Cracking Passwords using Keyboard Acoustics and Language Modeling

Andrew Kelly

Master of Science
Artificial Intelligence
School of Informatics
University of Edinburgh
2010

Abstract

This project looks at how user input can be reconstructed from an audio recording of a user typing. The keyboard acoustic attack outlined in [35] is fully reimplemented and a number of extensions to their methods are developed. A novel keystroke error function is demonstrated which allows for optimal thresholds to be found when extracting keystrokes and a bagging technique is applied to previous clustering methods which increases the text recovery accuracy and removes the necessity for hand-labelled data. The properties of keystroke audio recordings are also examined and some of the limiting factors which impact on keystroke recognition are explored.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Andrew Kelly*)

Contents

1 Introduction **7**

2 Previous Attacks **9**

2.1 Keyboard acoustic emanations 9

2.2 Attacks without labelled training data 10

2.3 Inter-key timing analysis 13

2.4 Dictionary attacks using key proximity constraints 14

3 Building Datasets **15**

3.1 Data Collection 15

3.1.1 Keylogging 15

3.1.2 Audio recording 18

3.1.3 Experimental setup 19

3.2 Keystroke extraction 20

3.2.1 Using keystroke labels 20

3.2.2 Extraction error function 21

3.2.3 Unsupervised keystroke extraction 23

3.3 Problems with the data 26

3.3.1 Keystroke overlap 26

3.3.2 Phantom keystrokes 27

3.3.3 Background noise 27

4 Extracting Features **28**

4.1 Plain audio 28

4.2 FFT features 29

4.3 Cepstrum features 32

5 Keystroke Clustering **34**
5.1 Clustering methods 34
5.2 Assigning keys to clusters 37
5.3 Cluster voting 38

6 Language Model **40**
6.1 Spellcheck 40
6.2 Word Frequencies 41
6.2.1 Collecting language statistics 42
6.2.2 Extracting sentences 43
6.2.3 Second-order Viterbi algorithm 44
6.3 Model Performance 45

7 Supervised Classification **47**
7.1 Training a classifier 47
7.2 Overall attack performance 48

8 Conclusion **49**

Bibliography **50**

List of Figures

1.1 Attack overview 8

2.1 Audio signal of a single keystroke 10
2.2 Recognition rates using FFT and cepstrum features [35] 11
2.3 Keystroke Clustering HMM 12
2.4 Trigram language model with spell correction [35] 12
2.5 Categorized inter-keystroke timings [29] 13

3.1 Typing Recorder user interface 16
3.2 UK keyboard layout [34] 17
3.3 US keyboard layout [34] 17
3.4 convertKeys function 18
3.5 Spectogram of keystroke audio recording 19
3.6 Example key events in reference and extracted sequences 23
3.7 Triwave function of reference timestamps 23
3.8 Triwave function of extracted timestamps 23
3.9 Resultant error for example reference and extracted data 23
3.10 Example unsupervised keystroke detection 24
3.11 Unsupervised threshold search on dataset B 25
3.12 Unsupervised threshold search on dataset A 25
3.13 Overlapping keystrokes 27
3.14 Frequency domain of background noise 27

4.1 Classification accuracy using plain audio features 29
4.2 Scatterplot of FFT features 1 and 16 (out of 50) for dataset A 30
4.3 Performance evaluation on varying the number of FFT bins using datasets A and B 31
4.4 Frequency domain of keys in dataset B (Logitech keyboard) 31

4.5 Frequency domain of keys in dataset A (Dell keyboard) 31
4.6 Frequency domain of keys in dataset D (Logitech keyboard) 32
4.7 Frequency domain of keys in dataset C (Dell keyboard) 32
4.8 Supervised classifier accuracy over different feature sets 33

5.1 K-means cluster variance for different numbers of clusters (dataset A) 36
5.2 Gaussian Mixture cluster variance for different numbers of clusters (dataset A) 36
5.3 Cumulative frequency of keys (dataset A) 36
5.4 Normalised Mutual Information for varying numbers of clusters with K-means (dataset A) 36
5.5 Average cluster purity for different clustering methods (dataset A) . . 36
5.6 Clustering text recovery rate for different feature sets (dataset A) . . . 37
5.7 Text recovery rate of cluster voting (dataset A) 39
5.8 Text recovery rate for voted training set (dataset B) 39

7.1 Final results of attack model 48

Chapter 1

Introduction

Security researchers have extensively analyzed the ways in which computer systems can unintentionally leak information thereby leading to a compromise of system security. Attacks of this nature are known as side-channel attacks and can utilize many forms of information leakage such as the fluctuations in thermal [23] and electrical energy [15, 19], variance in the time taken to perform calculations [14, 10] and the emanation of electromagnetic or acoustic signals [11, 26, 28]. In this case, we will be examining how the sound of a user typing can be used to fully reconstruct the user's input. This will be achieved by reimplementing the keystroke eavesdropping attack outlined in [35] which allows a classifier to be trained on an unlabeled audio recording by taking into account the underlying statistics of English text. We will also explore the methods used in other acoustic attacks to see if they can be incorporated and a number of extensions on the attack will be implemented which improve its effectiveness.

The next chapter provides a detailed explanation of existing acoustic attacks and also discusses a related technique which uses keystroke timing analysis to extract information from interactive SSH sessions. Chapter 3 then discusses how data was collected for this project, how it was processed and discusses some of the problems faced when dealing with keystroke recordings. A novel keystroke error function is also developed which allows for optimal thresholds to be found when extracting keystrokes from an audio recording. Chapter 4 details the different methods of feature extraction that were used and demonstrates how a combination of FFT and cepstrum features outperforms feature sets examined in previous attacks. Chapter 5 describes the clustering methods that were reimplemented from [35] and also describes an extension to their work which uses a voting system to fully automate the clustering process and increase its text recovery rate. This voting system is also then shown to provide a means of generating

a classifier training set without performing language model corrections. Chapter 6 describes the reimplementation of the language model used for correcting errors in the recovered text and finally chapter 7 describes how a supervised classifier was used to complete the attack by performing a number of iterations of classification and training label correction using the language model. An overview of how these different stages form the final attack is provied in figure 1.1.

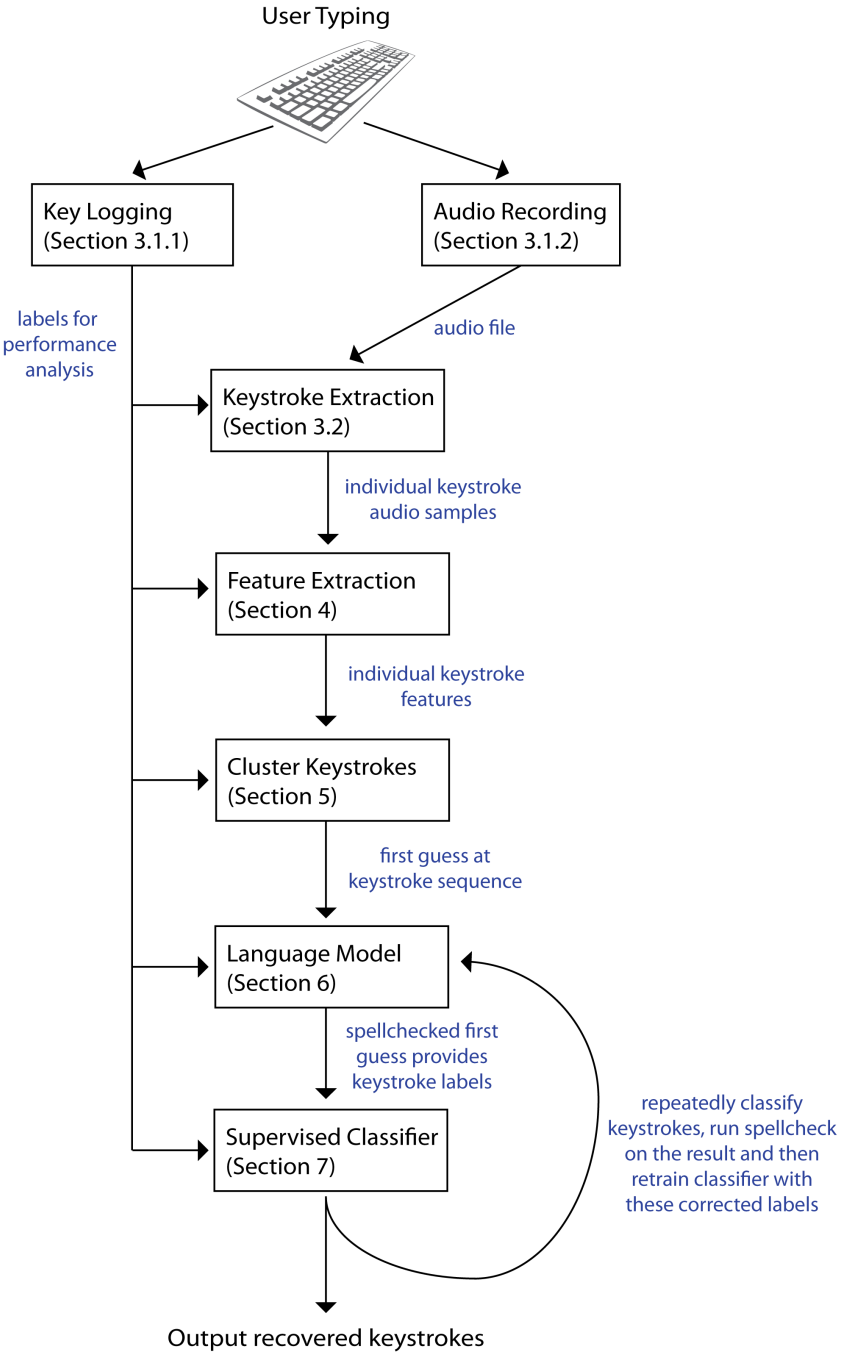

Figure 1.1: Attack overview

Chapter 2

Previous Attacks

2.1 Keyboard acoustic emanations

In the seminal paper [3], it was shown that each key on a computer keyboard makes a slightly different sound when struck due to the keyboard support plate being struck in different locations. These differences allowed a neural network to be trained to identify which keys had been pressed based solely on an audio recording of the user typing. To perform this classification they first identified the location of each keystroke in the recording and extracted an audio sample. They noticed that whilst each keystroke lasts approximately 100ms, there are actually two distinct audio events (shown in figure 2.1) which coincide with when the key is pressed (the press-peak) and when the key is released (the release-peak). Since the frequency distribution of the release-peak was much lower, the press-peak was chosen as a potential candidate for training features. However, on closer examination of the press-peak, they found that it too was also in fact comprised of two distinct events which correspond to when the finger first hits the key (the touch-peak) and when the key hits the keyboard support plate (the hit-peak). Since the touch-peak was found to be more clearly expressed in their recordings, they chose this as the source of their classifier features.

A set of features for each keystroke was then generated by calculating the Fourier transform coefficients of each event and normalizing them within the range [0,1] required for a neural network. The neural network was comprised of a set of input nodes equal in number to the extracted features, 6-10 hidden nodes and a set of output nodes totalling the number of keys used in the experiment. The network was then able to correctly classify 79% of keystrokes when evaluated against 300 test clicks comprised of 10 audio samples for each of the 30 keys they had selected for the experiment.

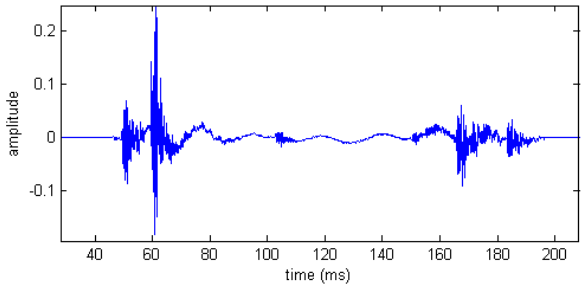

Figure 2.1: Audio signal of a single keystroke

While this attack showed that user input could be reconstructed from audio recordings, it did require a set of labelled data with which to train the network. It was also found that varying the number of fingers or the amount of force used to strike the keys meant that the network had to be retrained. This limits the attack's effectiveness since the network must be retrained for each individual user's typing style and to obtain a properly labelled training set, an attacker would have to know exactly what the user was typing whilst making a recording. It should be noted, however, that there were no differences in classifier performance between different typing styles once the network had been trained against that particular style.

2.2 Attacks without labelled training data

The attack outlined in the previous section was later extended in [35], where up to 90% of keystrokes were correctly recovered from a 10 minute audio recording without the need for labelled training data. Their method was comprised of the following stages:

1. Extraction of keystroke features from the audio recording
2. Unsupervised clustering of the keystrokes
3. Using character unigram and bigram statistics to map clusters to the correct keys
4. Applying a language model using word trigram statistics and a dictionary to spellcheck the recovered text
5. Training a supervised classifier with the corrected text
6. Running a number of iterations of using the supervised classifier on the original samples, reapplying the language model correction to its output and retraining the classifier.

The first stage was achieved by first extracting audio samples for each keystroke as with the previous attack. However, for feature extraction they tested both FFT and cepstrum features and found that cepstrum features gave much higher supervised classification accuracy (see figure 2.2).

They then performed an unsupervised clustering on these cepstrum features, splitting the keystrokes into K classes where K was chosen to be slightly greater than the the number of keys used in the experiment. Since these clusters will most likely contain mixtures of different keys rather than a one-to-many mapping, the cluster a keystroke belongs to can be represented as a random variable conditioned on the key's true label. This enables us to model the sequence of keystrokes along with the corresponding observed clusters as a Hidden Markov Model as shown in figure 2.3. By analyzing a large corpus of English text, we can also find a set of unigram and bigram statistics for the set of keys we wish to model which provides a prior distribution and transition matrix for the HMM. This just leaves the observation matrix, O, and the sequence of key states that are unknown. To find these values, the Expectation Maximization algorithm was used to alternately search on either the observation matrix or the key sequence whilst keeping the other fixed for a number of rounds. Since the results of this EM step were quite unstable, they initialized the observation matrix with random values but then seeded the matrix with the correct values for the space key. This was achieved by listening to the audio samples and hand labelling the space keys which have a distinctive sound when pressed. They then determined which clusters these space keys had ended up in and calculated the probability of the space key being in each cluster. Once a suitable observation matrix had been found, the viterbi algorithm was then used to find the most likely sequence of keys given the sequence of clusters. They found that after applying this clustering and HMM to the samples, they were able to recover keystrokes with just over 60% accuracy.

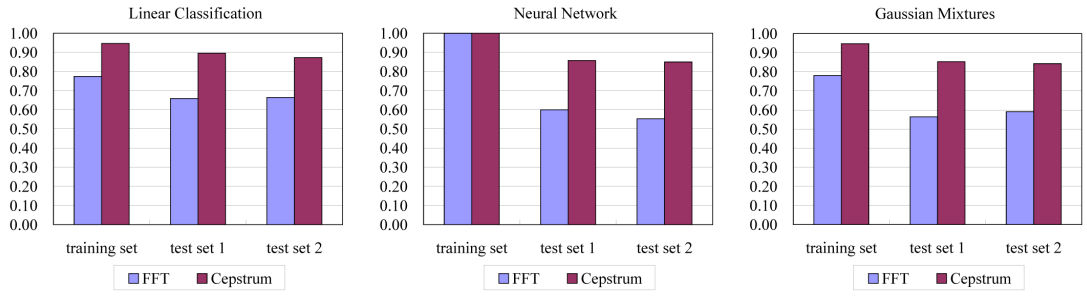

Figure 2.2: Recognition rates using FFT and cepstrum features [35]

They then improved the accuracy of the classification by correcting spelling mistakes present in the recovered text. This was done in two stages. First a dictionary file was used to find the 20 closest matching words to each recovered word and a probability was assigned to each one based on the number of characters difference between it and the original. Next they modelled each word in the recovered text as a random variable conditioned on the true word from the keystroke sequence. This enabled them to construct a second-order HMM (see figure 2.4) with a transition matrix calculated from English text word trigram statistics. After applying the language model, any words that had remained at least 75% unchanged were taken to be the correct spelling suggestion and their letters were then used to generate a labelled subset of audio samples. This subset of labelled keystrokes was then used to train a supervised classifier. The final step of the attack was to use this supervised classifier to repeatedly classify the entire set of keystroke samples, correct the resultant recovered text with the language model and then retrain the classifier. This was done until the classifier's performance no longer improved.

One of the major weaknesses of this attack is that it was only performed against a limited subset of keys and did not account for editing keys such as backspace. [30] has shown that in normal computer usage, the backspace key is infact more common than any of the alphanumeric keys. For such an attack to be successful in a real-world setting, the language model would need to be able to take these keys into account.

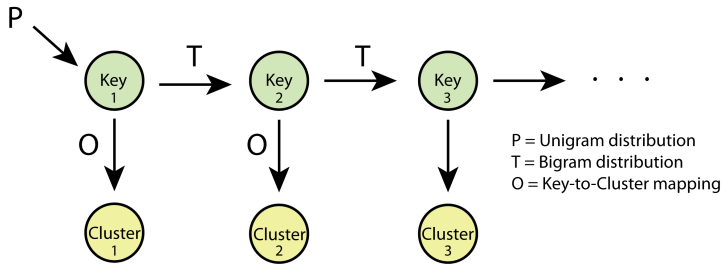

Figure 2.3: Keystroke Clustering HMM

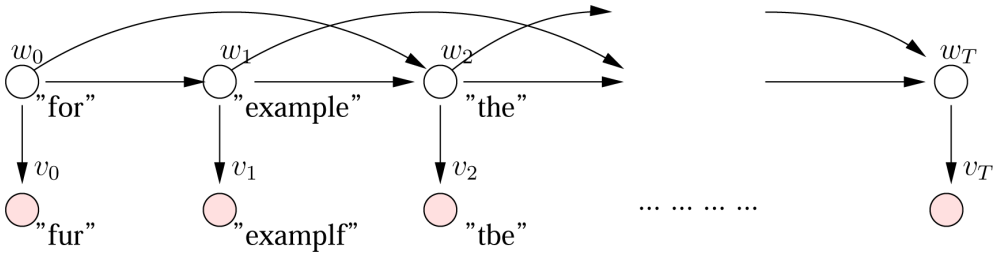

Figure 2.4: Trigram language model with spell correction [35]

2.3 Inter-key timing analysis

Since the only data likely to be available to an attacker will be the audio recording, it would be prudent to try to extract as much information from the audio data as possible. [29] has shown that the time between press-peaks can be used to reveal approximately 1 bit of information per keystroke pair. Their attack targetted passwords sent across interactive SSH sessions where each individual keypress is sent immediately in a separate IP packet thus providing a very accurate measure of keystroke timing. They modelled the inter-key time between every combination of key pairs as Gaussian distributions and found that they could group the timings into distinct categories (see figure 2.5). They then used these distributions to form a Hidden Markov Model and modified the Viterbi algorithm to output not one but n most likely candidate character sequences. The experimental results showed that by using this timing information during password entry it is possible to reduce the amount of time required to exhaustively search the password keyspace by a factor of 50, reducing the normal attack time of 65 days on a standard desktop computer down to just 1.3 days. This technique would appear to offer valuable information which an advanced acoustic attack may be able to utilize.

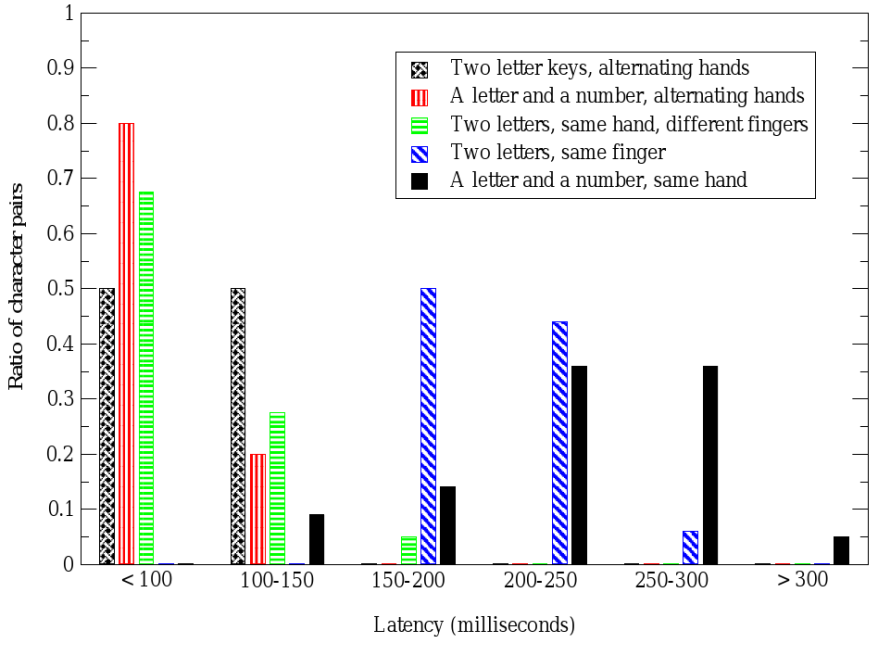

Figure 2.5: Categorized inter-keystroke timings [29]

However, an updated version [27] of the unlabeled attack paper has shown that after incorporating this timing information into their original attack, it only achieved a 0.16% improvement in classifier accuracy. They suggest that this may be due to the fact

that the original timing attack was performed against user passwords which are typed at a consistant typing rate, whereas in a more general acoustic attack, typing speed varies greatly throughout the recording. Although [27] is currently pending publication and so it has not yet been fully peer reviewed, it would seem unwise to persue this avenue of research during the project in light of these developments.

2.4 Dictionary attacks using key proximity constraints

An alternative approach to acoustic keylogging was presented in [4] which utilized a dictionary-based attack to predict individual words from short (5 second) audio samples. This attack exploits the fact that the distance between keys in feature space is proportional to the physical distance between them on the keyboard. This provides similar distance information to that of [29] but they then used this similarity measure to form a constraint satisfaction problem whereby a word of between 7 and 13 characters was represented by the distances between every combination of key pairs, categorizing them as either the same key, adjacent keys, near keys (at most 2 keys apart) or distant keys. They then applied this constraint to a dictionary of words to discover which were the most probable. Different feature types were tested for this feature proximity property by calculating their precision and recall whereby the "precision measures the fraction of constraints that hold true for the real word, relative to the number of constraints produced, for that specific constraint-type" and "recall measures the fraction of true constraints with respect to the total number of possible constraints in that category." Their tests showed that FFT features were able to achieve 28% precision and 29% recall whereas cepstrum features were able to achieve 39% precision and 42% recall. However, it was in fact the cross correlation of the raw audio signals which gave the best results with 65% precision and 89% recall. They found that the best constraints were found if they took the cross correlation between the press-peaks of the two keys and also between the release-peaks of the two keys and then calculated the average of these two values.

Unfortunately, this attack was only able to achieve a 73% success rate at including the correct word in a list of the top 50 most probable words. Furthermore, how applicable such a dictionary attack is to intercepting passwords which are unlikely to be a single dictionary word is unclear but their experiments clearly show that the cross correlation of raw audio provides a good distance measure between keys.

Chapter 3

Building Datasets

3.1 Data Collection

The first stage of the project was to collect audio recordings of a user typing against which an attack model could be tested. To acquire keystroke labels for these audio recordings, a keylogger was used to record which keys were pressed during the experiment.

3.1.1 Keylogging

A number of keylogging software packages were tested such as KeyCapture [31] and pykeylogger [1] but they either did not provide timestamps for each key event or in the case of KeyCapture had a tendency to crash whilst exporting the keylog, losing valuable experimental data. Instead, a custom data collection program, KeyRecorder, was developed specifically for these experiments. KeyRecorder was implemented in C++ using the Qt application framework [25] and provides a simple interface (see figure 3.1) which consists of a textbox for specifying the output filename, a text input area for typing the recorded text and experiment start and stop buttons. When the experiment is started, the program performs the following operations:

1. opens the output file with write permissions
2. installs an event filter which calls a logging function on every key event
3. clears the input area of any text and gives it keyboard focus
4. emits a 200ms beep and initializes a timer to zero

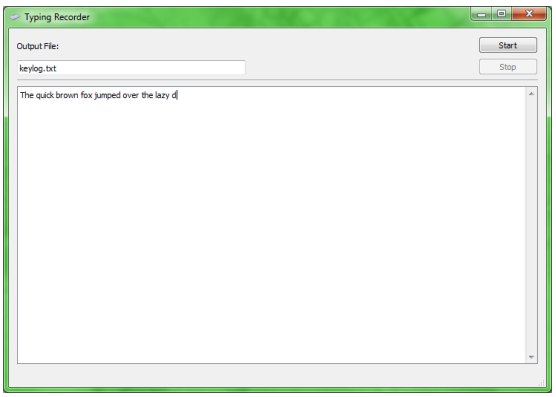

Figure 3.1: Typing Recorder user interface

The beep emitted at the start of the experiment signals in the audio recording when the key log timer was initialized. This allows for the key log and audio recording to be properly synchronized after the experiment. The logging function called by the event filter outputs three values to the log file on every call. These values are the timestamp of the key event in milliseconds, a plus or minus symbol for denoting whether the event is a key-press or key-release respectively and the key label as provided by the Qt library's nativeVirtualKey() function. Since plus and minus symbols are used to differentiate between press and release events rather than numerical values such as 0 and 1, it allows for either type of event to be easily filtered from the results using the grep utility. To stop the experiment, the user can either press the stop button or hit the escape key, at which point another 200ms beep is emitted and the log file is closed.

Since Qt is a cross-platform and internationalized library, the nativeVirtualKey() function outputs key symbols in its own custom representation. This project also uses its own representation for keys because we will be representing which physical key was pressed rather than the intended symbol - for example lower case and capital letters will share the same representation as they are produced by pressing the same physical key. To futher complicate matters, keyboard localization affects which physical keys correspond to which symbols. For example, figures 3.2 and 3.3 show the standard UK and US keyboard layouts. Notice that the number two and number three keys provide different secondary symbols depending on the localization used. The physical layout is also different in that, for example, there are two keys to the left of the Z key on a UK keyboard whereas the US keyboard only has one. With all these possible variations of symbol to physical key mappings it was decided that the KeyRecorder software should output the native Qt key codes to the log file and a separate convertKeys Matlab function was constructed which could convert between each different type of repre-

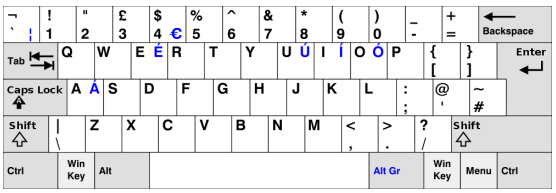

Figure 3.2: UK keyboard layout [34]

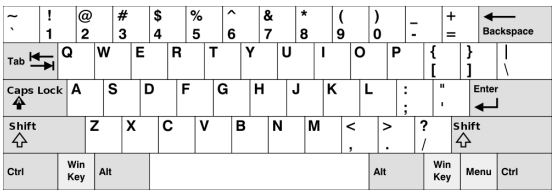

Figure 3.3: US keyboard layout [34]

sentation. The advantage of this method is that the convertKeys script can be easily updated for alternative keyboard layouts without having to recompile the KeyRecorder software or altering any other scripts in the project.

Before processing the keylog file, we must first define which physical keys are to be represented in our model and how these keys map to the various different key representations. The convertKeys script contains four data structures which allow us to makes these definitions:

- keyLayout
- qtKeys
- asciiKeys
- textKeys

First of all we must populate the keyLayout data structure with an ordered list of labels for the physical keys we wish to model. These labels are purely descriptive so typical values will be “Letter A” or “Enter”. The last label in the list should always be the “Unmapped” key which provides an internal representation of any keys we wish to ignore without having to explicitly list every key on the keyboard. If we are only interested in a subset of the keys used during an experiment, we can selectively unmap those keys we wish to ignore and they will be automatically assigned to the unmapped label. The rest of the project represents keystrokes by a simple numeric value which is calculated by convertKeys from the position of the keystroke in this list of labels.

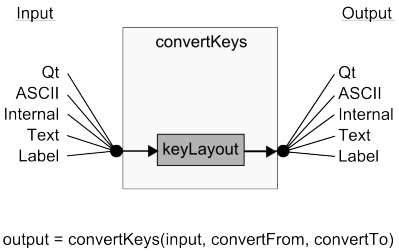

Figure 3.4: convertKeys function

Once we have defined these key labels, the three other data structures must then be populated. These provide the mappings between our internal keyLayout representation and the other representations we will need to convert between. The qtKeys and asciiKeys provide mappings for Qt's internal representation and the ASCII character table respectively. The textKeys mapping, however, is similar to the asciiKeys mapping but maps all non-printing characters to the underscore character. This representation is used throughout the project when outputting keystrokes to screen as it provides a visible placeholder to show where non-printing keys such as backspace have occurred.

The convertKeys function (see figure 3.4) converts between two different representations by first converting the input into the experiment's own internal representation and then converts this into the desired output representation. Consequently, converting to and from the same representation will not necessarily give an identical output as some keys may be unmapped but this provides a useful means of stripping an input of those characters we wish to ignore.

3.1.2 Audio recording

To record and edit the audio samples for this experiment, the open source editing software Audacity [20] was used. All recordings were initially recorded at a 96KHz sampling rate as 32-bit float WAV files. It was noted in [3] that keystrokes mainly occupy the 400-12,000 Hz frequency range and taking a spectogram of one of the audio recordings (see fig 3.5) shows this to be the case. Since higher sampling rates can greatly increase the amount of time required to process the data, it would be advantageous to reduce the sampling rate as far as possible without losing valuable information. The Nyquist sampling theorem states that "every signal of finite energy and bandwidth W Hz may be completely recovered, in a simple way, from a knowledge of its samples taken at the rate of 2W per second (Nyquist rate)" [16]. This means that for a maxi-

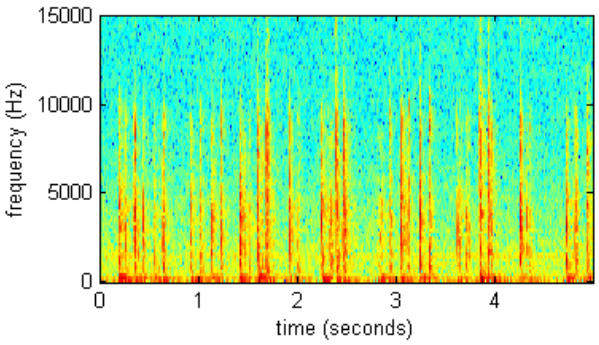

Figure 3.5: Spectogram of keystroke audio recording

mum keystroke frequency of 12 KHz, a minimum sampling rate of 24 KHz is required to fully capture the signal. However, whilst the majority of the signal is within this range the spectogram shows that some keys do infact extend into much higher frequencies which may provide a useful means of differentiating them from others. Therefore, the audio samples were downsampled to a rate of 44.1 KHz which provides a reasonable trade-off between computational requirements and retaining possibly valuable frequencies.

The editing process simply involved locating the two KeyRecorder beeps present in the recording and then extracting the audio between these two points. A quick check was then performed to ensure that the sound of the first keystroke coincided with the first timestamp recorded in the keylog.

3.1.3 Experimental setup

Keystroke recordings were taken in a sound-proofed audio lab using a studio quality microphone. Two different keyboards were used for data entry, namely a Dell RT7D50 USB Keyboard and a Logitech G35 Gaming Keyboard. The language corpus used in this project was the “Australian Broadcasting Commission 2006 Science Corpus” [8] and samples of text for typing were taken from both this corpus and also “Flatland: A Romance of Many Dimensions” [2] which is a short fictional novel that contains a large amount of scientific terminology. Table 3.1 provides a listing of the datasets recorded in these experiments.

Dataset	A	B	C	D
Keyboard	Dell	Logitech	Dell	Logitech
Total keystrokes	1813	1202	1021	1923
Audio duration	10:39	08:14	05:51	06:29
Typing rate (cpm)	170	146	175	297

Table 3.1: List of recorded datasets

3.2 Keystroke extraction

Once a recording has been obtained of a user typing, we must then identify each individual keystroke within the recording and extract an audio sample of each keypress event.

3.2.1 Using keystroke labels

Before implementing an automatic keystroke extractor, a supervised version was created which used the timestamps provided by KeyRecorder to locate precisely when each key event had occurred. An initial implementation of this module simply read in each timestamp, T_i, from the keylog and extracted a sample of audio between times T_i and $T_i + \Delta$ with the sample length, Δ, specified as a parameter at runtime. After testing this implementation, however, it became clear that the keylog timestamps were not completely accurate and could deviate from the correct value by as as much as 4ms when compared to the actual keystrokes present in the audio recording. One possible explanation for this was that the Qt library uses a queued event loop for processing all window events including keystrokes. If the computer was under heavy load then this may have caused a delay between the actual keypress and the logging function being called. To test this hypothesis, another set of data was recorded by first terminating all non-essential processes on the computer and running the keylogger at a high priority. However, this made no improvements on the timestamp accuracy.

Recall from section 2.1 that each keypress event is comprised of a touch-peak, when the finger touches the key, and a hit-peak, when the key makes contact with the keyboard backplate. The computer on the other hand, will receive a keypress event when the key compresses the rubber membrane beneath it, far enough to close the electrical contact. If one gradually presses down on a key, the point at which it starts

to generate characters actually occurs before the point where the key fully reaches the backplate. This means that timestamps recorded in the keylog will correspond to a time somewhere in between the touch-peak and hit-peak in the audio recording. Since the speed at which the key is pressed will affect the length of time between these two events, this may have been a contributing factor to the timestamp inaccuracies.

To eliminate these inaccuracies and ensure each audio sample is taken from the very beginning of the keystroke, the method for extracting keystrokes demonstrated in [4] was adapted to analyze the audio signal within the region of each timestamp. Their method for identifying keystrokes was to first split the audio into 10ms time bins and calculate the FFT coefficients for each bin. The coefficients corresponding to the 400-12,000 Hz keystroke frequency range were then summed over, giving a set of bin energy levels. They then normalized these energy levels to values between 0 and 1 and calculated a delta vector by subtracting from each bin the energy of its predecessor. Large values in the delta vector were then searched for which would indicate a surge in energy due to a key being pressed. This method was incorporated into the extractor by only analyzing a 40ms time window centered around each timestamp, ensuring that we do not wander too far and invalidate the keystroke label. The time bin FFT coefficients were then calculated using the specgram() function from Matlab's signal processing toolbox [18] and from these the delta vector was calculated. The largest two elements of the delta vector were then searched for which correspond to the touch-peak and hit-peak. Once the touch-peak had been found, a backwards search was performed, traveling down the peak's gradient until the local minimum was reached signifying the start of the peak and the beginning of the keypress event. The keylog timestamp was then updated with this new value and a sample of length Δ was extracted.

3.2.2 Extraction error function

Once an accurate source of keystroke timestamps had been generated, the performance of potential unsupervised extractors could be tested. This required a method of comparison which would give an indication of how close extracted timestamps were to to the true values. An initial approach was to loop through each key in the extracted sequence and find the closest matching reference key. The time difference between each pair of keys was then calculated and summed across the entire dataset. However, this method does not take into account missing keys that have failed to be detected or completely erroneous keys where a key event has been mistakenly identified. This

motivated the development of a more advanced error measure which could take all of these factors into consideration. When creating such an error function, we would like it to adhere to a number of sensible constraints:

- the error is 0 if all keys are correctly extracted with identical timestamps to the reference data
- the error increases as the time difference between the extracted and reference keys increases
- the error increases when a key is missing from the extracted key sequence
- the error increases when erroneous key events have been detected which did not really occur
- the error incurred by an inaccurate timestamp cannot exceed the error for missing a key entirely

An error function which meets these requirements can be achieved by first representing each key event by a triangular waveform centered at its timestamp, with width W and area normalized to 1. Figure 3.6 shows an example set of reference and extracted key events. The reference keystrokes were taken from a set of 10 events spaced 300ms apart but with the 4th event removed. The extracted keystrokes were taken from a similar set of 10 events but with the 2nd event missing, event 6 shifted forward by 50ms, event 8 shifted backward by 25ms and an additional event placed 5ms after the last reference event. Figures 3.7 and 3.8 show the result of applying this triangle wave function to each set of events. Once both sets of keystrokes have been converted into this new form, the error function can then be calculated by taking the absolute difference between the two signals as shown in figure 3.9.

Notice in figure 3.9 that a key missing in the extracted sequence (at 600ms) and an erroneous key in the extracted sequence (at 1200ms) both produce an increase in error by 1 since the triangular waveform is normalized to have an area of 1. Also, for the two extracted events which were shifted away from the reference timestamp, an error is produced within the range [0,1] proportional to the distance away from the true timestamp. For these events which are detected but not properly aligned at the true timestamp, we can specify over what time interval the error increases before being considered as a completely erroneous detection by adjusting the triangle wave width, W. Finally, the last extracted event (at 3005ms) demonstrates that the error produced by a particular event is completely unaffected by other neighboring key events.

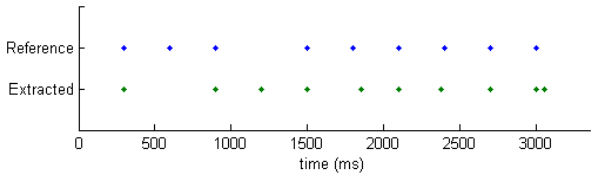

Figure 3.6: Example key events in reference and extracted sequences

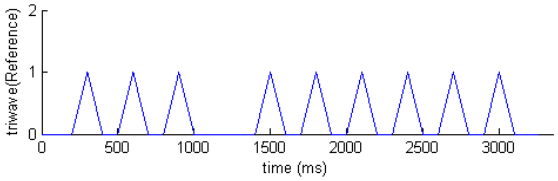

Figure 3.7: Triwave function of reference timestamps

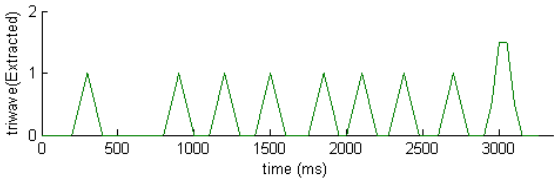

Figure 3.8: Triwave function of extracted timestamps

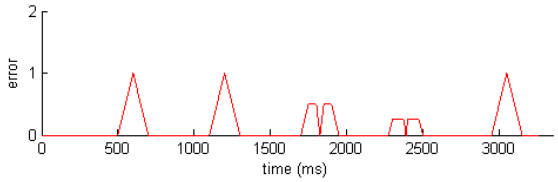

Figure 3.9: Resultant error for example reference and extracted data

3.2.3 Unsupervised keystroke extraction

The three acoustic attacks discussed in chapter 2 all used slight variations of the same method for detecting keystrokes without any clear advantages between them. Since the attack outlined in [35] was the one being reimplemented, their method was chosen over the others.

The bin energy levels of the audio signal were calculated using the FFT method described in section 3.2.1. A key event was then detected when the energy level reached a manually set threshold, at which point the next 100ms of audio was extracted as a keystroke sample. Figure 3.10 shows an example of the extractor detecting a keystroke

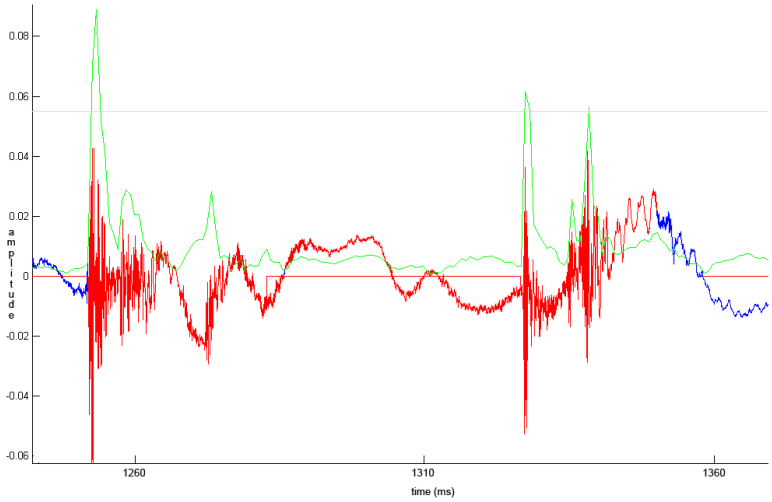

Figure 3.10: Example unsupervised keystroke detection

event. Here the audio signal is represented by the blue line and the portion of the signal the extractor has identified as a keystroke is in red. The bin energy levels are represented by the green line and finally the threshold value is represented by the yellow line. Here you can see that once the bin energy level reached the threshold, the next 100ms were taken to be a keystroke.

To automate the process of threshold selection, the timestamps of keys present in the first 15 seconds of audio were found using the supervised extractor. However, they could also have been obtained by simply hand labelling the data if the attack was actually being carried out. The error function described in the previous section was then applied to these timestamps together with the output of the unsupervised extractor. By searching over a range of threshold values, the error was then minimized to find the threshold which gave the best results. Figure 3.11 shows the results of performing this search over dataset B. The first 15 seconds of the dataset were comprised of 45 keystrokes and at a threshold of 0.078 the error was minimized to a value of 11.4, recovering all 45 keys in the sample. Although the extractor was able to recover all of the keys, the error rate was still high due to poor alignment of the timestamps. Next, the extractor was run on the entire dataset (totalling 1202 keystrokes) using the threshold found. It recovered a total of 1203 keystrokes with an average absolute timing error of 14ms. Similarly, figure 3.12 shows the results of performing the search over dataset A. The sample of this dataset contained 47 keystrokes and only managed to achieve a minimum error of 20.1 with a threshold value of 0.040. Running the extractor over the entire dataset (totalling 1813 keystrokes) gave a total of 1909 recovered keys with an average absolute error of 36ms.

Previous attacks have all stressed the need for manually checking and correcting the keystroke extraction phase of the attack. Since the supervised extractor already provides a full correct set of timestamps, it was used as the source of keystroke samples for the remainder of the project rather than correcting results by hand. In the next section we will explore the reasons for the unsupervised extractor's poor performance.

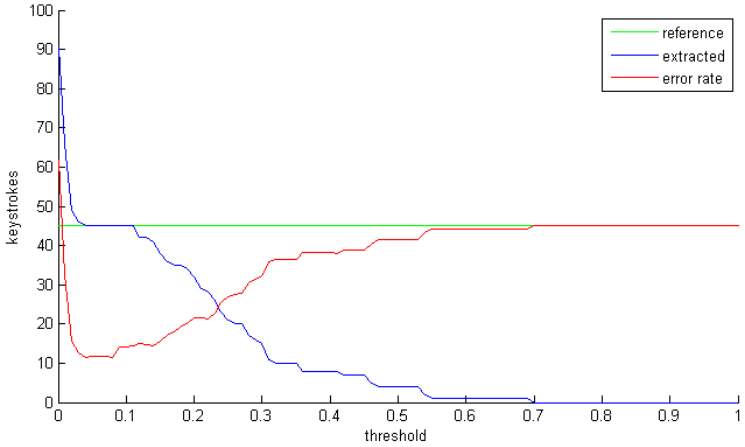

Figure 3.11: Unsupervised threshold search on dataset B

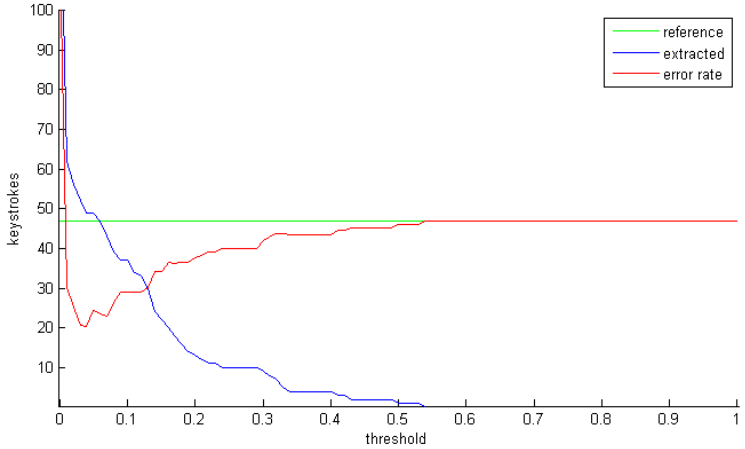

Figure 3.12: Unsupervised threshold search on dataset A

3.3 Problems with the data

When the authors of [35] tried to reproduce the results presented in the original attack [3] they found that they were unable to match the high performance rate presented for supervised classification on FFT features. Similarly, the data collection phase of this project was very time consuming as the performance levels listed in both papers set a baseline on how well their models should perform. Certain properties of the dataset cause problems for both keystroke extraction and supervised classification.

3.3.1 Keystroke overlap

An examination of the data revealed that the assumption made in [35] that each keystroke will be separated by more than 100ms is not always true. It was argued that since each individual keystroke takes approximately 100ms and an average user types at under 300 characters per minutes, then the keystrokes should be easily distinguishable with a gap greater than 100ms between them. However, a user does not type at a constant speed but generally types in small short bursts corresponding to individual words or sentences. This means that whilst the overall average typing rate may be under 300 cpm, these short bursts contain much higher typing rates.

Dataset D contains a total of 1923 characters recorded over 389 seconds which gives an average typing rate of 296.6 characters per minute. However, when the time between keystrokes was calculated, it was found that 204 of these keystrokes occurred within less than 100ms of the previous key. This amounts to 10.61% of the total keystrokes present in the dataset. Furthermore, 76 of these keystrokes (3.95%) actually occurred before the previous key had been released resulting in an overlap of the two keys' audio signals. These overlaps still produce correct input for the user because the computer registers each keystroke by the keypress event meaning the user must only ensure proper ordering of the press peaks of each keystroke for the input to be valid. Figure 3.13 shows such an overlap between the keys "e" and "r" whilst typing the word "diverges". Such pairs of events are indistinguishable from a single keystroke using our current methods of unsupervised key extraction. At much lower typing rates such as in dataset B (146 cpm), all keystrokes were well separated by at least 100ms between each key.

3.3.2 Phantom keystrokes

Another source of errors in the dataset were phantom keystrokes. These phantom keystrokes were events in the audio sample that didn't correspond to any user input recorded in the keylog. There appear to be two contributing causes for these events. Firstly, hesitation strokes were events in the audio sample where only touch peaks occurred. This generally happened because the user had begun to press a key and then, realizing they had made a mistake, released the key before a hit peak occurred. This type of event occurs surprisingly often without the user consciously realizing. A second source of phantom keystrokes is due to the hand resting position used by the typist. The thumbs are often rested on the space key during typing which produces a noticeable rattle due to the key's large size. This was often loud enough to register as a touch peak in the audio sample.

3.3.3 Background noise

Since all of the recordings were made in a quiet environment, the attack did not have to contend with the normal background noise of a typical office environment such as people talking, phones ringing, a radio playing in the background, etc. However, the recording's spectogram (see figure 3.5) shows a clear band of activity at the low end of the frequency spectrum across the entire recording. To investigate this, a sample of audio was extracted from the recordings where no keystrokes were taking place. Figure 3.14 shows the Fourier transform of this sample. Unfortunately the laptop computer used to take the recordings had a particularly noisy fan which produced a clear peak at 1200Hz. This falls within the range of keystroke frequencies we wish to analyze and so may have an impact on the attack's performance.

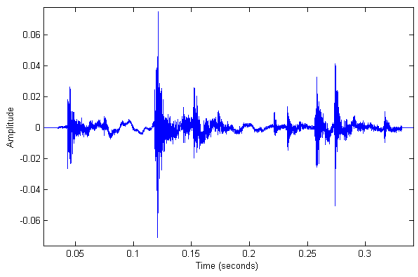

Figure 3.13: Overlapping keystrokes

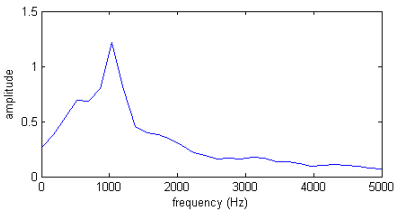

Figure 3.14: Frequency domain of background noise

Chapter 4

Extracting Features

Once keystroke samples had been extracted from the audio recording, they then needed to be converted into sets of features. The accuracy a classifier can achieve when using a chosen set of features will place an upper limit on the performance of the entire attack. This is because the majority of the attack model is concerned with generating accurate labels on which to train a supervised classifier. It is essential to find the best possible feature representation, to ensure the attack is capable of performing well. The previous acoustic attacks have shown that FFT features, cepstrum features and also simply using the raw audio signal as a set of features can all provide useful representations for classification. With this in mind, all three feature types were examined and a fourth hybrid feature set was introduced which uses a combination of FFT and cepstrum features. Once all four methods were implemented, they were then compared to see which gave the highest performance when trained against a labelled dataset.

4.1 Plain audio

The simplest feature set was simply to use the raw audio signal. It was shown in [4] that the correlation between the audio signals of two keys is proportional to the physical distance between them on the keyboard. This suggests that it may offer useful information when performing keystroke clustering in the next chapter. Due to its high dimensionality however, it is unlikely to perform well at supervised classification but for completeness it was also tested alongside the other feature sets. Since the keystroke extractor provides 100ms samples at a sampling rate of 44.1 KHz, this would produce 4410 features if the samples were passed through unchanged. Clearly, this number of features would be far too high so a number of feature sets were tested whereby only

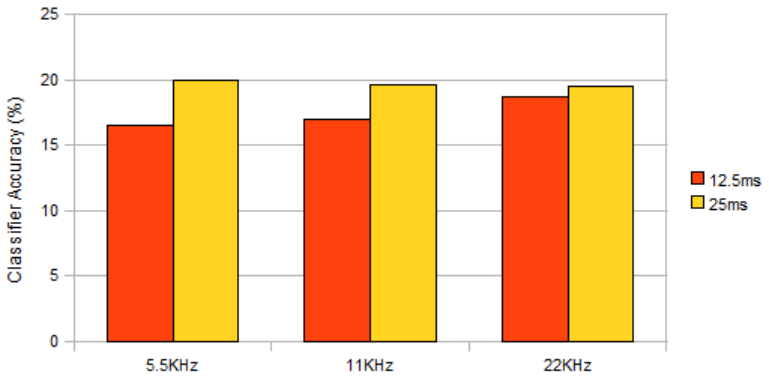

Figure 4.1: Classification accuracy using plain audio features

the first 25ms or 50ms were taken from the sample and this was then downsampled to 22KHz, 11KHz or 5.5KHz using Matlab's resample() function. Figure 4.1 shows the results of training a support vector machine on all of these variations and testing them using 10 fold cross-validation on dataset A. The resultant classifiers all performed badly and tended to classify every key as either the space key or E key. Since these two keys occur most frequently in the dataset, the classifier was still able to achieve up to 20% accuracy.

4.2 FFT features

FFT features were used in the original acoustic attack [3] which enabled a neural network to achieve an accuracy of 79% on a labelled set of data. The method of extraction was to first take the Fourier transform of the audio signal and select the coefficients which correspond to the keystroke frequency range. These coefficients were then split into n bins and each bin was summed over giving a total of n features. This was achieved in the project by using Matlab's fft() function and once the features had been found for all keystrokes, they were then normalized by subtracting the mean and dividing by the standard deviation across each feature.

An examination of these features revealed that whilst none of the keys were easily separable from all other keys, there were clear boundaries present between pairs of keys in a lot of the feature space. For example, figure 4.2 shows a plot of features 1 and 16 from dataset A using 50 bins. Here you can see that whilst neither the E or T keys are easily separable from the entire population of keys, there is a clear boundary between them when looked at in isolation.

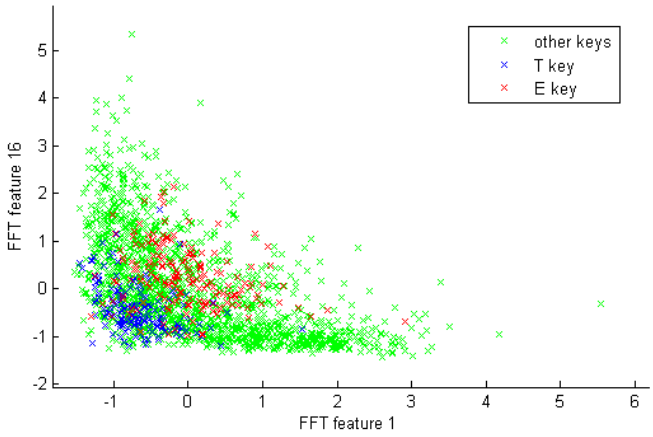

Figure 4.2: Scatterplot of FFT features 1 and 16 (out of 50) for dataset A

When calculating the FFT features there is also the question of how many bins will give the best classifier performance. Whilst examining this question, an interesting set of results was found. Figure 4.3 shows the results of testing between 5 and 100 FFT bins in 5 bin increments using a support vector machine and 10 fold cross-validation. Since the most common key, the space key, makes up approximately 18% of both datasets, it would not be surprising to find a minimum classification accuracy of 18% since this can easily be achieved by always picking the most popular key. However, the classifier seemed to achieve an unusually high level of accuracy on dataset B and most surprisingly, when the feature set size was reduced to only 2 bins the classifier obtained an accuracy of 29.7% and could correctly classify 100% of the space keys with 0 false positives.

Figures 4.4 and 4.5 show the average Fourier transform of the space keys and of the remaining keys for datasets B and A respectively. These show that in dataset B, the space key has an unusually high magnitude across the entire frequency domain. It was first assumed that this was due to the space key being hit with much greater force than normal during the recording. However, figures 4.4 and 4.5 show that these space key characteristics are infact consistant for each keyboard across different experiments. It would seem that the Logitech keyboard has a loud space bar which makes it very easy to distinguish from the rest of the keys. The space keys from both keyboards show an above average response for frequencies below 2KHz so it seems this may be a natural consequence of the key being considerably larger than the rest of the keys on the keyboard. However, without analyzing recordings from a number of different keyboards it is unclear if the loudness of the space key on the Logitech keyboard is

an anomaly or a common occurrence. Since the language model in this attack heavily relies on the correct identification of space keys to delimit words, any results based on the Logitech keyboard should be treated with suspicion before first determining if it has an atypical advantage over other keyboards. Taking results from only the Dell keyboard for FFT classification still gives a maximum accuracy of 85% which is 6% higher than that achieved in [3].

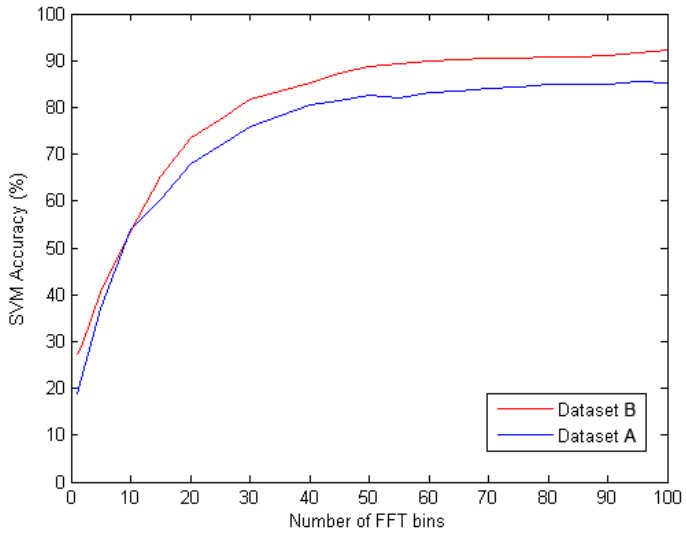

Figure 4.3: Performance evaluation on varying the number of FFT bins using datasets A and B

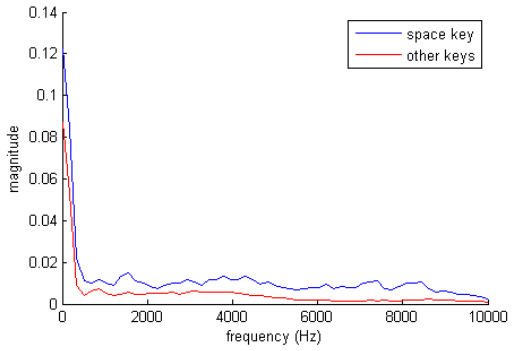

Figure 4.4: Frequency domain of keys in dataset B (Logitech keyboard)

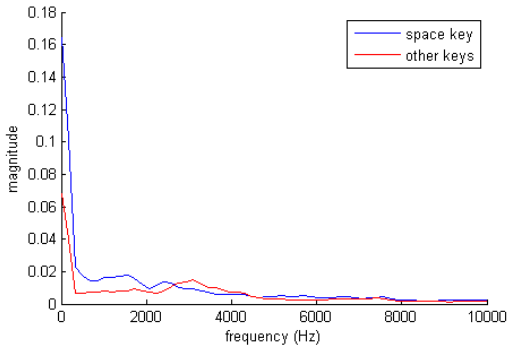

Figure 4.5: Frequency domain of keys in dataset A (Dell keyboard)

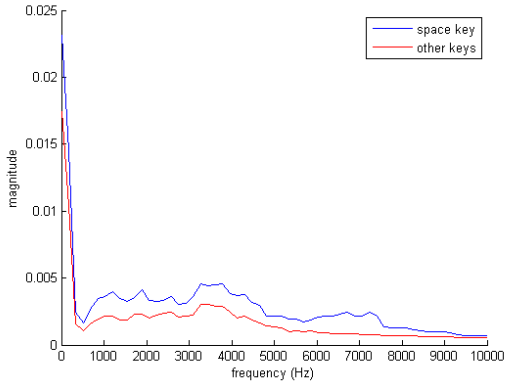

Figure 4.6: Frequency domain of keys in dataset D (Logitech keyboard)

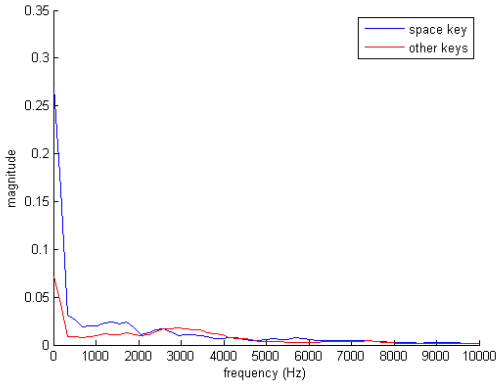

Figure 4.7: Frequency domain of keys in dataset C (Dell keyboard)

4.3 Cepstrum features

Cepstrum features were first introduced in [5] where they were used to find echo arrival times in a composite signal. The cepstrum of a function is defined as the power spectrum of the logarithm of the power spectrum of that function. [35] showed that using a particular type of cepstrum features, Mel-Frequency Cepstral Coefficients [13], to represent keystrokes gave much higher classification accuracy than FFT features achieving accuracies as high as 93% on labelled training data. To calculate the cepstrum features in this project Voicebox [22], a speech processing toolbox for Matlab, was used. The optimal parameters found in [35] were used with a filterbank of 32, a window time of 10ms, window shift of 2.5ms and the first 16 coefficients were taken. On testing these features with the collected data using a support vector machine and 10 fold cross-validation, they achieved a classification accuracy of 88.9%. However, these cepstrum features were then combined with the FFT features calculated in the previous section by simply concatenating the two sets of values. This increased the performance up to an accuracy of 93.9%. Figure 4.8 provides the results of testing these different feature methods using various classification methods. Clearly, the combination of FFT and cepstrum features gives much higher classification performance than either set individually.

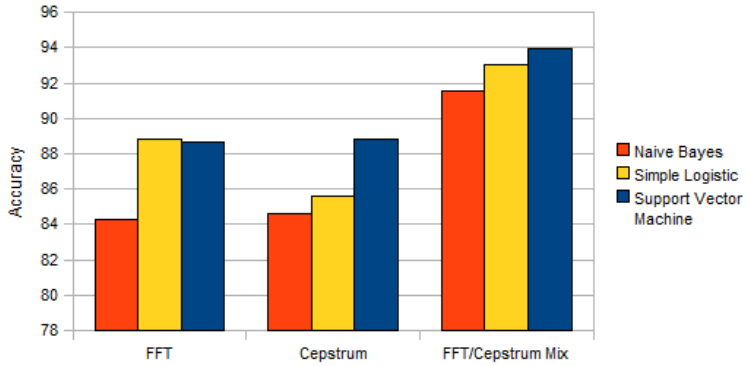

Figure 4.8: Supervised classifier accuracy over different feature sets

Chapter 5

Keystroke Clustering

Once a suitable set of features had been found, the next stage of the attack was to perform unsupervised clustering on the keystrokes. Two different clustering methods were explored, namely K-means and Gaussian Mixtures. Once the data had been clustered, the correct cluster to key mapping was then estimated using a HMM to give an initial attempt at classifying the audio samples. Although the mixture of cepstrum and FFT features gave the highest performance for supervised classification, there is no reason against extracting two separate sets of features for classification and clustering if a different set of features gives superior clustering performance. Since [4] has shown that the correlation between keystroke audio signals allows us to infer information about their physical proximity, the different feature sets including raw audio signals were tested.

5.1 Clustering methods

An initial assessment of clustering performance tried to determine the appropriate number of clusters to use with the dataset. Figures 5.1 and 5.2 show the cluster variance for K-means and GM respectively with the number of clusters ranging from 2 to 100. Figure 5.1 shows that the K-means algorithm rapidly improves as the number of clusters approaches 10, at which point there is a slight elbow to the curve and the variance begins to decrease less rapidly. This is not surprising when we look at the distribution of keys in the dataset. Figure 5.3 shows the cumulative frequency of keys that have been sorted into descending order of frequency. Here you can see that the first 10 most popular keys represent 76.7% of all the keystrokes present in the dataset and the first 25 keys represent 99.2% of the dataset. Clearly, if we wish to map clusters to

individual keys, the number of clusters should be at least as many as the total number of keys present in the recording. However, the cluster variances do not seem to give a good indication of how many this should be.

It is suggested in [17] that cluster purity cannot be used to determine the correct number of clusters because as the number of clusters approaches the number of datapoints, the purity will tend towards 100% where each datapoint is assigned its own individual cluster. Instead, they suggest that the Normalised Mutual Information is used as a comparison as it provides a measure of the mutual information between clusters and classes but unlike the purity score, it is normalized by the entropy of the clustering which tends to increase with the number of clusters. For a set of clusters $\Omega = \{w_1, w_2, ..., w_K\}$ and set of classes $C = \{c_1, c_2, ..., c_J\}$, the NMI is defined as,

$$NMI(\Omega, C) = \frac{I(\Omega; C)}{[H(\Omega) + H(C)]/2}$$

Figure 5.4 shows a plot of the NMI for the K-means clusterer across varying numbers of clusters. Again, this shows that once we reach 10-15 clusters, the NMI tends to plateau out. It would seem that due to the highly skewed distribution of keys, clustering can only capture the general structure of the data determined by the 15 most frequent keys.

However, we must also consider how these clusters are going to be used in the next stage of the attack. [35] likened the task to breaking a substitution cipher, where we will first try to determine a map between clusters and keys and then given this map, we will recover the key sequence. In terms of clustering, these two tasks place directly opposing constaints on the number of clusters. As the number of clusters increases, the purity of each cluster also increases which means given the appropriate mapping we can obtain a more accurate recovery of the original text. However, as the number of clusters increases, it also dilutes the n-gram statistics in much the same way that a homophonic cipher [32] substitutes the plaintext symbols with more than one ciphertext symbol to defend against statistical analysis. The more clusters there are, the harder it will be to find the appropriate mapping. Consequently, the number of clusters will need to be tested directly against the recovery rate of the next section to determine the best value.

Cluster purity does however provide a means of comparing clustering methods. When both K-means and GM were tested, K-means consistently achieved a higher average cluster purity across all feature sets and for all cluster counts tested. Figure 5.5 shows an example of one of these tests giving the average cluster purity for K-

means, GM and random assignment over varying numbers of clusters using the FFT and cepstrum mixture features. It was for this reason that K-means was selected as the clustering method for the remainder of the experiments.

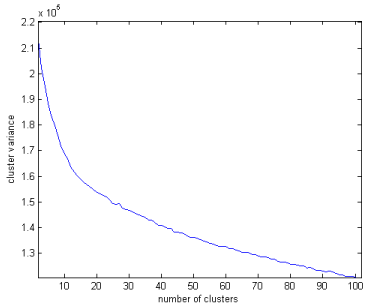

Figure 5.1: K-means cluster variance for different numbers of clusters (dataset A)

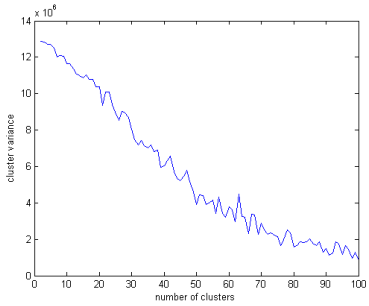

Figure 5.2: Gaussian Mixture cluster variance for different numbers of clusters (dataset A)

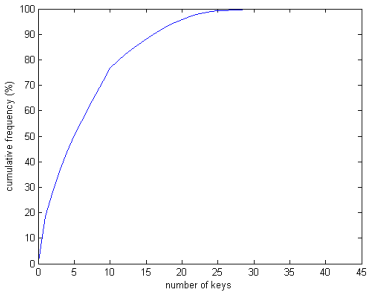

Figure 5.3: Cumulative frequency of keys (dataset A)

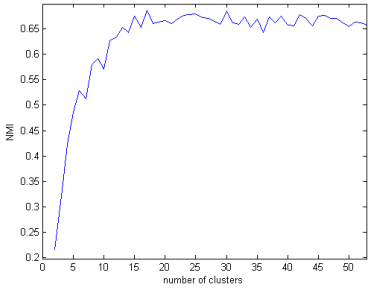

Figure 5.4: Normalised Mutual Information for varying numbers of clusters with K-means (dataset A)

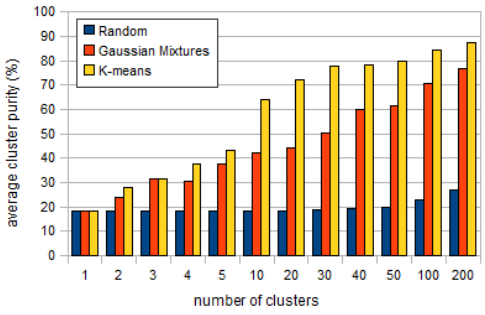

Figure 5.5: Average cluster purity for different clustering methods (dataset A)

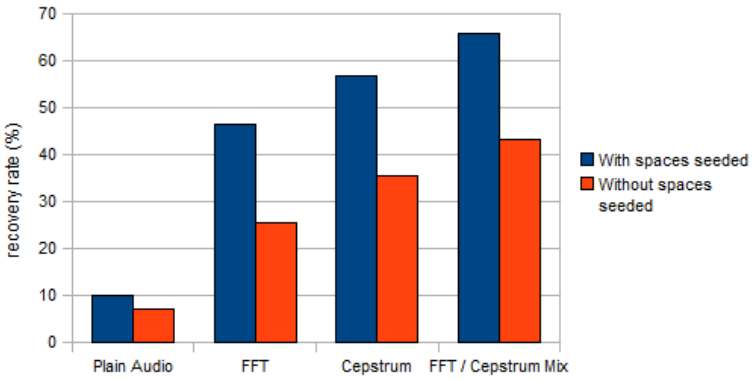

Figure 5.6: Clustering text recovery rate for different feature sets (dataset A)

5.2 Assigning keys to clusters

As discussed in section 2.2, the method used in [35] to map clusters to their appropriate keystrokes was to represent the cluster as a random variable conditioned on the keystroke. This enabled them to then model the key sequence using a HMM (see figure 2.3). In this project, the Hidden Markov Model Toolbox for Matlab [24] was used to create the model. Firstly, character unigram and bigram statistics were extracted from the project corpus and converted into the project's internal representation using the convertKeys script (see section 3.1.1) thus providing a prior distribution on the first keystroke and a transition matrix for the HMM. A random observation matrix was then created and the row corresponding to the space key was seeded with the correct values. During their clustering experiments, [35] found that the EM algorithm was highly unstable and required a number of trials before a suitable observation matrix was found but seeding the matrix with space key values greatly reduced the problem. They stated that the seed was obtained by listening to the audio samples and hand labelling the space keys which have a distinctive sound when pressed. They then determined which clusters these space keys had ended up in and calculated the probability of the space key being in each cluster. However, since a full set of keystroke labels were available, this process was automated by simply running through the cluster assignments and counting how many spaces had been assigned to each cluster and then normalizing the counts so that they summed to 1. The dhmm_em() function was then used to perform Expectation Maximization on the observation matrix and once a suitable matrix had been found, the viterbi_path() function was used to find the most likely sequence of keys given the sequence of clusters.

After testing different numbers of clusters, it was found that 50 clusters gives the

highest text recovery rate. Figure 5.6 shows the average text recovery rate from either seeding or not seeding the observation matrix using K-means with 50 clusters on each different feature set. Here you can see that seeding the matrix greatly increases the average text recovery rate and also that the FFT and cepstrum mixture of features achieves the highest accuracy.

5.3 Cluster voting

The biggest criticism of the clustering method described in the previous section is that it requires manual labelling of the space key for the EM algorithm to reliably converge on a suitable observation matrix. Even when the space key is used to seed the matrix, it is also difficult to determine which approximation of the matrix is best because a higher likelihood as determined by the viterbi algorithm doesn't necessarily mean the recovered text is more accurate. This motivated a deeper examination of the problem to try and find a solution which doesn't require hand labelling of the data.

To determine an upper limit on the text recovery rate, the entire observation matrix was calculated by taking counts of which clusters each key was placed in. Using this matrix, an average text recovery rate of 86.3% is possible.

A new method of clustering was then developed whereby both the K-means clustering and HMM were run multiple times and then from these multiple sets of recovered text, a count was taken on each character to find the most popular prediction. This is akin to bagging [6] on the HMM as the results of the clustering algorithm are essentially being resampled on each iteration. Figure 5.7 shows the results of performing this procedure over 10 iterations. This new method achieves higher accuracy without the space key seeded than the original method can achieve with the seed. If a seed is used with this voting method then a text recovery rate of 75% can be achieved.

Furthermore, if the percentage of votes each character received are then taken as confidence values, a training set can be generated from these results by setting a threshold on the total votes received. Figure 5.8 shows the results of training a probabilistic neural network on a training set generated from dataset B using a seeded observation matrix and with a vote threshold of 70%. Remarkably, this classifier is able to achieve an accuracy of 84.1% at recovering the keystroke sequence. Using the same parameters for dataset A achieved a slightly lower accuracy of 77.66%.

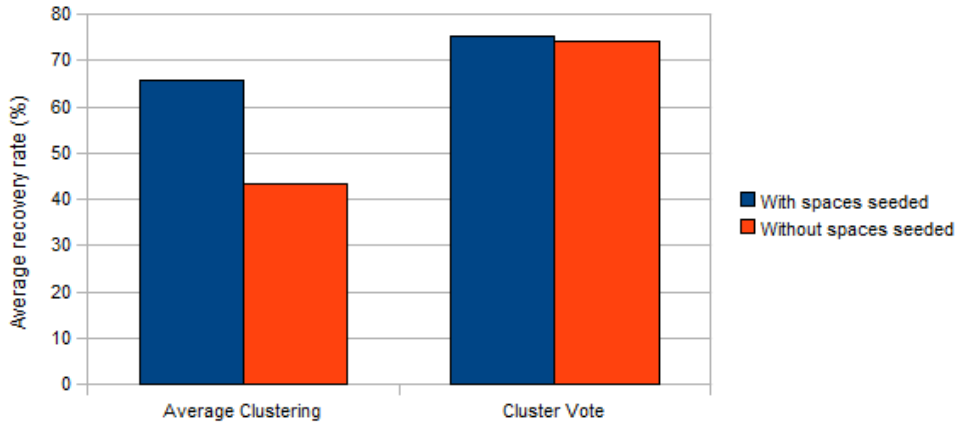

Figure 5.7: Text recovery rate of cluster voting (dataset A)

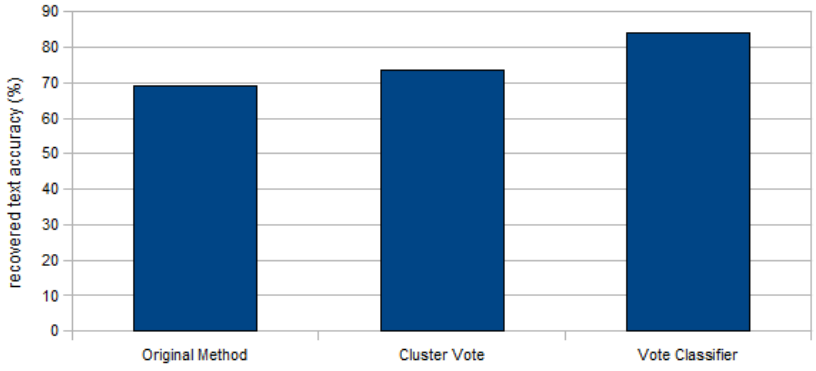

Figure 5.8: Text recovery rate for voted training set (dataset B)

Chapter 6

Language Model

After an initial attempt at recovering the keystroke sequence has been made by the clustering algorithm, the next stage of the attack is to run a spellcheck over the recovered text to improve its accuracy. These improved results are then used to form a set of labelled training examples for a supervised classifier discussed in the next chapter. The spellcheck first selects the top 20 closest matching words for each word in the recovered text and then applies a word trigram language model to find the most likely sequence of words. To improve the supervised classifier's accuracy a number of iterations are performed whereby the classifier is trained, used to classify the recorded set of keystrokes and the results are then fed back into the language model to be corrected and form a new set of labelled training data.

6.1 Spellcheck

Since the language model and classifier are run a number of times over the same set of recorded keystrokes, we can keep track of which classification errors are most likely to occur and take this into account on the next iteration. To achieve this, the confusion matrix of the previous classifier is used to provide a probability distribution of each key being predicted given the correct key. Section 7.1 provides a detailed explanation of how this confusion matrix is found so for now assume that it is available. On the first iteration, since a confusion matrix will be unavailable, the probability matrix is simply initialized with a probability of 0.5 that the classified key is the correct key and the remaining probability is distributed uniformly across the rest of the keys. On subsequent iterations the matrix is calculated from the confusion matrix using equation 6.1 where y is the recovered key, x is the correct key and N_{xy} is the number of times

the correct key x was classified as key y. However, before this calculation is performed every combination of correct and classified keys have their counts incremented by 0.1 to account for the sparsity of the confusion matrix with such a small set of data.

$$E_{i,j} = p(y = i|x = j) = \frac{N_{x=j,y=i}}{N_{x=j}} \tag{6.1}$$

Once the probability distributions have been calculated, the recovered keystroke sequence is split up into individual words. Each of these words is then compared with a dictionary and the 20 most probable spelling corrections are found. These are determined by treating the letters as independent and taking the product of the conditional probabilities of each letter in the recovered word given the corresponding letter in the dictionary word (see equation 6.2).

$$p(Y|X) = \prod_i p(Y_i|X_i) = \prod_i E_{y_i,x_i} \tag{6.2}$$

The spellcheck assumes that words in the recovered text are all properly partitioned by whitespace. As we have seen in the previous section, the space key is indeed well separated from the other keys during clustering but when a mistake is made it can cause serious problems for the language model. One issue that arose during experiments was that when a space key had been misclassified, it resulted in a string of characters longer than any word in the dictionary. This meant no suggestion could be given for that particular word. One option was to leave the word unchanged and assign it a probability of 1. However, as we will see in the next chapter, we select training examples from those words which are changed the least by this spellchecking stage so leaving it unchanged would be problematic. An alternative approach was used which was to split the string into two words by changing the central most character to a space key.

6.2 Word Frequencies

Once a set of spelling suggestions and their corresponding probabilities had been generated for each word, a second order HMM could be constructed to find the most probable sequence of words (see figure 2.4). In their original attack, [35] stated that they used an adaptation of the HMM outlined in [33]. Whilst this model was designed for part-of-speech tagging it presents a useful means of representing individual sentences and also incorporates a new method of smoothing for dealing with the sparse

data problem. In their model, they represented the sequence of words as observed nodes and the sequence of corresponding word tags as hidden nodes. Since each hidden node in a second-order HMM is conditioned on the two previous hidden nodes, they then prepended two new tag symbols, NONE and SOS, to the beginning of the tag sequence to signify the start of the sentence. This allowed for a more sensible probability distribution on the first two real tags rather than using unigram and bigram statistics. To overcome the data sparsity problem, the state transition probabilities used variable weighting. For a tag τ_x and state sequence $t_a t_b t_c$ this probability was given by

$$P(\tau_x = t_c | \tau_{x-1} = t_b, \tau_{x-2} = t_a) = k_3 \frac{N_3}{C_2} + (1-k_3)k_2 \frac{N_2}{C_1} + (1-k_3)(1-k_2)\frac{N_1}{C_0} \quad (6.3)$$

N_1 = number of times t_c occurs

N_2 = number of times sequence $t_b t_c$ occurs $\qquad k_2 = \frac{log(N_2+1)+1}{log(N_2+1)+2}$

N_3 = number of times sequence $t_a t_b t_c$ occurs $\qquad k_3 = \frac{log(N_3+1)+1}{log(N_3+1)+2}$

C_0 = total number of tags that appear

C_1 = number of times t_b occurs

C_2 = number of times sequence $t_a t_b$ occurs

The variables k_2 and k_3 effectively reweight the equation depending on the existence of bigram and trigram statistics in the training corpus. This model was implemented for the project but where the original model represented tags and words, these were substituted with words and attempted classifications of words respectively. The implementation of this method was performed in three stages. Firstly the language statistics required for such a model were gathered. Secondly, sentences were extracted from the recovered text and the language statistics were filtered to reduce processing time. Finally, a second-order viterbi algorithm was used to find the most probable set of words.

6.2.1 Collecting language statistics

To collect word frequency statistics the Cambridge Statistical Language Modeling Toolkit [7] was used. This toolkit provides the text2wngram utility which generates an

ordered list of word n-tuples along with their number of occurrences for all non-zero occurrences in a given input text. Before generating these statistics, the project's reference corpus (see section 3.1.3) was processed so that each sentence was prepended with the string "<NONE> <SOS>" and all punctuation except apostrophes was removed. Unigram, bigram and trigram statistics were then taken from this new text. A first attempt at importing the trigram data into Matlab was to use a three-dimensional sparse array but unfortunately this proved too large to fit into memory. This meant an alternative approach was implemented using 4 one-dimensional cell arrays where element p of the first 3 arrays stored the first, second and third words of the p^{th} trigram and the last array stored the p^{th} trigram's total number of occurrences. Matlab was able to generate this data structure but as we'll discuss in section 6.3, it was a poor choice in terms of performance. The unigram and bigram statistics were also included in this same data structure by assigning the "<NONE>" symbol to the first or first and second trigram words where appropriate.

6.2.2 Extracting sentences

Once word statistics had been collected, the recovered text was split into sentences. Since fullstops have a low occurrence rate, they were often misclassified as other keys resulting in extremely long sentences. To reduce the computational complexity of analyzing these long sentences, a maximum length of 100 characters was enforced by splitting any sentences over this threshold. The sentences were then prepended with two control symbols. If the sentence was the first half of a split sentence or a whole sentence then it was prepended with "<NONE> <SOS>" whereas if the sentence was from the second half of a split it was prepended with "<NONE> <NONE>". Since the unigram frequencies had been added to the word statistics prefixed with two <NONE> symbols, this effectively places a unigram prior on the first word of sentences that are from the second half of a split.

Each sentence was then processed in turn. The lists of spelling suggestions for every word in the sentence were concatenated with any duplicates removed, giving a unique set of vocabulary that may occur in the sentence. Before each sentence was processed by the viterbi algorithm, a new set of word statistics was generated by extracting trigrams which only contained words from the vocabulary and the control symbols "<NONE>" and "<SOS>". This greatly reduced the time required to search for the correct statistic. Finally, an observation matrix was constructed whereby the

rows of the matrix represented each word position in the sentence and the columns of the matrix represented each word in the sentence vocabulary. The spelling suggestion probabilities were then used to populate the matrix and all other values set to 0. The observation matrix, vocabulary, and filtered word statistics were then passed to the viterbi algorithm to find the most likely sequence of words for the sentence.

6.2.3 Second-order Viterbi algorithm

To implement this part of the project, the method outlined in [12] for extending the viterbi algorithm to second-order was used. They argued that for a state sequence X and observation sequence Z, one can introduce a combined state sequence Y which consists of pairs of states from X where $y_1 = x_1$ and $y_i = x_{i-1}x_i$ for $2 \leq i \leq K$. This then gives,

$$p(y_1) = p(x_1)$$

$$p(y_2|y_1) = p(x_2|x_1)$$

$$p(y_i|y_{i-1}) = p(x_i|x_{i-1}, x_{i-2}) \qquad 3 \leq i \leq K$$

therefore for a partial state sequence $X_i = (x_1, x_2, ..., x_i)$ and a partial observation sequence $Z_i = (z_1, z_2, ..., z_i)$ and $3 \leq i \leq K$ we have

$$P(X_i, Z_i) = P(X_{i-1}, Z_{i-1})p(y_i|y_{i-1})p(z_i|x_i) \tag{6.4}$$

This means that to find the maximum $P(X_i, Z_i)$ for each y_i we must:

- store the maximum $P(X_{i-1}, Z_{i-1})$ for each y_{i-1}
- extend each of these probabilities to every y_i by using equation 6.4
- select the maximum $P(X_i, Z_i)$ for each y_i

The algorithm simply repeats these three steps, incrementing i on each iteration and remembers the path taken. Once the last node is reached, the algorithm backtracks through the recorded path to find the best sequence. The probability $p(z_i|x_i)$ is provided by the spellcheck suggestion probabilities and $p(y_i|y_{i-1})$ is calculated from the word trigram statistics using equation 6.3.

6.3 Model Performance

Despite filtering the word trigram statistics on the sentence vocabulary, the language model still took much longer than the implementation described in [35]. They stated that the time required to process their attack, using 3 iterations of the language model, was approximately 30 minutes using a Pentium IV 3.0GHz desktop computer with 1GB RAM. However, for the maximum sentence size of 100 characters (approximately 16 words), the project's viterbi implementation takes around 11 minutes to determine the best sequence using an AMD Turion 2.0GHz laptop with 4GB RAM. This means that for dataset A, the language model takes a total of 3 hours to complete a single iteration, bringing the total attack time required to just over 9 hours for 3 iterations.

In [35] they stated that the second-order viterbi algorithm's complexity is $O(TN^3)$ where T is the length of the sentence and N is the number of possible values for each hidden variable but since they were only selecting the top M spelling suggestions, this reduced the complexity to $O(TM^3)$ with a value of $M = 20$ used in their experiments. It would appear that the reason for the large time difference is in the way the top M suggestions are processed. The project's implementation (see algorithm 6.1) stores the sentence's vocabulary in an array and then at each position in the sentence it loops over the vocabulary of words and checks to see if that particular word is one of the possible suggestions for that position in the sentence. However, this method is extremely inefficient because although each check takes a fraction of a second to perform, there are three of these loops nested under each other. A more efficient method would have been to store the spelling suggestions in a 2D array where each row represents the position in the sentence and each column represents the M spelling suggestions. Then rather than searching over the entire vocabulary and rejecting those that aren't suggestions, only the actual spelling suggestions for that sentence position could be looped over.

Unfortunately, there was not enough time to implement and thoroughly test these optimizations in the project but whilst the language model takes a long time to execute, the results it produces will be identical to an optimized version so the accuracy of the attack will remain unaffected.

Algorithm 6.1 Pseudocode of main loop in viterbi implementation

```
for pos = 3 to sentenceLength

    for word3 = 1 to vocabularyCount


        if (observationMatrix(pos,word3) == 0) then continue;

        for word2 = 1 to vocabularyCount


            if (observationMatrix(pos-1,word2) == 0) then continue;

            for word1 = 1 to vocabularyCount


              if (observationMatrix(pos-2,word1) == 0) then continue;

              #....calculate probability and check if
              # it's the best path so far....


            end
        end
    end
end
```

Chapter 7

Supervised Classification

7.1 Training a classifier

Once the recovered text has been corrected by the language model, a set of labelled training data can then be extracted with which to train a supervised classifier. To determine which keys to assign to the training set, the method used in [35] was used which compared the corrected text with the original recovered text and any words that had more than 75% of their characters left unchanged were used as training examples. Each letter from these corrected words and their corresponding keystroke features were then used to train a supervised classifier. In this project, a probabilistic neural network was used for classification using Matlab's newpnn() function and once it had been trained with the newly created training set, it was then used to classify the entire dataset. The key feature in the [35] model is that once this classification has been performed, the results are then fed back into the language model, corrected and a new training set is generated with which to retrain the classifier. This retraining process is performed repeatedly until no improvement in accuracy is obtained.

To generate the confusion matrix for the spellcheck module, each time a training set is extracted the training words are compared with their original spelling in the recovered text. This means that the next time a spellcheck is performed, the corrections that needed to be made on the last set of training words will have greater probability than random corrections.

7.2 Overall attack performance

Figure 7.1 shows the results of the overall attack on datasets A and B. For the initial text recovery stage, the votes of confidence classifier was used (section 5.3) and then a further 2 iterations of the language model were performed. The maximum text recovery rates for both datasets were 81.47% for dataset A and 91.26% for dataset B.

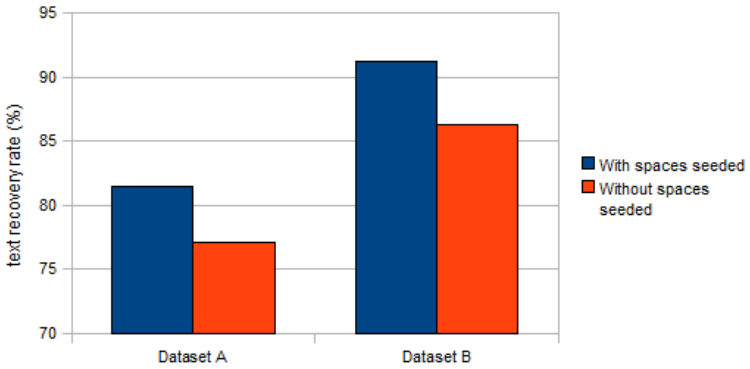

Figure 7.1: Final results of attack model

As previously discussed in section 4.2, the keyboard used to record dataset B seems to produce keystrokes sounds which are more readily distinguishable which is why there is a large difference between the two sets of results. As we have seen in section 3.3, properties of the keystroke audio recording such as typing rate and background noise can also have a large impact on the performance of the attack. Unfortunately the dataset used by [35] is not publically available and so a direct comparison between models could not be performed.

Chapter 8

Conclusion

The acoustic eavesdropping attack outlined in [35] has been successfully reimplemented with text recovery accuracies similar to those stated in the original paper. A method for collecting accurate labelled datasets was developed and a novel error function for determining thresholding values during keystroke extraction was demonstrated. Some of the problems faced by unsupervised keystroke extraction methods have also been explored and it was found that a mixture of FFT and cepstrum features out-perform either set of features individually. Finally, it was shown that running the clustering algorithm over multiple iterations and taking a vote on the recovered keystroke sequence greatly increases the accuracy of the results.

Since these types of acoustic attack are still in their infancy, there doesn't yet exist a common dataset on which to test competing models. Experiments so far have recorded a user copying a piece of sample text onto the computer and so the use of a standard English corpus has been a sufficient source of statistics. However, it has been shown [30] that written text samples poorly reflect the actual keystroke statistics of user input. Future work in this field would be greatly assisted by a large, publically accessible dataset on which to test competing attack models and which provides a more realistic set of data to attack.

The malicious use of surveillance for the purpose of industrial espionage is a growing threat to businesses. The German government recently stated that "German companies were losing around €50bn (£43bn) and 30,000 jobs to industrial espionage every year" [9]. Indeed, the United Kingdom's Security Service (MI5) currently lists espionage alongside terrorism and weapons of mass destruction as one of the primary threats against national security [21]. Clearly, the practical limitations of these types of attack must be explored so that appropriate defenses can be identified.

Bibliography

[1] Simple python keylogger. http://sourceforge.net/projects/pykeylogger/, June 2010.

[2] Edwin A. Abbott. *Flatland: A Romance of Many Dimensions*. Seely & Co., 1884.

[3] Agrawal R. Asonov, D. Keyboard acoustic emanations. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, pages 3 – 11, May 2004.

[4] Yigael Berger, Avishai Wool, and Arie Yeredor. Dictionary attacks using keyboard acoustic emanations. In *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, pages 245–254, New York, NY, USA, 2006. ACM.

[5] B. Bogert, M. Healy, and J. Tukey. The quefrency alanysis of time series for echoes: Cepstrum, pseudo-autocovariance, cross-cepstrum and saphe cracking. In *Proc. Symp. on Time Series Analysis*, pages 209–243, 1963.

[6] Leo Breiman. Bagging predictors. *Mach. Learn.*, 24(2):123–140, 1996.

[7] Philip Clarkson. Cambridge statistical language modeling toolkit. http://svr-www.eng.cam.ac.uk/ prc14/toolkit.html, June 1999.

[8] Australian Broadcasting Commission. Science news corpus. http://www.abc.net.au/, 2006.

[9] Kate Connolly. Germany accuses china of industrial espionage. The Guardian newspaper, July 2009.

[10] Jean-François Dhem, François Koeune, Philippe-Alexandre Leroux, Patrick Mestré, Jean-Jacques Quisquater, and Jean-Louis Willems. A practical implementation of the timing attack. In *CARDIS '98: Proceedings of the The Inter-*

national Conference on Smart Card Research and Applications, pages 167–182, London, UK, 2000. Springer-Verlag.

[11] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic analysis: Concrete results. In *CHES '01: Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems*, pages 251–261, London, UK, 2001. Springer-Verlag.

[12] Y He. Extended viterbi algorithm for second order hidden markov process. In *Markov Process, Proc. 9 th IEEE Conf. Pattern Recognition*, pages 718–720, 1988.

[13] Daniel Jurafsky and James H. Martin. *Speech and Language Processing: An introduction to natural language processing*. Prentice Hill, 2000.

[14] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *CRYPTO '96: Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, pages 104–113, London, UK, 1996. Springer-Verlag.

[15] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *CRYPTO '99: Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, pages 388–397, London, UK, 1999. Springer-Verlag.

[16] H.J. Landau. Sampling, data transmission, and the nyquist rate. *Proceedings of the IEEE*, 55(10):1701 – 1706, oct. 1967.

[17] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 1 edition, July 2008.

[18] Mathworks. Specgram function. http://www.mathworks.com/access/helpdesk-r13/help/toolbox/signal/specgram.html, 2010.

[19] Rita Mayer-Sommer. Smartly analyzing the simplicity and the power of simple power analysis on smartcards. In *CHES '00: Proceedings of the Second International Workshop on Cryptographic Hardware and Embedded Systems*, pages 78–92, London, UK, 2000. Springer-Verlag.

[20] Dominic Mazzoni. Audacity editing software. http://audacity.sourceforge.net/, June 2010.

[21] UK Security Service (MI5). The threats. https://www.mi5.gov.uk/output/the-threats.html, 2010.

[22] Imperial College Mike Brookes, Department of Electrical & Electronic Engineering. Voicebox: Speech processing toolbox for matlab. http://www.ee.ic.ac.uk/hp/staff/dmb/voicebox/voicebox.html, 2010.

[23] Steven J. Murdoch. Hot or not: revealing hidden services by their clock skew. In *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, pages 27–36, New York, NY, USA, 2006. ACM.

[24] Kevin Murphy. Hidden markov model toolbox for matlab. http://www.cs.ubc.ca/ murphyk/Software/HMM/hmm.html, June 2005.

[25] Nokia. Qt application framework. http://doc.qt.nokia.com/, 2010.

[26] Jean-Jacques Quisquater and David Samyde. Electromagnetic analysis (ema): Measures and counter-measures for smart cards. In *E-SMART '01: Proceedings of the International Conference on Research in Smart Cards*, pages 200–210, London, UK, 2001. Springer-Verlag.

[27] S. Shieh S. Huang and J. D. Tygar. Keyboard acoustic emanations revisited. http://www.cs.berkeley.edu/ tygar/papers/Keyboard-Acoustic-Emanations-Revisited/tiss.preprint.pdf, 2010.

[28] Adi Shamir and Eran Tromer. Acoustic cryptanalysis: On nosy people and noisy machines. Eurocrypt 2004 Rump, May 2004.

[29] Dawn Xiaodong Song, David Wagner, and Xuqing Tian. Timing analysis of keystrokes and timing attacks on ssh. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, pages 25–25, Berkeley, CA, USA, 2001. USENIX Association.

[30] R. William Soukoreff and I. Scott MacKenzie. Input-based language modelling in the design of high performance text input techniques. In *Graphics Interface*, pages 89–96, 2003.

[31] William Soukoreff and Scott MacKenzie. Keycapture. http://www.dynamicnetservices.com/w̃ill/academic/textinput/keycapture/, July 2003.

[32] Fred A. Stahl. A homophonic cipher for computational cryptography. In *AFIPS '73: Proceedings of the June 4-8, 1973, national computer conference and exposition*, pages 565–568, New York, NY, USA, 1973. ACM.

[33] Scott M. Thede and Mary P. Harper. A second-order hidden markov model for part-of-speech tagging. In *In Proceedings of the 37th Annual Meeting of the ACL*, pages 175–182, 1999.

[34] Wikipedia. British and american keyboards. http://en.wikipedia.org/wiki/British-and-American-keyboards, August 2010.

[35] Li Zhuang, Feng Zhou, and J. D. Tygar. Keyboard acoustic emanations revisited. In *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security*, pages 373–382, New York, NY, USA, 2005. ACM.