers some of the most commonly used paths leaving the testing person to concentrate on the hard-to-reach paths in other phases of testing. This also assures a high degree of reliability on the operation of the entire system.

Recall that in black box testing, internal system design is not considered. While applying black box testing to user re-authentication system, all the twenty six letters are typed from the keyboard. Also, the mouse is moved in all directions and mouse buttons are clicked. The basic training phase of user re-authentication system consists of seven windows '*forms*' that take in user input through the mouse and keyboard. During testing of the basic training phase, these 'forms' are traversed through with different inputs to them. This ensures that different paths are chosen every time for various inputs. The black box testing procedure for our system is described with a sample walkthrough in Appendix C.

In our system, the total number of paths covered by black box testing is 346504. This translates to 13.63% of path coverage. Although statement coverage reaches 100%, path coverage is not satisfactory for black box testing. Some of the reasons for this are:

- There is a number of file manipulation operations that require the use of 'using' statement and 'try-catch' block. These code segments generally take the same path during successive system runs.
- A number of path divisions happen inside the heuristics and their environments. These path divisions are due to 'if' conditions that are placed as checkpoints to ensure that the heuristic is operating properly. Most of these paths are chosen based on previous stage input and it is likely that they follow the same path for valid data entries.

The results for path coverage are shown in Table 8 with the process and class name, total number of paths, number of paths covered through black box testing and the number of paths yet to be covered. For example, consider the Main_Env class. It has 17 paths of execution from all the functions in it. All of them are covered by black box testing. However, for class Formf1(), we see that there are 125 paths among which 124 of them are covered by black box testing. One path is yet to be covered. As a result, the path coverageof class Formf1() is 99.2%.

Table 8: Path count of Black box testing of user re-authentication software

Process and Class name	Total number of paths	Number of paths executed	Number of paths yet to be covered
Main_Env	17	17	0
Form_t	7	7	0
Formf1()	125	124	1
InputListener	12	11	1
MouseMoveEventArgs : EventArgs	1	1	0
KeyPressEventArgs : EventArgs	1	1	0
chker1	45	29	16
chker2	30	30	0
chker3	75	75	0
chker4	10	10	0
chker5	5	5	0
chker6	35	34	1
Formf2	141	140	1
chker7	31	14	17
prof_analyse	236312	91896	144416
comb_analyse	977440	121340	856100
Program1	288	186	102
Program2	5774	5771	3

prof_ab	2040	1356	684
Usertest	92178	23050	69128
Usertrain	1228817	102409	1126408

5.2.1.2 White box testing

When we analyze the remaining paths that have not been covered, we see that most of them are constrained by file operations such as file read, file write, file append, etc. In our code, they have been implemented in blocks initialized by 'using' statement. Since, in our analysis, the error messages are unnecessary, we do not use a 'catch' block. The file manipulations are done inside the 'using' block as it takes care of safely disposal of the file handlers. In order to test the 'using' statement, it is enclosed within a 'try' block with a respective 'catch' block. After the 'using' block is executed, the control always passes through the 'finally' block. In our implementation, the results from the 'try' and 'catch' blocks in the test bench are stored in resultlog.txt inside the user directory. A detailed explanation of these C# statements is given in Chapter 2.

Further, some of the paths are not executed during normal operation but are added to the code as safe guards against memory allocation errors, function mishandling errors, etc. For example, most objects are tested for null errors before they are used. This is especially true in cases where file handling is done as there are higher chances of file mismanagement to occur at the OS level. However, when a system successfully performs its intended operations, these paths are not executed. Various studies show that these errors occur less than 0.02% of the total number of executions of any software system. Even though 0.02% is a small fraction, yet for a large code, this may amount to substantial number of paths. These paths are important and cannot be excluded from the code. So, unit test cases that target and simulate these errors are necessary to test these paths.

Once the code segments before and after the 'using' statement have been modified to increase the testability of the software, the unit testing of the function is done. The unit testing is enabled/disabled by defining the token named TEST during compilation.

White box testing (also called Glass Box testing) is used for a few cases. This type of testing is based on knowledge of the internal logic of the code of an application. Internal software and working of the code should be known for this type of testing. Test cases are based on coverage of code statements, branches, paths, conditions. A complete test suite of white box testing and black

box testing will give 100% path coverage. In our case, black box testing followed by white box testing of the software increases path coverage from 13.67% to 86.73%. The results of unit testing are shown in Table 9. Consider class Formf1() with 125 paths. Among them, 124 paths are covered by black box testing. One path is covered by appropriate unit test case in white box testing. So, this class has 100% path coverage. However, when we consider the InputListener class, we see that there are 12 paths among which 11 are executed by black box testing. There remains one path that is not covered by both forms of testing. So, the path coverage of this class is only 91.67%.

Table 9: Path count of White box testing of user re-authentication software

Process and Class name	Total number of paths	Number of paths already tested	Number of paths uniquely covered by unit testing	Number of paths yet to be covered
Main_Env	17	17	0	0
Form_t	7	7	0	0
Formf1()	125	124	1	0
InputListener	12	11	0	1
MouseMoveEventArgs : EventArgs	1	1	0	0
KeyPressEventArgs : EventArgs	1	1	0	0
chker1	45	29	16	0
chker2	30	30	0	0
chker3	75	75	0	0
chker4	10	10	0	0
chker5	5	5	0	0
Chker6	35	34	0	1
Formf2	141	140	1	0
chker7	31	14	17	0
prof_analyse	236312	91896	144416	0
comb_analyse	977440	121340	849080	7020
Program1	288	186	102	0
Program2	5774	5771	3	0
prof_ab	2040	1356	684	0
Usertest	92178	23050	46072	23056
Usertrain	1228817	102409	819194	307214

On further analysis of the code, we see that there are some ‘if – else if – else’ blocks whose ‘else’ paths are never reached. When these code segments were modified accordingly and white box testing was repeated again, the path coverage increased to 99.99%. This is shown in Table 10.

Table 10: Path count of White box testing of user re-authentication software with design changes

Process and Function name	Total number of paths	Number of paths already tested	Number of paths uniquely covered by unit testing	Number of paths covered after code changes	Number of paths yet to be convered
Main_Env	17	17	0	0	0
Form_t	7	7	0	0	0
Formf1()	125	124	1	0	0
InputListener	12	11	0	0	1
MouseMoveEventArgs : EventArgs	1	1	0	0	0
KeyPressEventArgs : EventArgs	1	1	0	0	0
chker1	45	29	16	0	0
chker2	30	30	0	0	0
chker3	75	75	0	0	0
chker4	10	10	0	0	0
chker5	5	5	0	0	0
Chker6	35	34	0	1	0
Formf2	141	140	1	0	0
chker7	31	14	17	0	0
prof_analyse	236312	91896	144416	0	0
comb_analyse	977440	121340	849080	7020	0
Program1	288	186	102	0	0
Program2	5774	5771	3	0	0
prof_ab	2040	1356	684	0	0
usertest	92178	23050	46072	23056	0
usertrain	1228817	102409	819194	307214	0

There was only one path that could not be tested by both black box testing and white box testing. It involved a ‘while’ loop with a condition that always evaluated to ‘true’. This results in the ‘false’ path of ‘while’ loop never being executed. From software testing of user re-authentication system, we conclude the following:

- Since heuristics make up a larger part of this system, black box testing can only cover a small number of paths. Hence, path coverage by black box testing is small. This is in contrast to POCKET which is mostly user-interactive and hence has a high path coverage through black box testing.
- Most of the paths are covered by white box testing due to the presence of large number of ‘using’ blocks.
- A number of paths were found to be untestable because they were the result of ‘if-else if-else’ block. Depending on the condition of ‘else-if’ block, some of the ‘else’ paths never gets executed. Once this was corrected, the path coverage increased to satisfactory levels.

5.2.2. Software Validation

The memory validator [36] and performance validator [37] tools, when applied on the code, showed areas with high memory usage and regions with maximum and minimum performance percentages. Based on these results, the design was changed in the profiler.exe and full system.exe files. The initialization and comparison of data were conducted separately. This alleviated the need of creating objects every time data comparison and analysis was performed leading to a more efficient system with better memory management and higher performance. In addition, this change reduced the number of paths of execution by half when compared to previous design for every process. Splitting the initialization and comparison of data resulted in an increase in the number of functions. Since the testing is easier when there are more functions, this move made testing easier. However, memory management becomes worse with an increase in the number of functions. So, there has to be a balance between the ease of testability of paths of execution and the number of functions in the system.

5.2.3. Software Verification

The code to be verified is written in the Spec# programming language with the requisite non-null conditions, assertions, pre and post conditions, invariants and assumptions. This is fed to the Spec# compiler that performs both static and dynamic checks and converts the code to bytecode. This is then given to the translator which converts it to a form with BoogiePL language and feeds it to the Boogie engine which creates the verification constraints and conditions and passes it on to the internal SMT solver (Z3 by default) [38]. The SMT solver solves the constraints and gives the list of errors. If there are no errors, it assumes that all the conditions are satisfied (i.e.) the system is correct by verification.

Since the system to be verified was tightly integrated with the Windows Forms code in .NET framework, it was not possible to verify the code that directly links to the Windows Forms code. Hence, that section was not chosen for verification. Sections that included most of the calculations and analysis of the system were critical to the operation of the system. Based on this, three major sections of the system were chosen for verification. The first section (Section 1) is present in the profiler.exe process and imports the collected data from Mprofile.txt, analyzes it and stores the results in nprofiler.txt and profiler.txt. The second section (Section 2) is present in the fullsystem.exe process and obtains the data from profiler.txt, analyses it with the incoming

data from the current user and identifies the user based on the stored profiles using correlation and other metrics of the collected data. The third section (Section 3) is present in the fullsystem.exe process and obtains the data from nprofiler.txt, trains the neural network with this data, analyses the data with the incoming data from the current user and identifies the user based on the stored profiles using neural networks. Since the rest of the system focuses primarily on the data collection from the user, it was integrated with the windows code which has been tested adequately through both black box and unit testing explained earlier. So, these parts are not verified using Spec#.

Table 11: Software verification features of user re-authentication system

Feature	Section 1	Section 2	Section 3	Section 2 + Section 3
Number of properties verified	61	66	62	128
Number of non-null verifications	10	10	13	23
Number of assertions proved	12	7	3	10
Number of preconditions	0	0	4	4
Number of post conditions	0	0	6	6
Number of invariants proved	39	49	36	85
Number of verifications sections	4	5	14	19
Time taken for verification to be complete(average)	7.646875	1.503125	1.546875	2.921869
Time taken for verification to be complete(maximum)	8.125	1.625	1.90625	3.546875
Time taken for verification to be complete(minimum)	7.34375	1.4375	1.328175	2.53125
Number of warnings	303	90	23	111
Number of errors	0	0	0	0

Table 11 gives a snapshot of the number and types of properties verified for our user re-authentication system and the time taken to verify them (average, maximum and minimum of 15 trials). It can be seen that the time taken for verification is higher for section 1 since it verifies more complex invariant properties. An example has been given in Chapter 2. However, the sections 2 and 3 have simpler properties and some of them overlap with one another. As a result, proving one of the properties reduces the time taken for proving other related properties. The separation of the program into verification sections is done by the static verifier. From Table 12 and Table 13 which show the separate timing values of section 1 and section 2 & section 3, it can be seen that the properties are concentrated in one part of section 1 and it requires a lot of time for

proving/disproving it. However, the properties are more evenly distributed in sections 2 and 3 leading to lesser time required to tackle each part of the program.

Table 12: Software verification results for section 1

Section 1	avg	Max	Min
abstraction	5.223958	5.515625	5.03125
part 1	0.351042	0.40625	0.328125
part 2	0.010417	0.015625	0
part 3	2.025	2.140625	1.953125
part 4	0.036458	0.046875	0.03125
total	7.646875	8.125	7.34375

Table 13: Software verification results for section 2

Section 2	avg	Max	Min
abstraction	1.044792	1.09375	1.015625
part 1	0.339583	0.375	0.328125
part 2	0.009375	0.015625	0
part 3	0.095833	0.109375	0.09375
part 4	0.011458	0.015625	0
part 5	0.002083	0.015625	0
total	1.503125	1.625	1.4375

Table 14: Software verification results for section 3

Section 3	avg	Max	Min
abstraction	0.788542	0.890625	0.71875
part 1	0.360417	0.390625	0.34375
part 2	0.016667	0.03125	0.015625
part 3	0.0125	0.015625	0
part 4	0.188542	0.234375	0.171875
part 5	0.020833	0.03125	0.015625
part 6	0.016667	0.078125	0
part 7	0.029167	0.046875	0.015625
part 8	0.041667	0.046875	0.03125
part 9	0.010417	0.015625	0
part 10	0.014583	0.046875	0
part 11	0.007292	0.015625	0
part 12	0.013542	0.015625	0
part 13	0.021875	0.03125	0.015625
part 14	0.004167	0.015625	0
total	1.546875	1.90625	1.328125

The number of warnings reduces from section 1 to section 3. This is inversely proportional to the number of verification sections. Since most of the operations of section 1 are tightly integrated (i.e.) they are present in the same function, it is harder to verify them. In contrast, the operations of section 3 are split into a number of functions that operate with one another and hence it is easier to split them into distinct verification sections and prove/disprove the properties. However, the tight integration of the different operation of section 1 cannot be avoided or circumvented because of the inherent nature of the operations. The program in section 1 has a large number of variables that are used in every operation. Passing them between functions at every function call leads to a large memory management requirement. It is not advantageous to do so since the OS would spend half the time of operation in performing memory management functions. Any advantage obtained by splitting the operations into functions is lost due to this reason. But the program in section 3 has a small number of variables that are used in most functions and can be easily passed between functions without huge memory overhead.

We strived to reach zero errors in each of the sections and reduce the number of warnings in each section. However, some warnings are due to the method in which the compiler operates and cannot be rectified by the user. In particular, all the warnings that are accounted for in Table 14 belong to one of the two following warnings:

- *warning CS2614: Receiver might be null(of type object)*
- *warning CS2638: Using possibly null pointer as array.*

Due to the compiler's style of handling the code, the object_instance.component is used directly instead of capturing it to a variable and then being used. Inside the compiler, the object_instance is captured to a null point and its component is directly accessed. The contents of the component are used for calculation. For example, say we use profile[i].username. Inside the compiler, the profile[i] value is caught by a null point and its username component is redirected to the calling point. According to the static verifier, an object of non-null type cannot be assigned to an object of null type. This is due to the fact an object of null type can be successfully type cast into an object of any non-null type. However, the compiler cannot ensure that the new non-null type is the same as the original non-null type. This can lead to errors while applying functions to the non-null type values. So, it shows a warning. However, this warning is not shown by normal C# compiler. The C# compiler concentrates only on the operations done by the user and user-compiler interaction interfaces. Also, since this is a compiler-level warning, it is handled by the

C# compiler since it 'knows' that it will not assign an object of type null to an object of type non-null if the source object type and the destination object type are not compatible.

However, the object of null type, created by the compiler, cannot be overridden by the user and changed to an object of specified non-null type because it is done at a lower level. Hence, any such assignment would lead to a warning from the static verifier even though it does not affect the actual operation of the system. Since these warnings are caused by the operations of the compiler and cannot be changed by the programmer, they are ignored. This does not affect the final result from the verifier. The static verifier is run with the following options (the options that are not mentioned take the default value):

- modifiesDefault – 0 – the verifier verifies only the declared parameters of the particular function. This option is used because the sections have simple functions in them and do not need to check the properties of top level parameters too.
- loopUnroll – 0 – the verifier does only one stage of loop unrolling for 'for' loops and 'while' loops. It follows only the back edges up to one loop unrolling. All the remaining loop counts are treated to be a single sweep of execution. This reduces the complexity of the verification properties. This option is used because the 'for' loops and 'while' loops of the programs are used for simple assignment and calculation. No complex operation is done in them.
- summationStrength – 0 – less applicable triggers in the axiom.
- vcsMaxcost – 10 – the verification condition being proved will not be split unless the cost of a VC exceeds this number. This value has been set to a lower value in order to reach the results faster. There is the problem of some properties being unverified when a low value is set. But since the properties targeted are not very complex (due to limited loop unrolling and modification conditions), this value is sufficient to obtain a good result and to make sure that the properties are verified within the given limit.
- timeLimit – 100 – Limits the number of seconds spent trying to verify each procedure. From the results obtained, we see that the time taken to actually verify each procedure is very less. It was changed to 10 seconds in later experiments. However, there was not much difference in the actual time taken for verification. This is because this parameter sets a ceiling limit and does not make a difference to operations as long as they are within the range.

Although a very simple static verifier has been used for the verification of these sections, yet it has a myriad of options that can be used to verify more complex programs using suitable options. A snapshot of the verification is given in Figure 19. It shows a single trial of running both section 2 and section 3.

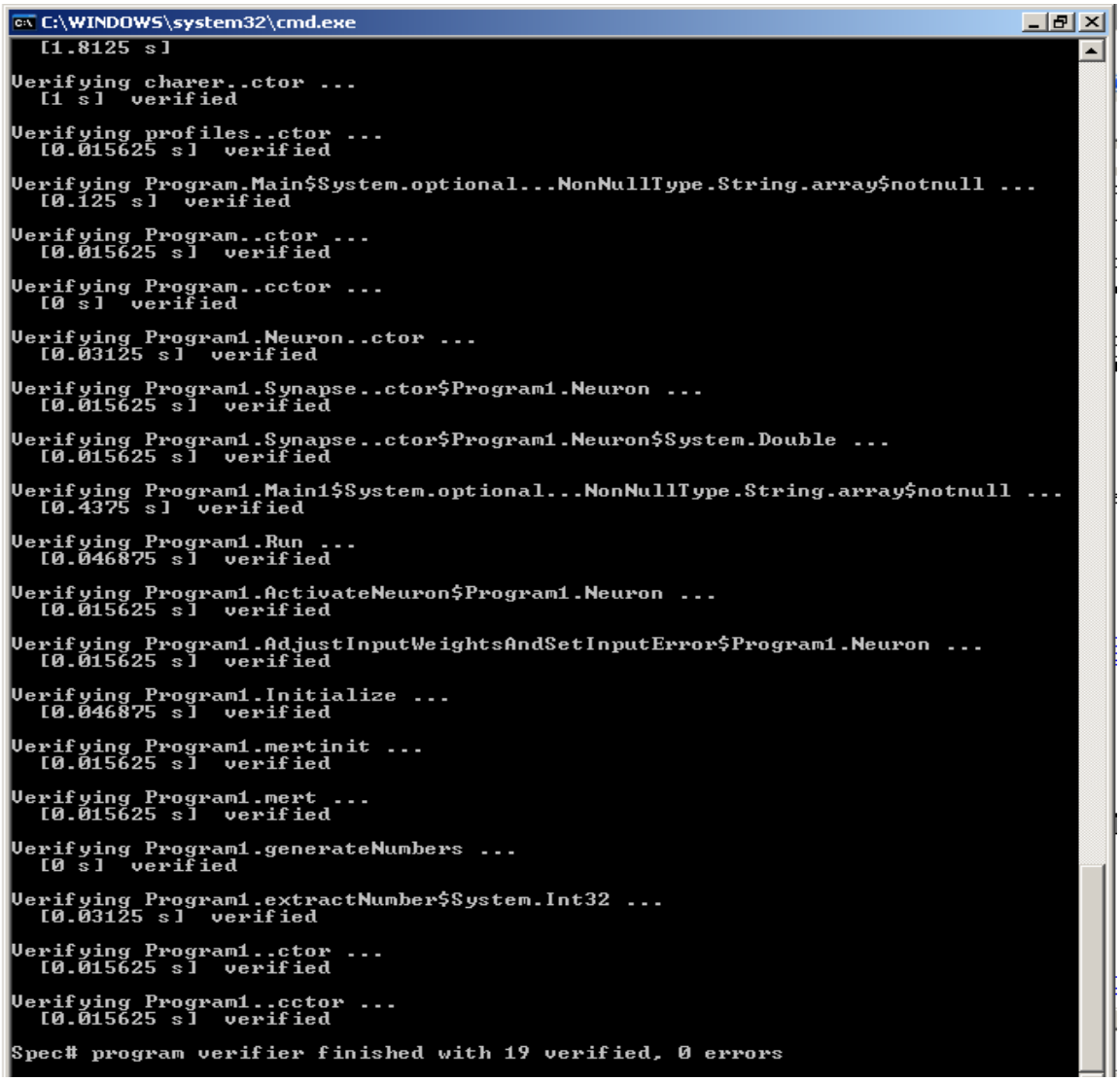

Figure 19: Snapshot of software verification of a part of user re-authentication system using Spec# and Boogie

5.4. Summary

In this chapter, we presented software testing, validation and verification procedures for the user re-authentication system. Every cycle of software testing, validation and verification lead to design changes and we observe that as the development cycle progresses, the frequency of bugs

and exceptions reduce elevating the coverage (statement and path) to 100%. We also find that the system conforms to expected requirements as defined in the SRS. Future work in this direction would include formal verification of other parts of the code and tighter integration of Spec# code with the C# code so that it becomes easier for the developer to verify the system thoroughly.

Chapter 6

Related Work

This chapter briefly discusses the related works done in the field of user authentication and re-authentication using keyboard and mouse attributes. In 1985, Denning suggested the use of behavioral attributes for distinguishing users in the system [1].

In the past, keyboard attributes were studied in a number of experiments. The earliest study using keyboard attributes found that human generated patterns can be used directly in systems to model normal behavior of a user through the user's command line input [2, 3, 4], keystrokes and other related parameters [5, 6]. In the last few years, Gaines, Lisowski, Press and Shapiro [13], Umphress and Williams [14], Garcia [15], Leggett, Williams and Umphress [16], Leggett and Williams [17] and Young and Hammon [18] have studied the use of keystroke characteristics in verifying identity of a person. Gaines et al. [13] use digraph latency times (viz., in, io, no, on, and ul) to obtain a zero false authentication ratio (FAR) and false rejection ratio (FRR). The experiments done in [14,16,17] use the user's typing speed and ratio of valid digraph latencies to total latencies in the test string for authentication. These works highlighted the insufficiency in [13]'s proposition of using only five digraph latencies for user authentication and proved that their experiments using all digraphs involving lower-case letters gave better results. Garcia [15]

used a person's name and performed complex analysis on it for user authentication. Young et al [18] employed a combination of features such as digraph latencies, typing speed, time to enter some common words (e.g., the, and, for) and used Euclidean distance for their comparison of registered and tested profiles. They also aimed for re-authentication (dynamic verification). However, the results were not satisfactory. Joyce and Gupta [19] extended Garcia's work by including PIN numbers along with a person's name. In [5], the authors used modified k-nearest neighbor algorithm for authentication. The experiments in [20, 21, 24, 25] achieve good results using back propagation method for limited number of users. However, they give a high FRR for larger databases. Some other algorithms used in keystroke authentication are fuzzy logic algorithms [22, 23], support vector machines (SVM) [26 27, 28] and rough sets induction algorithm [29]. Statistical classifiers were used in [32] to achieve high authentication accuracy using keyboard attributes.

The related work in keyboard authentication concentrates on analyzing the human behavior in keyboard usage and employs it to authenticate users in a controlled setting. Also, studies have been performed to differentiate the human behavior from that of robots using keyboard behavior. They leverage the intrinsic and fundamental differences between the human and bot keyboard characteristics to identify a bot from a human. This analysis is performed by developing a host based bot detection framework (TUBA) [47] and comparing the current user characteristics with that of bots and humans. These studies show that keyboard characteristics can be used for successful identification between humans and bots.

Mouse movements and other attributes were studied in [7, 8, 9, 10, 11, 12, 30]. Our work closely resembles [8] which also uses mouse attributes for re-authentication. In [8], the system consists of a training phase (includes collection of data, extraction of parameters and identification of user to the corresponding parameters) and a verification phase. It performs data collection over a window N to lower the false positive rate. It uses supervised learning and aims to detect anomalies using smoothing filter. Also, a decision tree classifier is used to authenticate the user. It recursively checks if the registered parameters match with the collected parameters for the logged in user. This experiment was performed for a single application.

In [6], cursor movements and mouse dynamics was examined in order to determine whether these attributes would be suitable for user re authentication. They defined user re-authentication as a technique suitable for detecting a hijacking scenario (e.g. someone has replaced the originally

logged in user). They considered both primary attributes such as cursor movements, mouse wheel movements, and clicks (left, middle, and center) and secondary attributes such as *distance*, *angle*, and *speed* between pairs of points. They used supervised learning techniques to identify the current user. The data was collected from eighteen subjects working within a single application, viz. Internet Explorer. A detailed user profile was created for each user using the collected data. The results generated an average false negative rate of 3.06% for all 18 users. The corresponding false positive rate was 27.5%.

Hashia [9] describes the results of authentication (active and passive) using different sets of parameters. Active authentication using mouse movements gives best results when all the parameters are within the range of **avg + 1.5 SD** and **avg - 1.5 SD**. However, their passive authentication study (using modified gift wrapping algorithm) did not prove to be effective. Hugo Gamboa et al [10] used statistical pattern recognition techniques to develop a sequential classifier that determines user authenticity based on a predefined accuracy level. Weiss et al [11] used the nearest neighbor algorithm for comparison with the pre-generated profile of every user. Some of the parameters used in [10,11] were mouse moving speed, number of clicks, variation of clicks and variation in mouse trajectory, etc.

Hocquet et al [48] perform an exploratory study to analyze user re-authentication by using a game based on fast mouse movements and clicks. A signature with attributes like speeds, accelerations, angular velocities, curvature, etc is created per user. These signatures are used to identify between users. The error rate of the experiment is 37.5 percent. Ahmed and Traore [30] try to improve on the attributes in [48] and used artificial neural networks to perform active authentication of users. Revett et al [31] deal with active authentication using mouse attributes through '*selection times*' and '*stroke times*'. Researchers at Pace University, NY performed a feasibility study in [33] to determine if mouse authentication is a viable security method using modified Next Nearest Neighbor algorithm.

Chapter 7

Conclusion and Future Work

Insider attacks are a huge but under-addressed problem in computer security. In this thesis, we have designed a user re-authentication system based on the behavioral attributes of the user which prevents unauthorized entry into the system. We first concentrated on improving the security of POCKET at the user level. Essentially, we try to identify a child from a parent even if the logged-on user is a parent. This system uses both keyboard and mouse attributes for analysis and identification. It is simple, continual, non-intrusive and quick in identifying anomalies in user behavior. It is independent of applications and can be used for re-authentication.

Firstly, a brief overview of the problem is given in Chapter 1. Secondly, in Chapter 2, we explained the technical details and implementation of POCKET. Also, we briefly discussed the various heuristics that serve as the base for our system. Important concepts such as biometrics, authentication and re-authentication have also been explained. Thirdly, in Chapter 3, applying software testing procedures to POCKET is discussed. We ensured that the system is extensively tested making sure that it is hard to be hacked as long as the user input is secure. We concluded that the absence of a user re-authentication system is the main drawback in the current setup.

In Chapter 4, we explained the design and development of the user re-authentication system using behavioral attributes from keyboard and mouse. '*Statistical analysis', 'feed forward neural network with back propagation'* and *'k-Nearest neighbor'* algorithms with IQ analysis proved to

be effective in differentiating between a variety of users. In Chapter 5, the testing and verification procedures for user re-authentication system have been discussed. The development cycle of user re-authentication system (testing, validation, verification and design changes) drives it towards achieving complete coverage for different metrics such as path coverage and statement coverage. We use Spec# to verify formal properties on our software system. Chapter 6 provides a brief analysis of relevant previous work in design of authentication and re-authentication systems using behavioral attributes. Finally, in Chapter 7, we conclude our work.

Overall, this work addresses one of the major security problems faced in today's computer world. The proposed system can successfully identify imposters from authentic users based on their behavioral attributes. Insider attacks can be reduced to a great extent with this system. On comparison with related work, our system has a lower false rejection rate. This in turn increases the accuracy of the system. The user re-authentication system for application based design has an accuracy of 96.4% while user re-authentication system for application independent design has an accuracy of 82%. Our results show that we have been successful to enhance the security of the system.

In future, physiological aspects of the user can be incorporated into our system. This may lead to higher accuracy in user re-authentication since it can model the user more accurately. In case of behavioral aspects, systems that include other mouse attributes such as mouse click, mouse double click, mouse wheel movement and keyboard attributes that include numeric keys, special keys such as Enter, Ctrl, F1-F12, symbols, direction keys can capture more information about the user's behavior may be designed. This leads to higher accuracy in identifying the user in a large user group. However, care should be taken when including these attributes to the system because a large attribute group can slow down the system's ability to quickly identify the user. Thus a trade-off between suitable attributes, data structures and heuristics is essential for the robustness of the system.

Bibliography

[1] D. E. Denning and P. G. Neumann. "*Requirements and model for IDES - A real-time intrusion detection system*". Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, 1985.

[2] S. Coull, J. Branch, B. Szymanski, and E. Breimer. "*Intrusion detection: A bioinformatics approach*". In Proceedings of the Nineteenth Annual Computer Security Applications Conference, pages 24-34, Las Vegas, NE, 2003.

[3] T. Lane and C. E. Brodley. "*Temporal sequence learning and data reduction for anomaly detection*". ACM Transactions on Information and System Security, 2(3):295-331, 1999.

[4] T. F. Lunt, A. Tamaru, F. Gilham, R. Jagannathan, P. G. Neumann, H. S. Javitz, A. Valdes, and T. D. Garvey. "*A real-time intrusion detection expert system IDES - Final report*". Technical Report SRI-CSL-92-05, SRI Computer Science Laboratory, SRI International, February 1992.

[5] F. Monrose and A. Rubin. "*Authentication via keystroke dynamics*". In Proceedings of the Fourth ACM Conference on Computer and Communications Security, pages 48{56, April 1997.

[6] J. Shavlik, M. Shavlik, and M. Fahland. "*Evaluating software sensors for actively profiling Windows 2000 users*". In Proceedings of the Fourth International Symposium on Recent Advances in Intrusion Detection, October 2001.

[7] J. Goecks and J. Shavlik. "*Automatically labeling web pages based on normal user actions*". In Procedings of the IJCAI Workshop on Machine Learning for Information Filtering, July 1999.

[8] M Pusara and C Brodley, "*User reauthentication via mouse movements*" In DMSEC 04, October 2004.

[9] S.Hashia, "*Authentication by mouse movements*", San Jose State University, December 2004.

[10] Hugo Gamboa & Ana Fred. "*An Identity Authentication System Based On Human Computer Interaction Behavior.*" In Pattern Recognition in Information Systems.2003

[11] Weiss et al, "*Mouse movements biometric identification: a feasibility study*", Pace University, May 2007

[12] Douglas A. Schulz, "*Mouse curve biometrics*", Pacific Northwest National Laboratory, U.S. Department of Energy

[13] Gaines, R., Lisowski, W.. Press, S., and Shapiro, N. "*Authentication by keystroke timing: Some preliminary results*". Rand Report R-256-NSF. Rand Corporation, Santa Monica, CA, 1960.

[14] Umphress, D., and Williams, G. "*Identity verification through keyboard characteristics*", Int. J. Man-Machine Studies 23, 3(Sept. 1985), 263-273.

[15] Garcia, J., '*Personal identification apparatus'*. Patent Number 4,6X,334. U.S. Patent and Trademark Office, Washington, D.C., 1986.

[16] Leggett, Williams, G., and Umphress, D. "*Verification of user identity via keyboard characteristics*". In Human Factors in Management Information Systems, J.M. Carey, Ed., Ablex Publishing, Norwood, NJ.

[17] Leggett, J., and Williams, G. "*Verifying identity via keyboard characteristics*". Int. J. Man-Machine Studies 23, 1 (Jan. 1988), 67-76.

[18] Young, J.R.. and Hammon, R.W. Method and apparatus for verifying an individual's identity. Patent Number 4,805,222 U.S. Patent and Trademark Office, Washington, DC., 1989.

[19] Joyce R and Gupta G, "*Identity authentication based on keystroke latencies*", Dependable computing.

[20] Bleha S et al, "*An application of fuzzy algorithms in a computer access security system*", In Pattern Recognition letters, 9:39-43, 1989

[21] Brown M et al, "*User identification via keystroke characteristics of typed names using neural networks*" International Journal of manmachine studies 39(6):999-1014, 1993.

[22] de Ru, W.G. and Eloff, J. "*Enhanced password authentication through fuzzy logic"*, *IEEE Expert*, Vol. 12, No. 6, pp.38–45, 1997

[23] Tapiador, M. and Siguenza, J.A. "*Fuzzy keystroke biometrics on web security*", AutoID '99 Proceedings Workshop on Automatic Identification Advanced Technologies. IEEE, pp.133–136, 1999

[24] Obaidat, M.S. and Sadoun, S. "*A simulation evaluation study of neural network techniques to computer user identification"*, Information Sciences, Vol. 102, pp.239–258, 1997

[25] Bleha, S.A., Knopp, J. and Obadiat, M.S "*Performance of the perceptron algorithm for the classification of computer users"*, Proceedings of the ACM/SIGAPP Symposium on Applied Computing, New York Press, 2002

[26] De Oliveira et al, "*User Authentication Based on Human Typing Patterns with Artificial Neural Networks and Support Vector Machines"*, SBC, 2005.

[27] Sang, Y., Shen, H. and Fan, P. "*Novel Imposters Detection in Keystroke Dynamics Using Support Vector Machines"*, LNCS Springer Berlin/Heidelberg, pp.666–669, ISSN 0302-9743, 2004.

[28] Sung, K.S. and Cho, S. "*GA SVM wrapper ensemble for keystroke dynamics authentication*", International Conference on Biometrics, Hong Kong, pp.654–660, 2006.

[29] Revett, K., Magalhaes, S. and Santos, H. "*Developing a keystroke dynamics based agent using rough sets"*, The 2005 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology Compiegne, pp.56–61 2005

[30] Ahmed A, Traore I, "*A new biometric technology based on mouse dynamics*", IEEE Transactions on dependable and secure computing, pp.165-179, 2007

[31] Revett K et al, "*A survey of User Authentication Based on Mouse Dynamics*"

[32] Araujo L, et al, "*User authentication through typing biometric features*", IEEE transactions on signal processing, Feb 2005.

[33] Weiss A et al, "*Mouse Movements Biometric Identification: A Feasibility Study*" conducted at the Pace University, NY, 2007

[34] R. Wright. "*2003 CSI/FBI computer security survey*", http://www.security.fsu.edu/docs/FBI2003.pdf, 2003.

[35] Hetzel, William C., "*The Complete Guide to Software Testing, 2nd ed"*. Publication info: Wellesley, Mass.: QED Information Sciences, 1988. ISBN: 0894352423.

[36] .NET memory validator http://www.softwareverify.com/dotNet/memory/index.html

[37] .NET performance validator http://www.softwareverify.com/dotNet/profiler/index.html

[38] Spec# Boogie http://research.microsoft.com/en-us/projects/specsharp/

[39] Vijaya Kumar BVK, et al, "*Correlation Pattern Recognition",* Cambridge University Press, ISBN : 0521571030

[40] Children's online privacy protection act, 1998 http://www.ftc.gov/ogc/coppa1.htm

[41] K. Channakeshava et al, "*On Providing Automatic Parental Consent over Information Collection from Children*", in Proc.the 2008 International Conference on Security and Management (SAM'08), July 2008

[42] P3P Project http://www.w3.org/P3P/

[43] Kung.S et al, "*Biometric Authentication, a machine learning approach*", Prentice Hall, ISBN: 0131478249, pg:1-20

[44] Looney.C, "Pattern Recognition using Neural networks – theory and algorithms for engineers and scientists", Oxford university press, ISBN: 0195079205, pg:124-126

[45] Micheli-Tzanakou.E, "*Supervised and Unsupervised Pattern Recognition – Feature Extraction and Computational Intelligence"*, CRC Press, ISBN 0849322782, pg:265-267.

[46] Beizer B. *Black-Box Testing: Techniques for Functional Testing of Software and Systems.* Wiley: New York, 1995. ISBN: 0471120944.

[47] Stefan D, *A Cryptographic Provenance Verification Approach for Host-Based Malware Detection* Rutgers University Technical Report. Apr. 2009.

[48] Hocquet. S et. al, *Users Authentication by a Study of Human Computer Interactions*, Proc. Eighth Ann. (Doctoral) Meeting on Health, Science and Technology, 2004.

Appendix A

Black Box Testing Sample for POCKET

This Appendix section gives a sample walkthrough of black box testing procedure for POCKET software. It explains the steps of POCKET with snapshots. For technical details and implementation of POCKET, the reader is referred to Chapter 2 or [41].

Black box testing for windows based applications consists of checks [46] as shown below:

1. Editable Fields:
    a. Data validity for characters/strings
    b. Data validity for minimum/maximum/mid range fields
    c. Blank fields
    d. Minimum length requirement
2. Non-editable Fields:
    a. Appropriate confirmation/error messages at every step
    b. Test all menu items and their options and paths

These checks, along with intuitive checks such as testing the right click function of a mouse, window minimize, maximize, restore, user interface checks, etc. are used in black box testing.

The registration phase is executed at the beginning of setup. It requires the parent to create a new account. It is used to enable/disable POCKET and change preference files. The first test involves the log in dialog box with two tests, password length at or more than 6 characters and password consistency check. This is shown with Figure A1, Figure A2 and Figure A3.

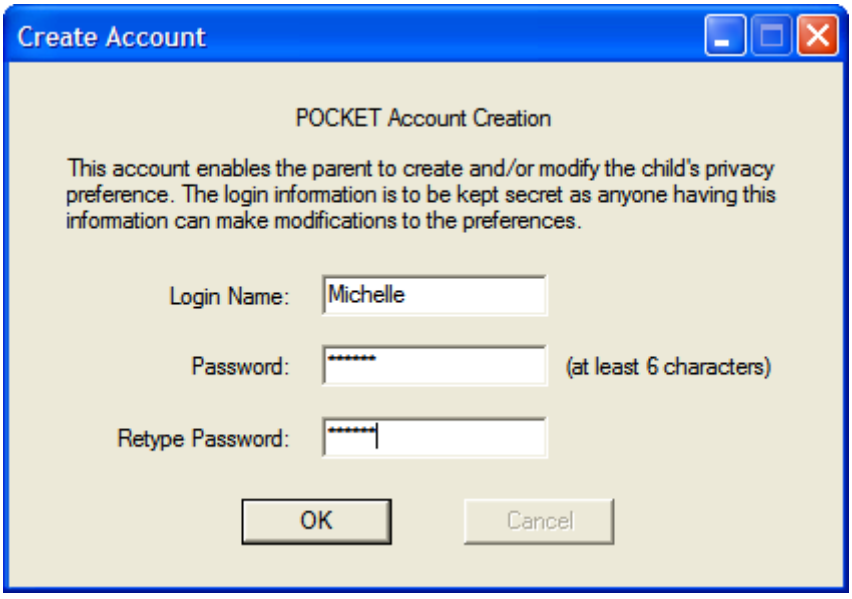

Figure A 1: POCKET registration - account creation

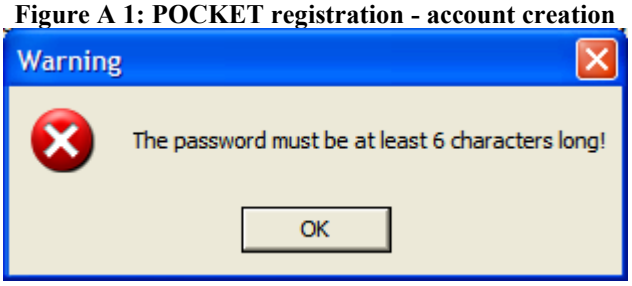

Figure A 2: Error message

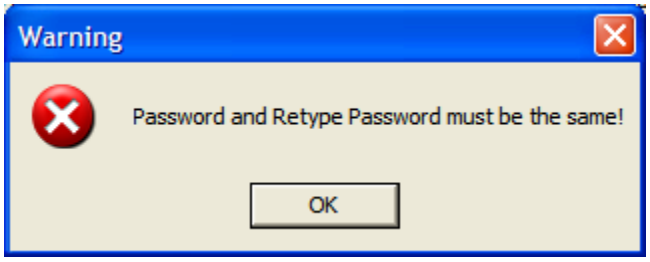

Figure A 3: Password consistency check

This test executes 4 paths in the AccountCreationDlg.cpp class.

The setup phase involves selecting the preferences for the child. POCKET shows a preference file creation dialog box using which parents can choose approved information. Some of these options have tooltip functions and they are also tested by placing the mouse on them and displaying the tooltip. This is shown in Figure A5. Different combinations of the preference options are selected. One such combination is shown on Figure A4. However, there are some mutually exclusive

options such as ‘Only First Name’, ‘Only Last Name’ and ‘Full Name’. If ‘Only Last Name’ option is selected, other two conflicting options must be disabled to avoid confusion of data approval. Tests are performed to check these options. Also, the child’s name cannot be left blank. This is also tested.

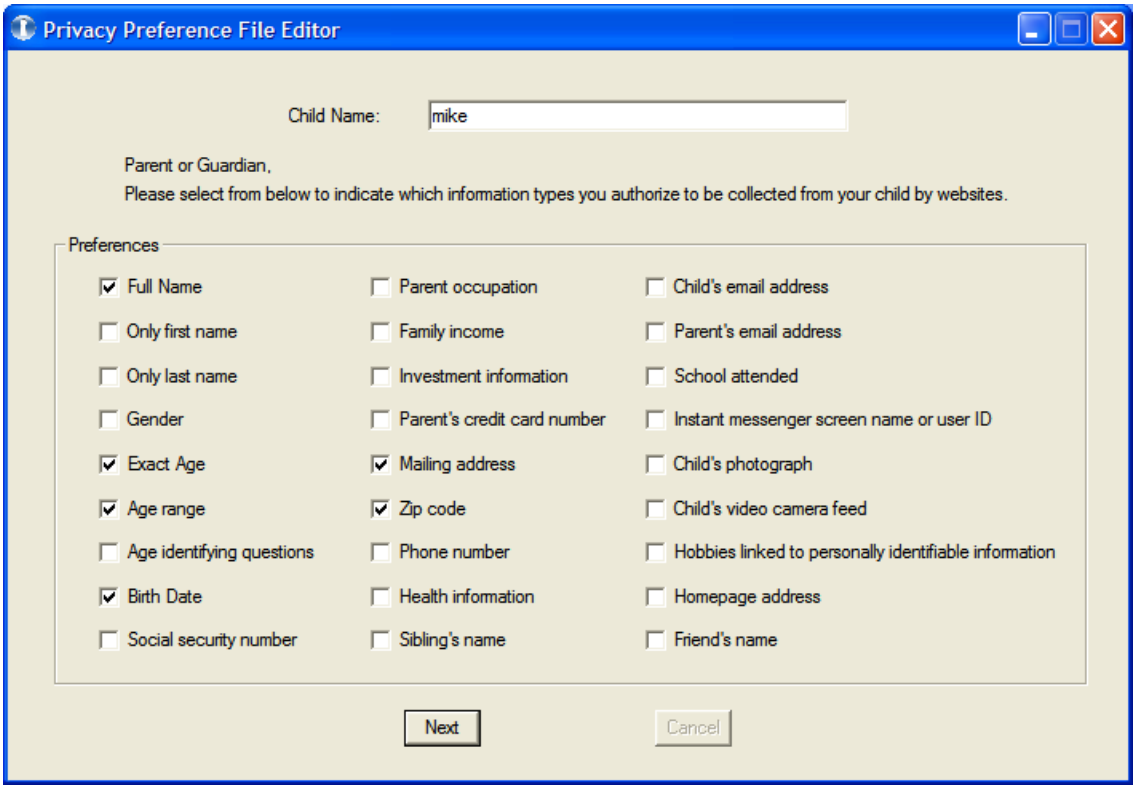

Figure A 4: Privacy preference editor form

Once the preferences are selected, the next dialog box opens up with chosen fields enabled and blank. The parent should enter data in these blanks. The checks of data fields are specified above and are applied to all the data fields in this system. Some checks that are done in this form are:

- Data validity – blanks that require number-only inputs must throw an error for alphabet-input. Vice versa is also true.
- Data consistency – fields that contain relative information are checked for consistency so that their data corroborates with each other.
- Data completion – all the approved data fields have to be filled in. Any unfilled blanks must throw an error and the system must not proceed to the next step until the errors are corrected.

These checks are done and their snapshots are shown in Figure A6, Figure A7 and Figure A8.

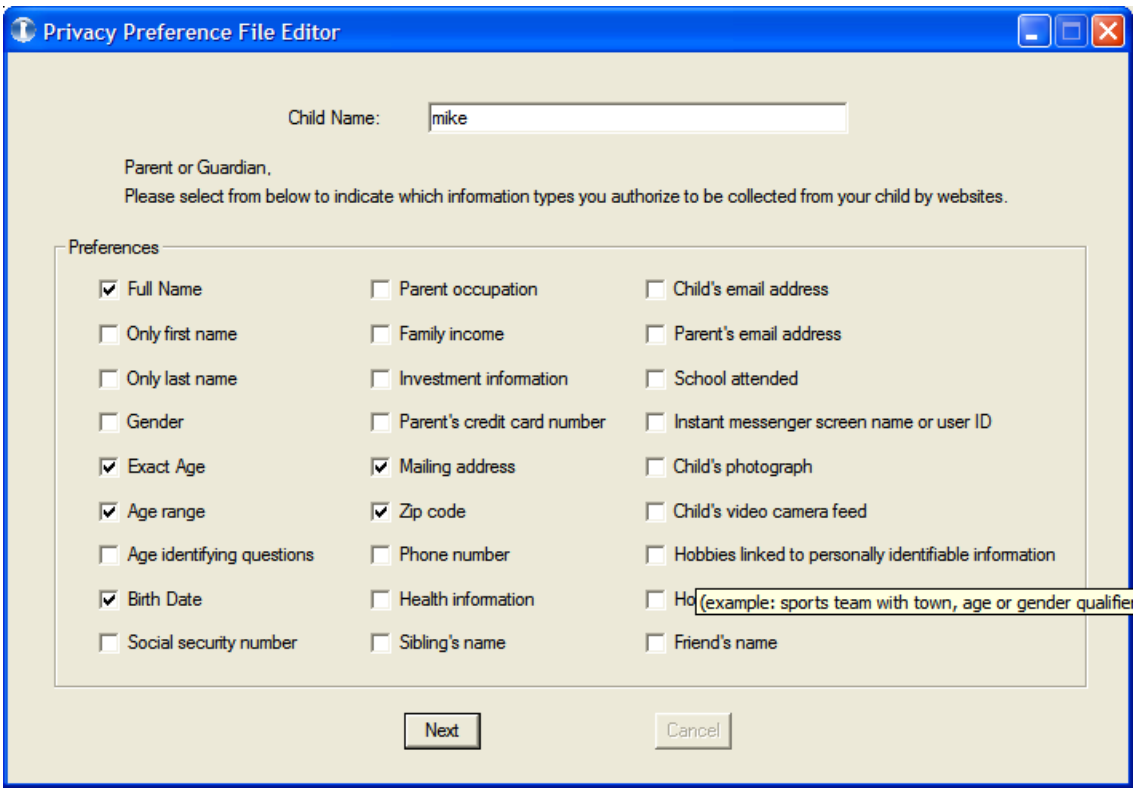

Figure A 5: Testing of tooltip

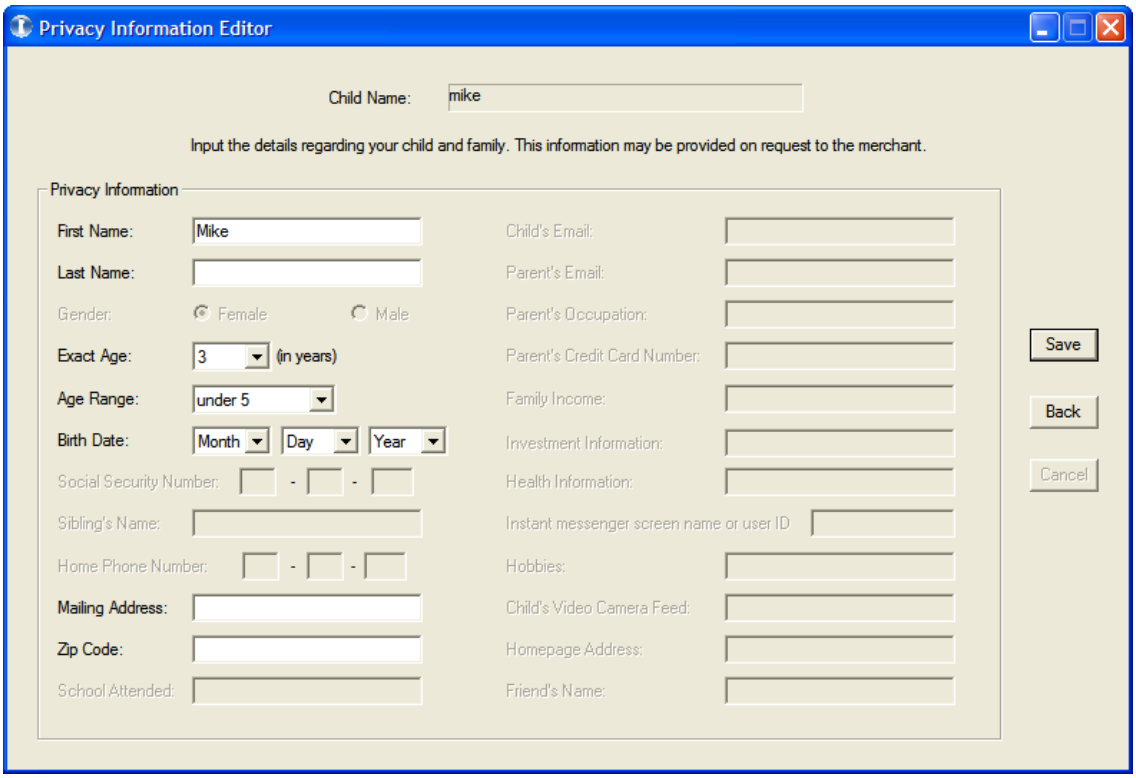

Figure A 6: Dialog box to enter information

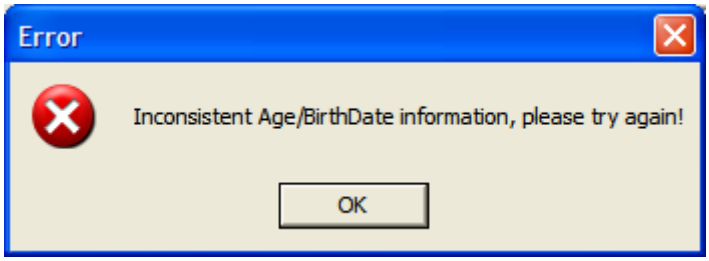

Figure A 7: Information consistency check

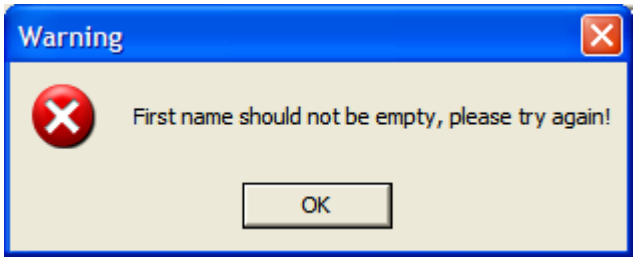

Figure A 8: Black field error message

After the preference file is created and saved (confirmation shown in Figure A12), POCKET installs the BHO plug in. This installation requires that all current IE windows are closed. The BHO is used for comparison of UPPF and MPPF. This step includes three choices. Applying each of them, we obtain Figure A9, Figure A10, and Figure A11.

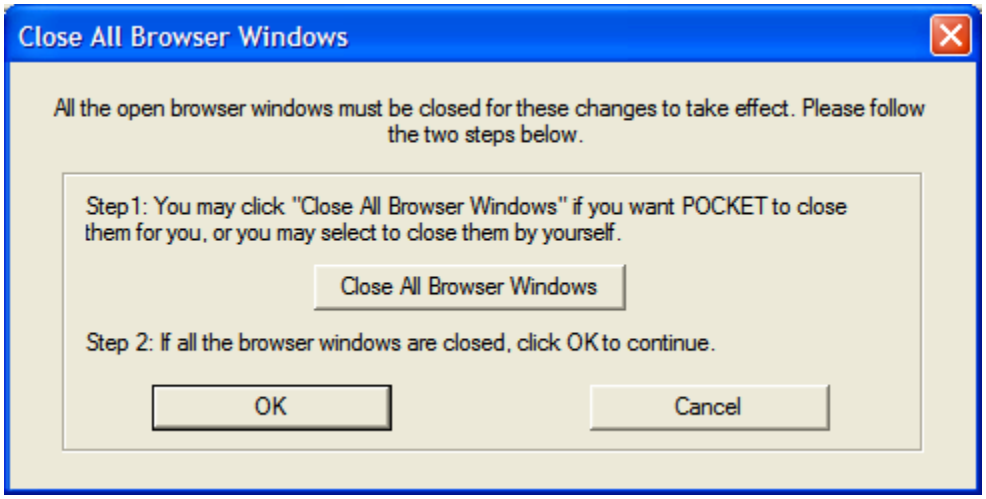

Figure A 9: Enable plugin message

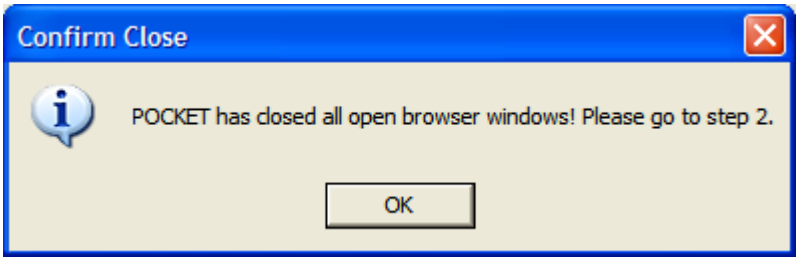

Figure A 10: Closure confirmation message

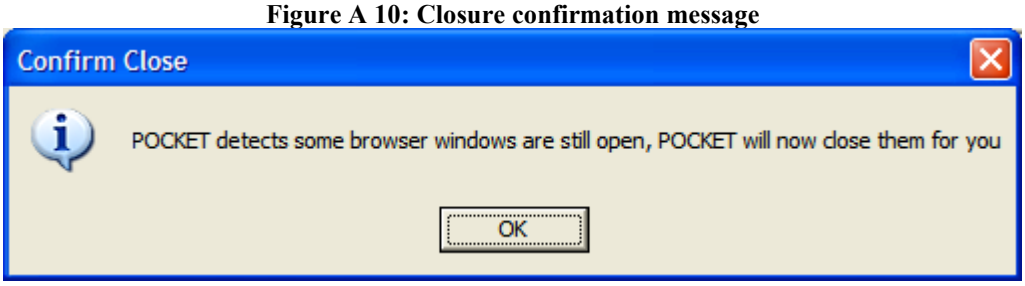

Figure A 11: Close confirmation error message

.

Figure A 12: Preference files have been successfully saved

POCKET, when enabled, shows the dialog box shown in Figure A13. Minimizing POCKET moves it to the System Tray. This feature is tested and shown in Figure A14. The options shown in that dialog box are tested. The Account menu has two options, 'Change password' and 'Exit' (shown in Figure A15). 'Change Password' brings up a dialog box that requests for authentication (Figure A25) followed by password change dialog box (Figure A16). If the password change is successful, it is shown by confirmation message (Figure A17). If it is unsuccessful, it is shown by an error message (Figure A18). Pressing 'Exit' (Figure A19) takes POCKET to authentication dialog box. Once authenticated and confirmed (Figure A20), POCKET shuts down.

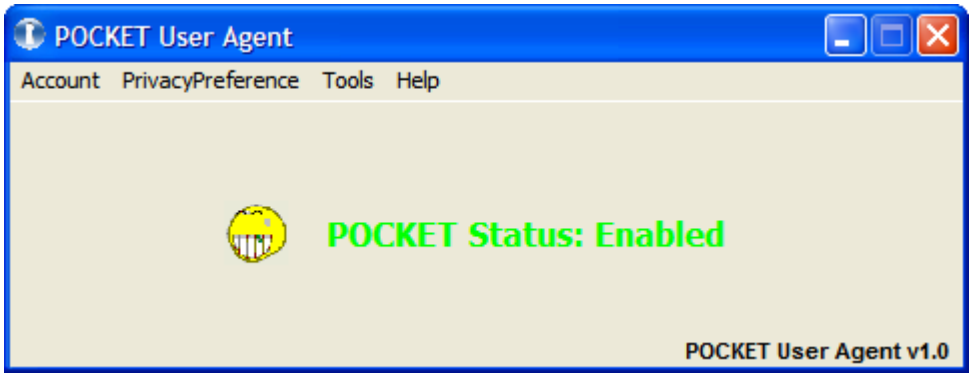

Figure A 13: POCKET Enabled status

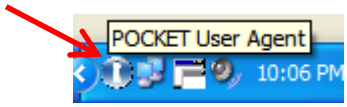

Figure A 14: Minimized POCKET in System tray

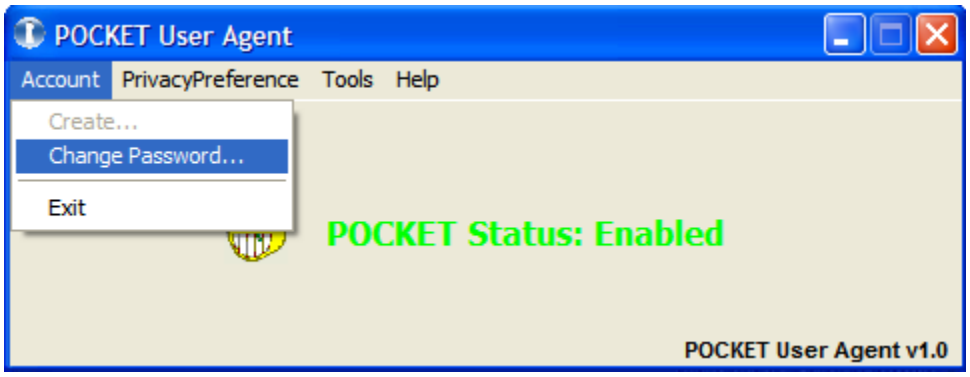

Figure A 15: Account Menu options

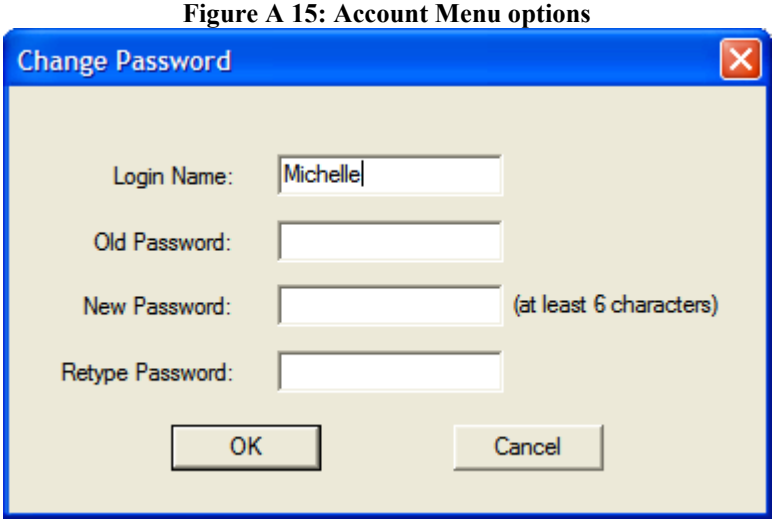

Figure A 16: Password change form

Figure A 17: Successful password change message

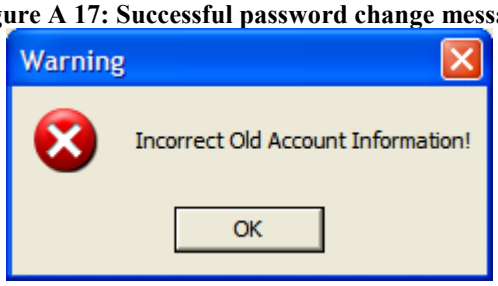

Figure A 18: Unsuccessful password change message

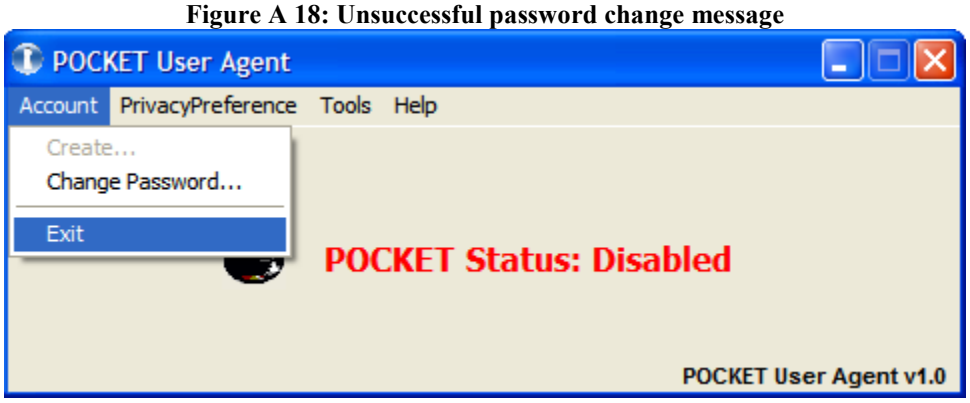

Figure A 19: Selection of Exit option

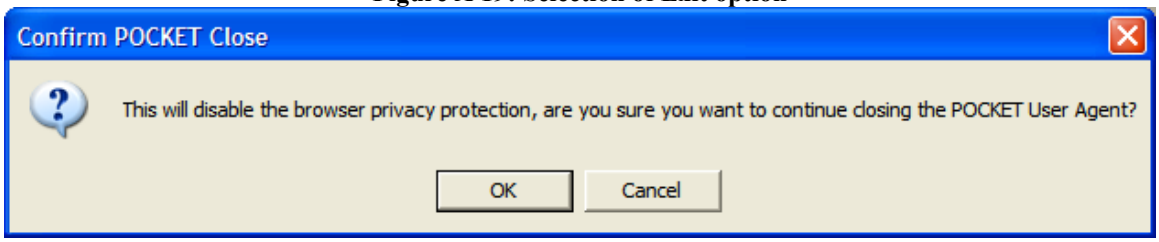

Figure A 20: Confirmation to disable POCKET protection

The 'PrivacyPreference' option is chosen (Figure A21) and the Update option is selected. This leads to the authentication dialog box (Figure A25). On authentication, the preference file search box is displayed (Figure A22). Both available and unavailable preference files are searched. Available files load into the system and open up as preference selection dialog boxes (Figure A5). Any changes are saved. Unavailable files throw up an error (Figure A23).

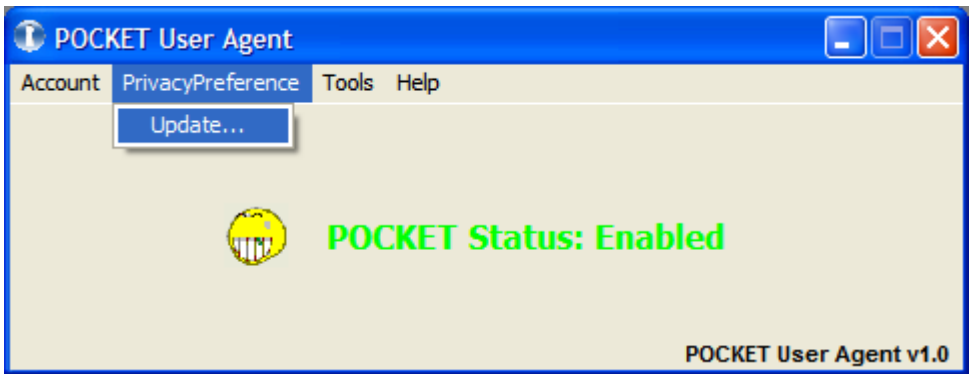

Figure A 21: Privacy preference update option

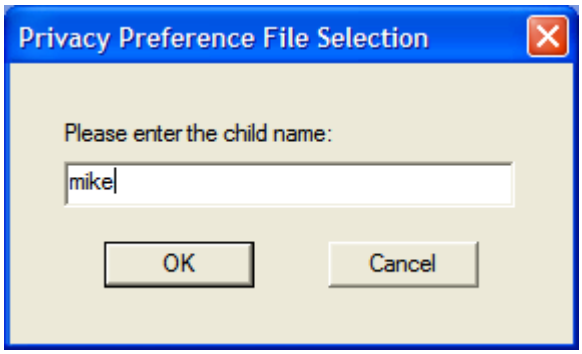

Figure A 22: Privacy preference file search dialog box

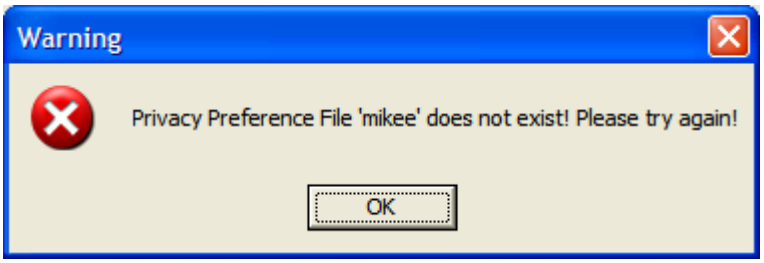

Figure A 23: Unavailable file message

'Tools' option consists of an option to enable/disable the BHO Plug in of POCKET (Figure A24). This operation requires authentication (Figure A25). If the authentication succeeds, the disabled confirmation message pops up (Figure A27). The status changes to Disabled and is shown on the dialog box as Figure A28. If the authentication fails, an error message is displayed (Figure A26). To enable POCKET again, authentication is required. Successful enabling of POCKET results in a message as shown in Figure A29.

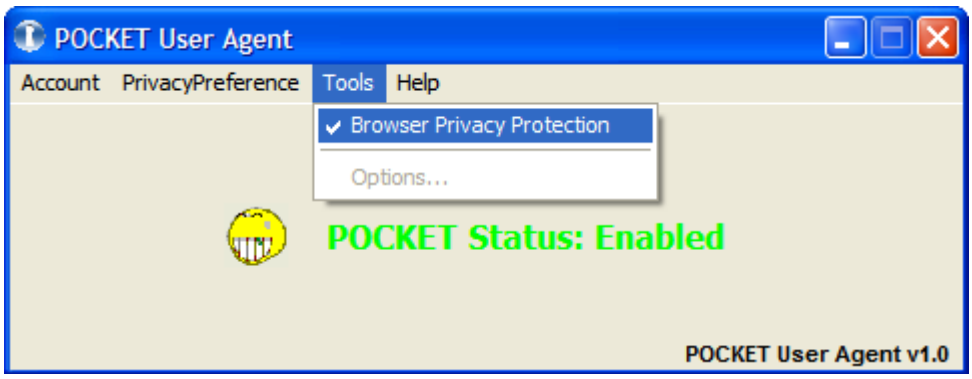

Figure A 24: Options in Tools menu

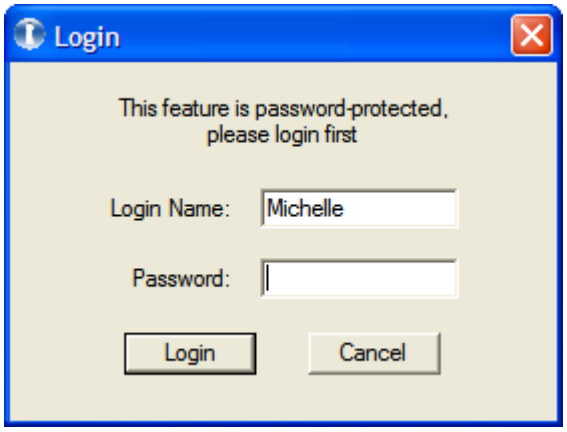

Figure A 25: Authentication window

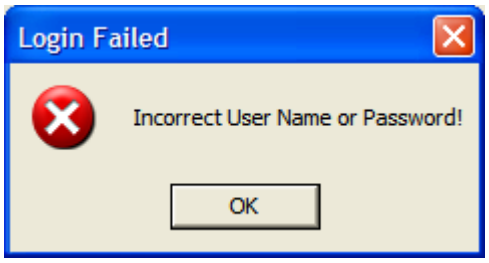

Figure A 26: Authenticatio failure message

Figure A 27: Disable successful message

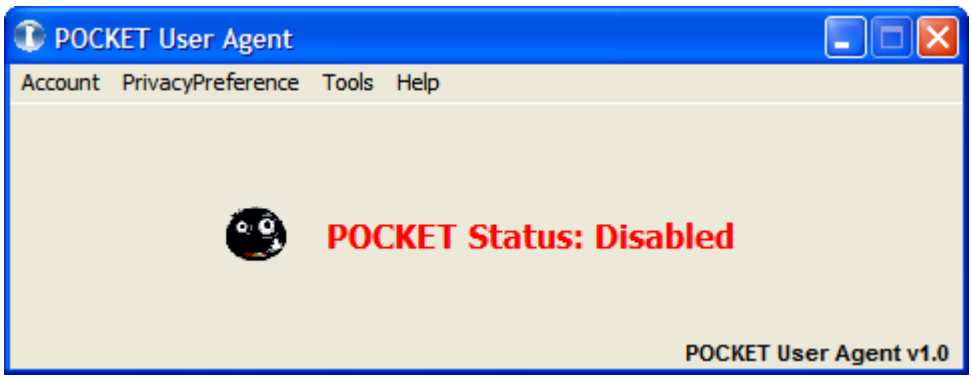

Figure A 28: Disabled POCKET status

.

Figure A 29: Enable successful message

This is followed by testing the last menu: Help. Here, the About option is selected (Figure A30) and it displays the About dialog box (Figure A31).

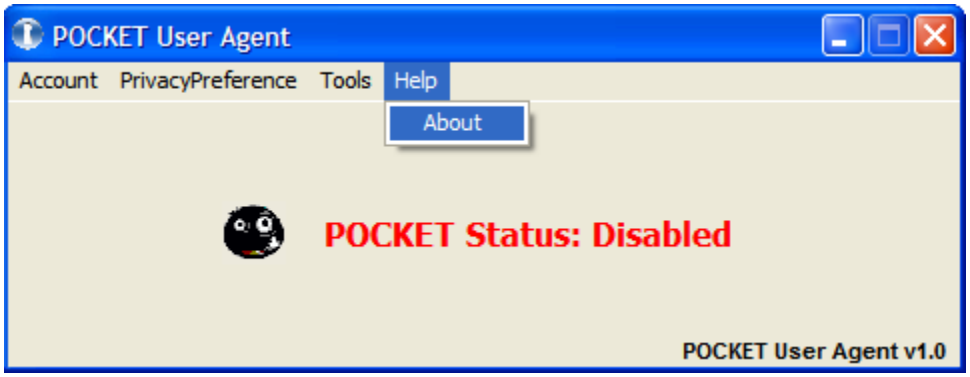

Figure A 30: 'About' option selection

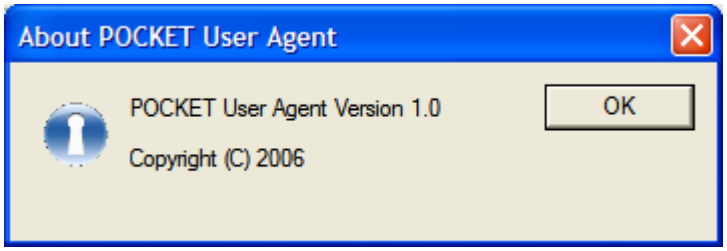

Figure A 31: Information box of POCKET

Once all the options of POCKET dialog box are verified against expected outputs, the options provided by POCKET in the System tray are tested. The system tray (Figure A32) contains all the options that were present in the POCKET dialog box. It also includes the 'restore' option which restores the POCKET application window to normal. Since the options are same (Figure A33), they are invoked from the system tray and checked again for their functionality.

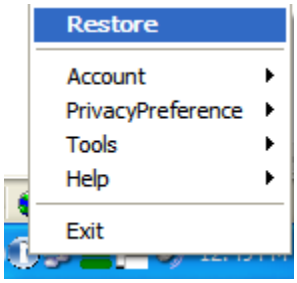

Figure A 32: POCKET options from System tray

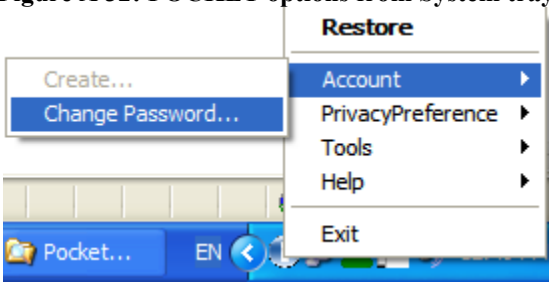

Figure A 33: Invoking menu options from System Tray

From this walkthrough, we understand that the functions in POCKET have been extensively tested. It requires authentication at each step and discourages hacking. However, if the log in information is stolen or lost, then POCKET security becomes ineffective. This result is reiterated in Chapter 3.

Appendix B

Software Requirements Specification (SRS) of User Re-authentication System

The software requirement specification document establishes the basis for agreement between customers and contractors or suppliers on what the software product is expected to do, as well as what it is not expected to do. Some of the features of SRS are

- It sets a rigorous assessment of requirements before design can begin.
- It sets the basis for software design, test, deployment, training etc. It also sets pre-requisite for a good design though it is not enough.
- It sets the basis for software enhancement and maintenance.
- It sets the basis for project plans such as Scheduling and Estimation.

It includes the system requirements specification for completion. The system requirements specification gives the high level specification about the entire system. Some features covered by the system requirements specification are:

- **Function of system:** The main aim of this system is to periodically re-authenticate the user based on the user's behavioral biometric characteristics in a non-intrusive manner. This system focuses on the keyboard and mouse attributes of the user. Every few minutes, the system obtains the mouse and keyboard data of the user and compares it with the predetermined profile of the given username. If the system finds any anomaly beyond a preset threshold, it is duly noted and the user may be blocked from using the system. This ensures that '*current-user*' is the '*logged-in*' user.

- **Hardware/software functional partitioning:** Since this system is a purely software system with very little dependence on hardware, there is no functional partitioning between hardware and software. However, the data from the hardware (namely, keyboard and mouse) is captured by the software and is used for analysis purposes for all users.
- **Performance specification:** This system is expected to have a high accuracy as it is an authentication method. Hence, false positives and false negatives cannot be very high. Also, the threshold of allowable error can be set based on the level of security needed.
- **Safety requirements:** This system can operate in the Windows operating system environment. All the threads and processes are handled by the OS with safe usage and deletion. So, there are no specific safety requirements that have to be handled by the system. Any exceptions that may occur are captured by the system at the required points.
- **User interface:** The user interface consists of three parts: initialization, training and testing part. The initialization and training parts are conducted once when the system is set up, while the testing phase is done whenever the user uses the system. The initialization phase should collect adequate data to train the system, the training phase trains the system and the testing phase verifies the user identity.

A detailed description of the processes is given in Chapter 4. The SRS specifications concentrate on:

- Functionality – the software's intended operations
- External interfaces – the interfaces of software with hardware, users, etc.
- Performance – various performance metrics to be considered such as speed, availability, response time, recovery time of various software functions, etc
- Attributes – definition of attributes of the software such as portability, correctness, maintainability, security, etc
- Design constraints imposed on an implementation – prerequisite conditions, standards to be followed, implementation language constraints, policies for database integrity, resource limits, operating environment, etc.

The specifications for each function were defined based on its expected functionality. Test cases are then written to test each of the specifications of the function. Some of the assumptions made while the test specifications were written are:

1. Most of the main classes (starting points of program execution in each process) are inherited from System.Windows.Forms namespace which is a pre-existing library in .NET framework. Hence, the inherited properties of these classes are assumed to be tested.

2. Further, the constructor and destructor functions of these classes are tested with black box testing during program execution as these functions cannot be directly called by a test case in an external test environment.
3. Every class contains a set of functions with different functionalities. The functions in a class use the class's member variables to interact with each other. So they do not have any particular external interface. The external interface of the class remains the same for all functions in it.
4. The performance metrics, software attributes and design constraints are common to all the classes in a process. Hence, it is sufficient if they are discussed for all the classes of a process collectively rather than being discussed for every function in every class of a process.
5. All the processes work in the same environment and are dependent on each other for correct operation. So, they have the same performance metrics, software attributes and design constraints unless otherwise stated.

Stage 1: Recorder.exe

This process consists of four classes, the main class and three supporting classes, in the Recorder namespace.

- public partial class Form1: Form
  - public Form1()
    - It initializes the form by calling initializeComponent() and sets the initial values of the member variables. It does not take any arguments nor does it return any arguments.
  - private void Form1_Load(object sender, EventArgs e)
    - It is a user modified version of the Form_Load() function. Here, the inputListener class is instantiated and the mouse and keyboard handles are added to the default list. It also activates the worker thread on inputListener class. It takes two arguments, a pointer to object calling it and the event handle that called it.
    - private void InputListener_KeyPress(object sender, KeyPressEventArgs e)
    - It obtains the data from the key buffer, extracts the required data and writes it to the output string.
  - private void InputListener_MouseMove(object sender, MouseMoveEventArgs e)
    - It obtains the data from the mouse buffer, converts the required data into analyzable form and writes it to the output string.
  - private void Form1_FormClosed(object sender, FormClosedEventArgs e)

        - It is a user modified version of the Form_FormClosed() function. It writes the data in the output string to the tester.txt file, ends the thread execution in inputListener class and safely exits the process.

- This class contains all the interfaces of the Form class that it inherits. It connects to classes in other namespaces through these interfaces. It has interface links to the inputListener class in the same namespace.

- public class InputListener
    - public void Run()
        - It creates a new thread and starts it.
    - public void End()
        - It aborts the thread safely before the entire process exits. This makes sure that there are no uncontrolled threads running in the system hogging the resources of the system.
    - private void RunThread()
        - This function is called by the thread created in Run(). It obtains the mouse and keyboard data from the handlers and sends it to the main class. The advantage of having a thread collect the mouse and keyboard data independent of the main class is that the thread does not miss any mouse or keyboard data. Since the entire application depends on the accuracy of the mouse and keyboard data collected, it becomes important that this is true.

- This class is connected to the main class, MouseMoveEventArgs class and KeyPressEventArgs class. It acts as the connecting point between these classes.

- public class MouseMoveEventArgs:EventArgs
    - public MouseMoveEventArgs(int X, int Y)
        - this class is inherited from EventArgs class and just captures the mouse data from the actual mouse event handler.

- This class is connected to the InputListener class

- public class KeyPressEventArgs: EventArgs
    - public KeyPressEventArgs(Keys ModifierKeys, int KeyCode)
        - this class is inherited from EventArgs class and just captures the keyboard data from the actual keyboard event handler.

- This class is connected to the InputListener class

Stage 2:

Profile collector.exe

This process consists of four classes, the main class and the three forms used for initial training phase.

- public partial class Form1: Form
  - public Form1()
    - It initializes the class variables.
  - private void button_try_click(object sender, EventArgs e)
    - It checks if all the forms have been completed. This function initializes other classes and executes them.

- This class contains all the interfaces of the Form class that it inherits. It connects to classes in other namespaces through these interfaces. It also has interface links to Form2 class, Form3 class and Form4 class

- public partial class Form2:Form
  - public Form2()
    - It initializes the class variables.
  - private void Form2_Load(object sender, EventArgs e)
    - It sets the focus of the form.
  - private void button1_Click(object sender, EventArgs e)
    - It checks if the output of the form satisfies the requirements.

- This class contains all the interfaces of the Form class that it inherits. It connects to classes in other namespaces through these interfaces. It also has interface links to the Form1 class

- public partial class Form3: Form
  - public Form3()
    - It initializes the class variables.
  - private void b_1_Click(object sender, EventArgs e)
    - It checks if the output of the form satisfies the requirements.
  - private void b_2_Click(object sender, EventArgs e)
    - It checks if the output of the form satisfies the requirements.
  - private void b_3_Click(object sender, EventArgs e)
    - It checks if the output of the form satisfies the requirements.
  - private void b_4_Click(object sender, EventArgs e)
    - It checks if the output of the form satisfies the requirements.
  - private void b_5_Click(object sender, EventArgs e)
    - It checks if the output of the form satisfies the requirements.

- private void b_6_Click(object sender, EventArgs e)
  - It checks if the output of the form satisfies the requirements.
- private void b_done2_Click(object sender, EventArgs e)
  - It checks if the output of the form satisfies the requirements.

- This class contains all the interfaces of the Form class that it inherits. It connects to classes in other namespaces through these interfaces. It also has interface links to the Form1 class

- public partial class Form4:Form
  - public Form4()
    - It initializes the class variables.
  - private void button1_Click(object sender, EventArgs e)
    - It checks if the output of the form satisfies the requirements.

- This class contains all the interfaces of the Form class that it inherits. It connects to classes in other namespaces through these interfaces. It also has interface links to the Form1 class

Profile analyser.exe

This process consists of only one main class and a user defined structure.

- class Program
  - static void Main(string[] args)
    - It obtains the data generated by the profile collector in the tester.txt and uses the golden reference file "mas1.txt" to analyze the data for each observation. The result is stored in Mprofile.txt for each observation instance.
- This class is a console application class and links to the profile system process.

Tester.exe

This process consists of only one main class and a user defined structure

- class Program
  - static void Main(string[] args)
    - It obtains the data generated by the recorder and uses the golden reference file "mas1.txt" to analyze the data for each observation. The result is stored in user_profile.txt for each observation instance.

- This class is a console application class and links to the full system process.

Stage 3:

Profile system.exe

This process consists of a main class and a user defined structure.

- public partial class Form1: Form
    - public Form1()
        - It initializes the class variables.
    - private void b_ok_Click(object sender, EventArgs e)
        - It gets the number of users for the system, checks if it is a valid number and performs the training for all the users.
    - private void caller()
        - It obtains the data generated by recorder and uses mas1.txt to analyse the data. The result is stored in Mprofile.txt.

- This class contains all the interfaces of the Form class that it inherits. It connects to classes in other namespaces through these interfaces. It is also linked to the profile collector process, profile analyzer process, the recorder process and the full system process.

Stage 4:

Profiler.exe

This process consists of one main class and two user defined structures.

- class Program
    - static void Main(string[] args)
        - It calls the profile system process and data analyser function.
    - private static void data_analyser()
        - It reads the Mprofile.txt file and creates the profile for each user. The result is stored in the profiler.txt. Further, data in Mprofile.txt is written to nprofiler.txt in a form suitable for back propagation algorithm used later during comparison.

- this process is a console application and connects to the full system process.

Stage 5:

Full system.exe

This process consists of a main class and two user defined structures.

- public partial class Form1: Form
    - public Form1()
        - It initializes the class variables and does the initialization and training of the system by calling the profiler process. It also does directory management and securitization of the files.
    - private void b_ok_Click(object sender, EventArgs e)

        - It gets the username of the person and performs initial checks before testing begins. It also does the testing phase of the system and analyses the given results.
    - private int testingga(int user_return)
        - It obtains the user diagonised by the correlation method and sees if the value returned by the neural networks is the same. If so, it is returned, else any value obtained by the neural network analysis is returned.
    - private void b_cancel_Click(object sender, EventArgs e)
        - It returns the control to user to enter the username again.
- class Program1
    - void preinit()
        - It does the pre-initialization operations for back propagation algorithm using the training data. Also the Mersenne Twister random number generator is created here.
    - public void Run()
        - It executes the back propagation network and trains the network for the collected data.
    - public int testing()
        - The input from the present user is fed to the network and a result is obtained based on previous training. This value is passed to the calling function to compare the users.
    - void activate(Neuron n)
        - It activates the neuron and gives the output based on the inputs to it.
    - void changeWeightsAndError(Neuron n)
        - It changes the neuron's input weights and computes the error for each of the inputs of the neuron. This function, along with the previous function, back propagates the error from the output to input layer and adjusts the network to be more related to the data set given.
    - void init()
        - It creates the network containing input, hidden and output layers of neurons each connected with one another. Also, the values of each of the neurons are initialized.
    - class Neuron
        - It contains the properties of a neuron in the network. It does not have functions of its own.

- class Synapse
  - It contains the properties of the synapse that connect to the said neuron. It contains only a constructor function.
- static void mertinit(int seed)
  - It initializes the Mersenne Twister random generator.
- static long mert()
  - It generates the mersenne number when called.
- static void generateNumbers()
  - It generates a new set of Mersenne array numbers.
- static long extractNumber(int i)
  - It extracts a tempered pseudorandom based on the counter value.

- class Program2
  - public void correldatareader()
    - It initializes the class variables and obtains the data from profiler.txt for analysis.
  - public int chkuser(string username)
    - It checks if the given username is among the list of valid usernames given during training phase of the system.
  - public int testing()
    - It gets the present user's inputs and tests them with the results of the analysis from the profiler.txt. Returns a value to the calling function to compare the users.

  - this class contains all the interfaces of the Form class that it inherits. It connects to classes in other namespaces through these interfaces. It is also linked to the profile system and tester processes.

These processes should ensure that the system has a very low false positive or false negative and the level of accuracy is high. Since recorder.exe works with threads, it must be made sure that the threads are handled correctly. The entire system can work only in a Windows-based environment. However, this is a justified constraint since this application is developed for POCKET which is a Windows-based product.

Appendix C

Black Box Testing Sample for User Re-authentication system

This section gives a sample walkthrough of black box testing procedure for the user re-authentication software. It explains the steps of user re-authentication system with snapshots. For technical details and implementation, refer to Chapter 4 of this thesis.

The user re-authentication system consists of three phases: initialization, training and testing. The initialization phase is made up of six forms. Each of these six forms requires the user to do different tasks. The main form is shown in Figure C1. When a user is added, the 'Add user!' button must be clicked. This moves the control to Training phase form (Figure C2).

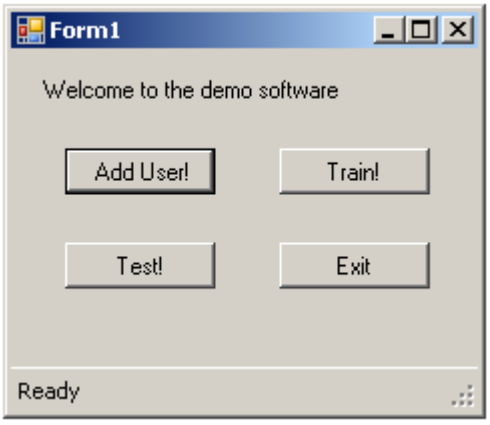

Figure C 1: Main form

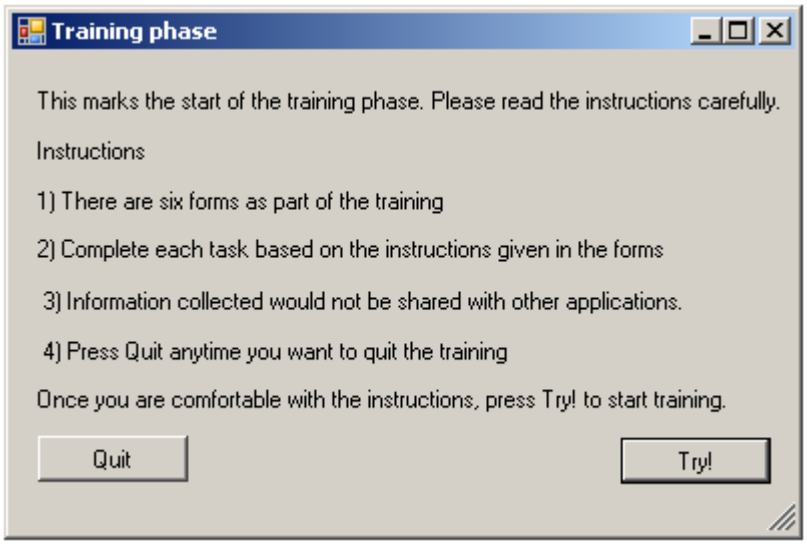

Figure C 2: Training phase

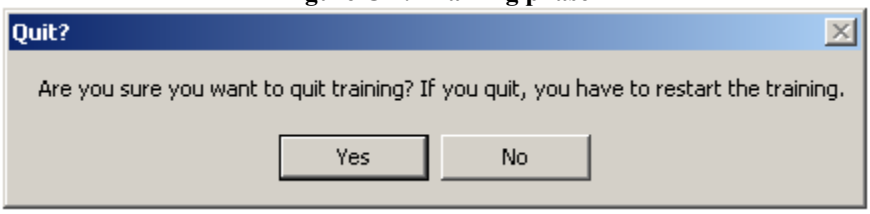

Figure C 3: Quit training confirmation prompt

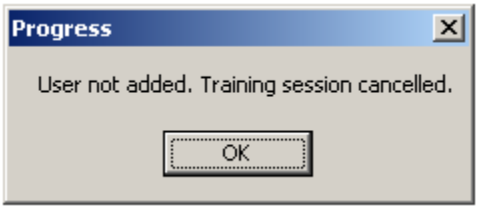

Figure C 4: User quit form

The training form contains the instructions to be followed for training. The user can quit training by pressing 'Quit' button any time. This leads to a prompt message asking for confirmation (Figure C3). If confirmed, it gives a 'end of training' message (Figure C4) and returns to the main form. The 'Quit' option is present in every step of training.

The training begins with Form 1 shown in Figure C5. A username with only alphabets is entered. Any numbers or a blank username leads to error messages (Figure C6). The sentence has to be input exactly. Any change or blank form results in error messages (Figure C7). If the sentence and username are input correctly, a congratulatory message is shown (Figure C8).

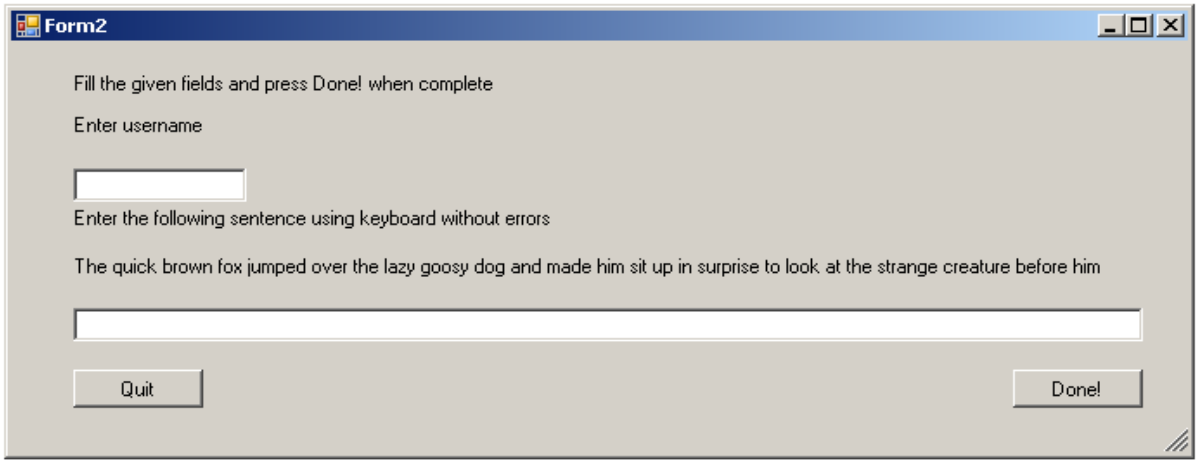

Figure C 5: Form 1

Figure C 6: Error message

Figure C 7: Error message for sentence

Figure C 8: Congratulatory message

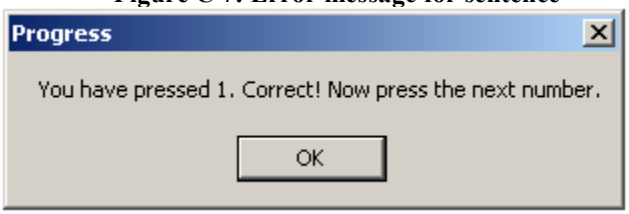

Figure C 9: Congratulatory message

The second form, Form 2, shown in Figure C11, requires the user to press the buttons in numerical order. A correct mouse click gives a congratulatory message (Figure C9) while a wrong mouse click gives a warning message (Figure C10).

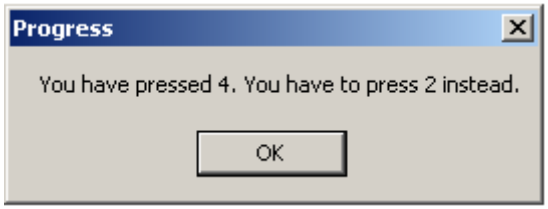

Figure C 10: Warning message

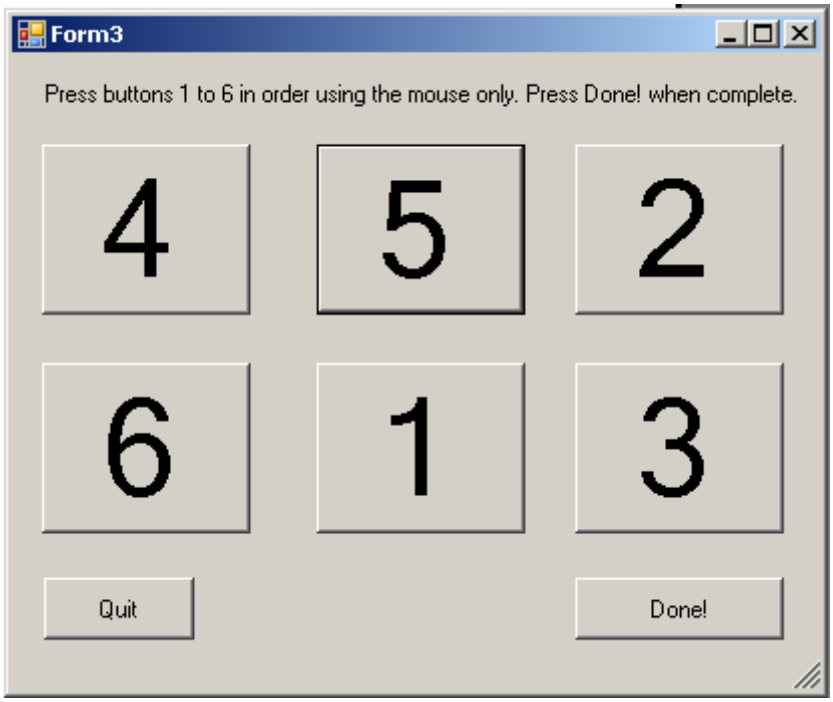

Figure C 11: Form 2

The third form is Form 3, shown in Figure C12. Two radio buttons should be chosen and their names should be entered into the text box. Various tests are done in this form such as blank fields, incorrect fields, one incorrect field, etc. All of them give a warning message (Figure C14). Only the correct input gives the congratulatory message (Figure C13).

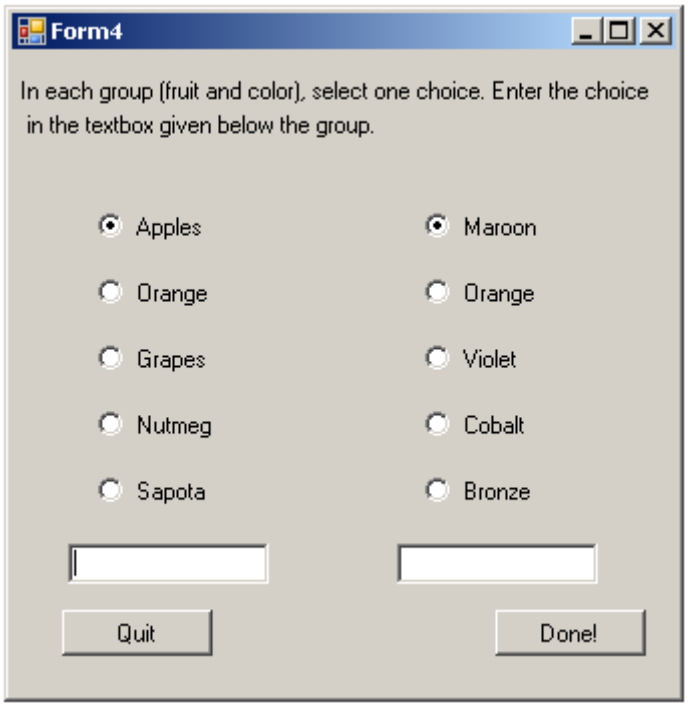

Figure C 12: Form 3

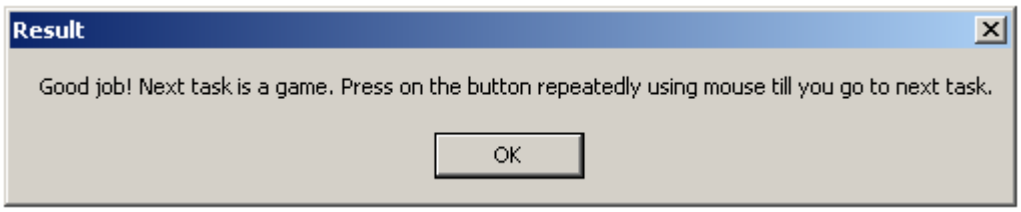

Figure C 13: Congratulatory message

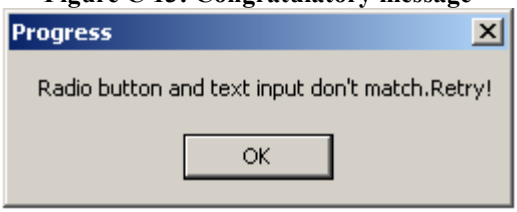

Figure C 14: Warning message

Form 4 is a representation of a simple game (Figure C15). The square moves to random locations and after 5 such clicks, the control moves to the next form. An introductory message is displayed (Figure C16). Form 5 consists of a picture (Figure C17) and requires that the user must enter at least 75 characters describing it. Blank and minimum character checks are done in this field (Figure C18 and Figure C19). This is followed by Form 6 that also has a picture (Figure C21) and requires at least 75 characters to be input as description (Figure C20). Form 6 also has colored buttons that need to be clicked in a particular order (Figure C22). At the end, a congratulatory message is shown (Figure C23). This ends the initial training phase for a user. This is repeated for all users. If the username already exists, the system requests for a different username (Figure C24). Then the system is trained.

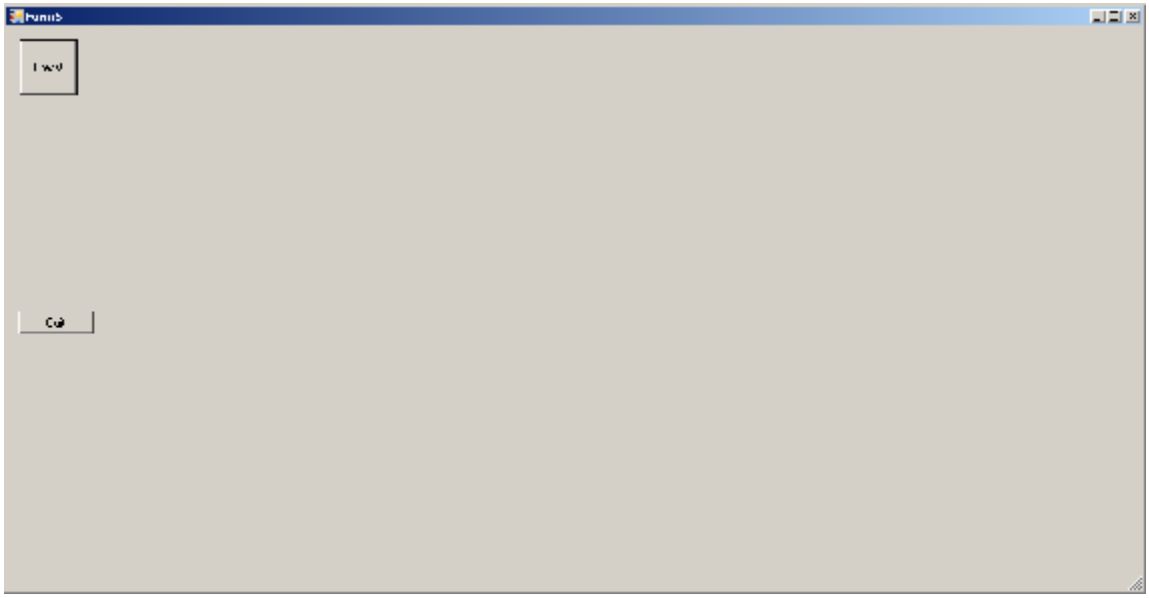

Figure C 15: Form 4

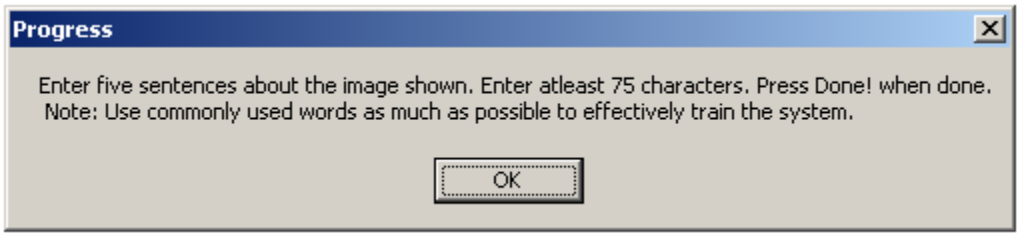

Figure C 16: Instruction message

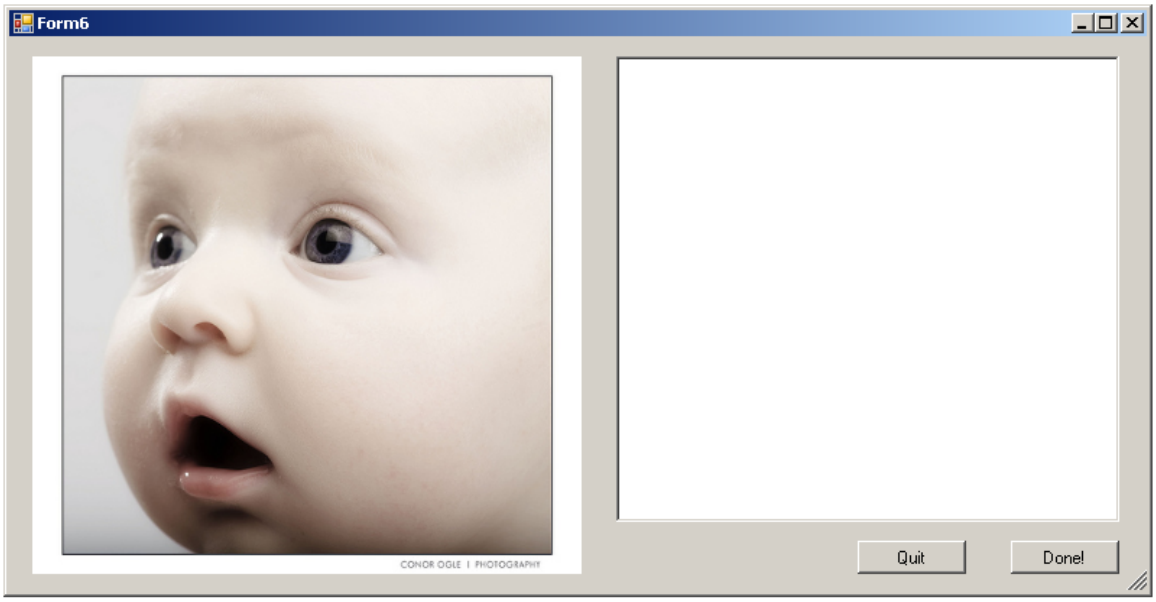

Figure C 17: Form 5

Figure C 18: Warning message

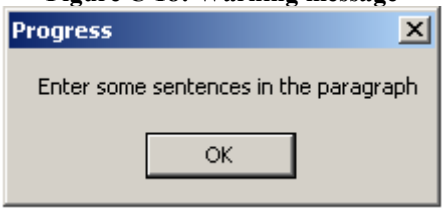

Figure C 19: Warning message

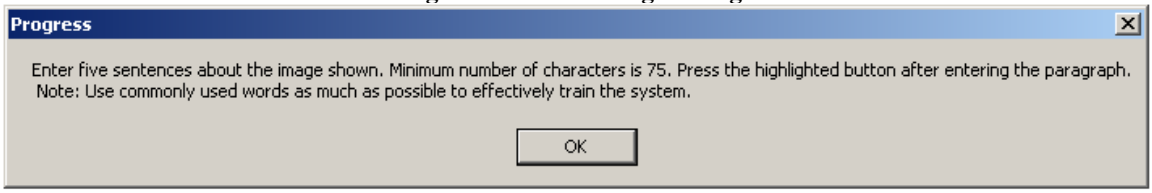

Figure C 20: Introductory message

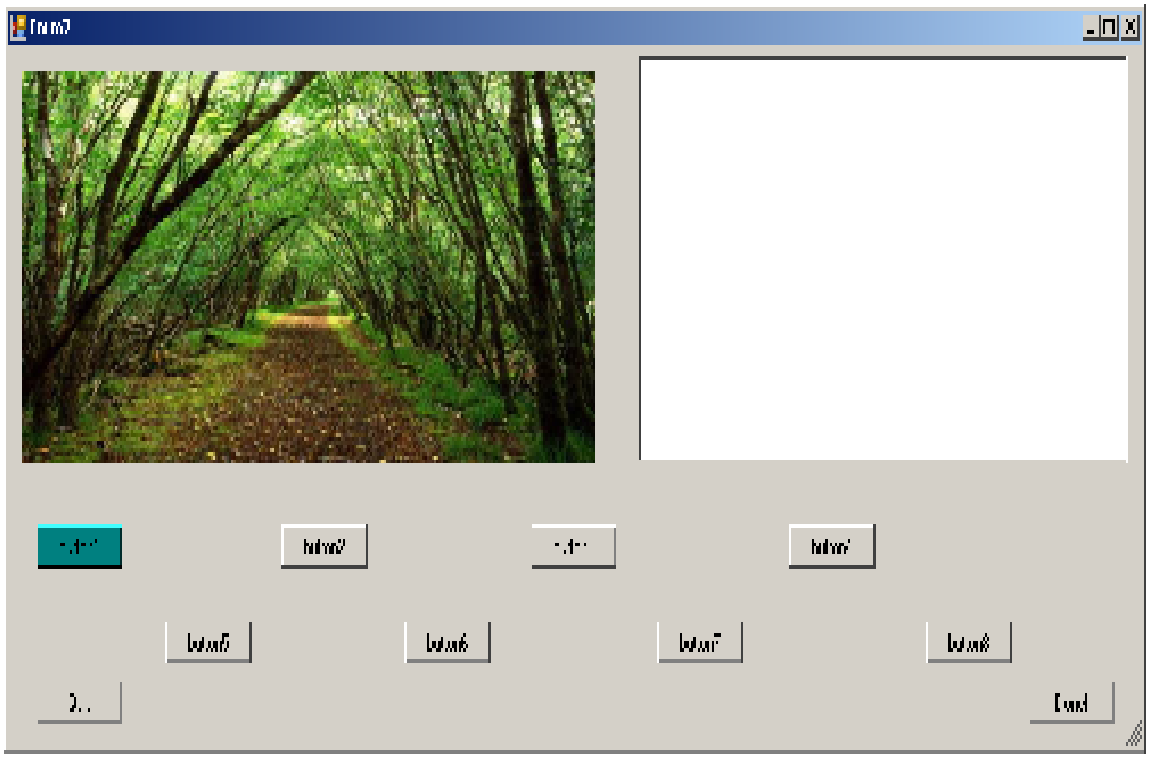

Figure C 21: Form 6

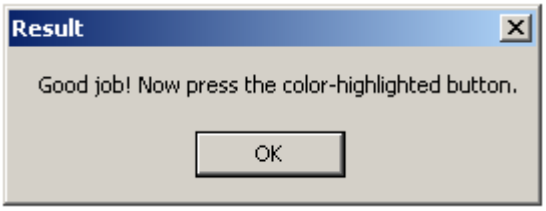

Figure C 22: Congratulatory message

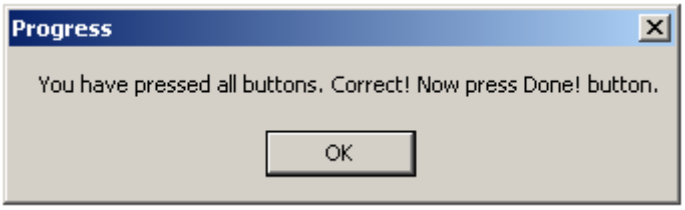

Figure C 23: End of test

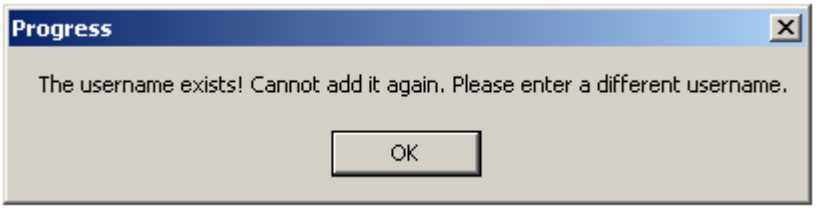

Figure C 24: Existing username warning

In the secondary training phase, the system specifies the user to be trained(Figure C25). Then the system begins training by asking for log-in information(Figure C26) and performs identification (Figure C27 and Figure C28) every minute It gets feedback from the user and continuously

retrains itself (Figure C29). This is repeated for all users. Then the system is ready for testing phase.

Figure C 25: Specifying the user for training

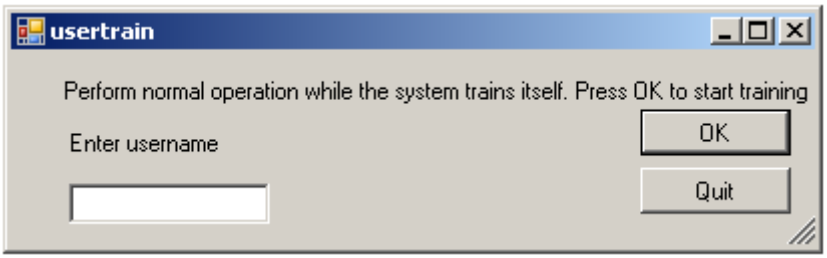

Figure C 26: Log in screen

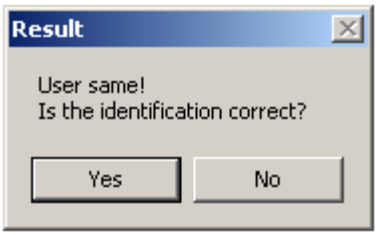

Figure C 27: Confirmation message

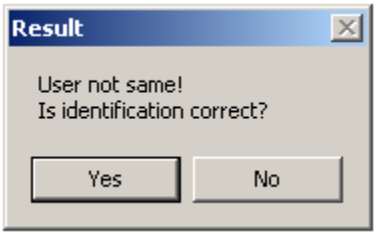

Figure C 28: Confirmation message

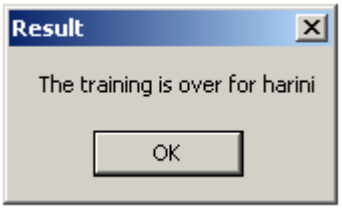

Figure C 29: End of training

In the testing phase, the system first checks for the number of users in the system. If there is only one user, there is no need for identification. So, the system requires two or more users (Figure C30). After this check is done, the system uses a log in screen (Figure C31). It gives a confirmatory prompt before loading the profile from the database (Figure C32). If the system

cannot find a corresponding trained username, then it issues a warning message (Figure C33). The information in the log in screen is compared with the trained profile. If it passes the initial authentication, then the user is subjected to continual re-authentication. If the system sees an anomaly in the behavior, it is flagged. If it is repeated beyond a threshold, the system warns the user and logs him out. After every message, the user has the option to quit from the testing phase (Figure C34 & Figure C35). If she quits, a confirmatory message is shown (Figure C36). This marks the end of testing phase.

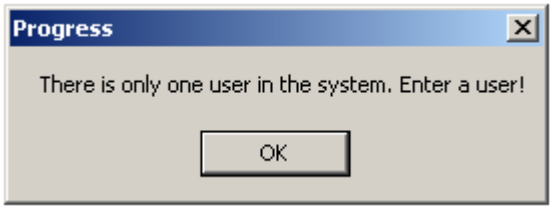

Figure C 30: User count check

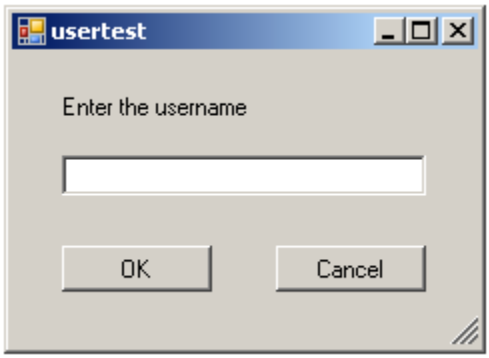

Figure C 31: Log in screen

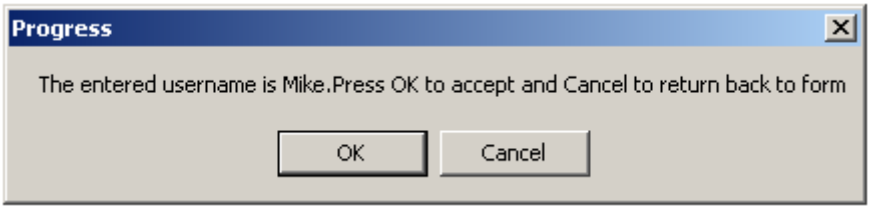

Figure C 32: Confirmation prompt

Figure C 33: Username existance check

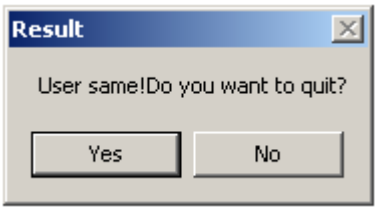

Figure C 34: Positive user identification message

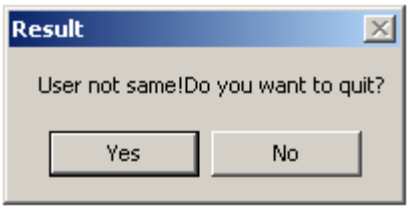

Figure C 35: Negative user identification message

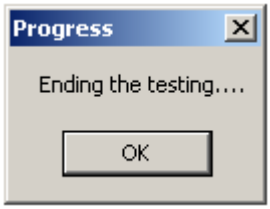

Figure C 36: End of test

This section shows a sample walkthrough of the user re-authentication system: initialization phase, training phase and testing phase. Many different paths are possible. We observe that the black box testing provides good statement coverage. However, path coverage is not satisfactory. This is due to the fact that most of the paths are present in the heuristics that are not dependent on the inputs given in the GUI. To obtain 100% path coverage, white box testing is necessary.