Dictionary Attacks Using Keyboard Acoustic Emanations

Yigael Berger
Dept. of Computer Science
Tel Aviv University
Ramat Aviv 69978, ISRAEL
yigael.berger@gmail.com

Avishai Wool
School of Electrical Engineering
Tel Aviv University
Ramat Aviv 69978, ISRAEL
yash@acm.org

Arie Yeredor
School of Electrical Engineering
Tel Aviv University
Ramat Aviv 69978, ISRAEL
arie@eng.tau.ac.il

ABSTRACT

We present a dictionary attack that is based on keyboard acoustic emanations. We combine signal processing and efficient data structures and algorithms, to successfully reconstruct single words of 7-13 characters from a recording of the clicks made when typing them on a keyboard. Our attack does not require any training, and works on an individual recording of the typed word (may be under 5 seconds of sound). The attack is very efficient, taking under 20 seconds per word on a standard PC. We demonstrate a 90% or better success rate of finding the correct word in the top 50 candidates identified by the attack, for words of 10 or more characters, and a success rate of 73% over all the words we tested. We show that the dominant factors affecting the attack's success are the word length, and more importantly, the number of repeated characters within the word. Our attack can be used as an effective acoustic-based password cracker. Our attack can also be used as part of an acoustic long-text reconstruction method, that is much more efficient and requires much less text than previous approaches.

Categories and Subject Descriptors

K.6.5 [**Security and Protection**]: Authentication; H.5.5 [**Sound**]: Signal analysis, synthesis, and processing

General Terms

Algorithms, Security

Keywords

Keyboard acoustics, Dictionary attacks, Password cracking

1. INTRODUCTION

1.1 Background

The study of signals emanating from electronic or mechanical devices goes back a long way in history. In the mid 1950's this subject gained immense public interest and awareness with the United States government coming out with its TEMPEST program [13], aimed at preventing the possibility of exploiting these compromising emanations, leaking from devices handling sensitive information. More recently, extracting information out of various types of emanations was demonstrated: electromagnetic emanations leaking video information [6], optical emanations such as in [7] and acoustic emanations from mechanical devices such as dot-matrix printers [3].

This paper deals with acoustic signals emanating from a PC keyboard in reaction to a person typing on it, and is a step further in the direction set by the recent works of [1] and [14]. The main observation, and the reason why keyboard acoustic emanations leak information, is that different keys on the keyboard make different click sounds.

1.2 Related Work

Attacks against emanations caused by human typing have attracted interest in recent years. In particular, the seminal works of [1] and [14] showed that keyboard acoustic emanations do leak information that can be exploited to reconstruct the typed text.

[1] used tagged recordings of a person typing on a keyboard, to train a neural network that would recognize subsequent keystrokes. They extracted *fft* coefficients out of the press segments of keystrokes and used them as features for comparison. Training a back-propagation neural net with 100 training samples per each of the 30 keys they were able to demonstrate an 80% recognition rate. They showed that performance degrades when applying the trained net on a different keyboard or person. They also showed that their method can be applied to other push-button devices such as telephone and ATM pads. However, the reliance on tagged samples significantly limits the scope of the attack.

Subsequently, [14] suggested a method to uncover the typed text without tagged training samples. In this work, Cepstrum [11] features were preferred over *fft* coefficients. Their attack is split into two phases, with the first being the *Unsupervised Training* phase, and the second being the *Recognition* phase that is based on the outcome of the first. Their method requires about 10 minutes of recording and roughly 30 minutes of computation to uncover up to 96% of the typed text.

The earlier work of [12] shows that human typing patterns leak information via timing information, and that this timing is noticeable even through SSH encryption. Inter-keystroke timing is also available from keyboard acoustic

CCS'06, October 30–November 3, 2006, Alexandria, Virginia, USA.

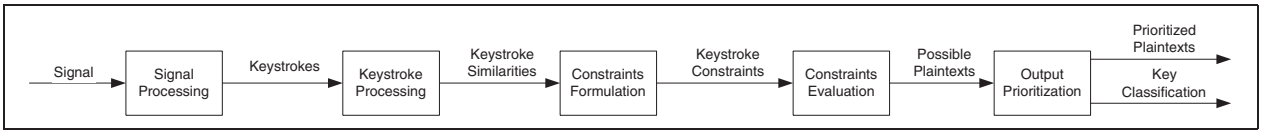

Figure 1: Attack stages.

emanations, so this type of attack can be combined with ours.

1.3 Contribution

Beyond the basic observation that different keystrokes produce different sounds, we make two new observations that are central to our attack: Firstly, the sounds that keystrokes make correlate to their physical positioning on the board. Specifically, keys such as `Q`, `W` and `E`, that are located close to one another, sound more alike than keys positioned far apart like `Z` and `P`. Secondly, we discovered that if we work at a granularity of *words*, rather than individual keys, we can exploit the statistical properties of the language in addition to the properties of the signal.

Based on these observations, we present a dictionary attack that is based on keyboard acoustic emanations. We combine signal processing and efficient data structures and algorithms, to successfully reconstruct single words of 7-13 characters from a recording of the clicks make when typing them on a keyboard. Our attack does not require any training, and works on an individual recording of the typed word (may be under 5 seconds of sound).

The attack is very efficient, taking under 20 seconds per word on a standard PC. We demonstrate a 90% or better success rate of finding the correct word in the top 50 candidates identified by the attack, for words of 10 or more characters, and a success rate of 73% over all the words we tested. We show that the dominant factors affecting the attack's success are the word length, and more importantly, the number of repeated characters within the word. Along the way, we tested various signal processing primitives, and discovered that the simple cross-correlation primitive is more effective in this attack than other known methods (like *fft*, and *Cepstrum*) used by previous authors.

Our attack can be used as an effective acoustic-based password cracker. The attack can also be used as part of an acoustic long-text reconstruction method, that is much more efficient and requires much less text than previous approaches.

Organization: The next section gives a general overview of the whole attack. Section 3 describes the signal processing we used in the attack. Section 4 describes the combinatorial constraint generation methods, and Section 5 describes our constraint satisfaction algorithms and their implementation. In Section 6 we analyze the attack's performance, and in Section 7 we present our conclusions and suggestion for further research.

2. ATTACK OVERVIEW

Our attack takes as input an audio signal containing a recording of a single word typed by a single person on a keyboard, and a dictionary of words. We assume that the typed word is present in the dictionary. The aim of the attack is to reconstruct the original word from the signal. We concentrate on handling extremely short audio signals containing a single word, with seven or more characters long. This means that the signal is only a few seconds long. It is well known that such short words are often chosen as a password (cf. [5]). The attack does not require any training.

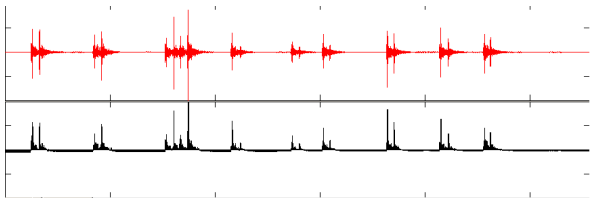

Figure 2: Example of a bare signal (top) and its corresponding energy bins (bottom) for the keystrokes of the word "difference".

Our attack comprises of several stages: (see Figure 1) (i) Signal Processing and Feature Extraction; (ii) Keystroke Processing; (iii) Constraint Formulation; (iv) Constraint Evaluation; and (v) Outcome Prioritization. Below we provide a general overview of each stage. In Sections 3, 4, and 5 we provide the full details.

2.1 Signal Processing

We begin by processing the signal in order to separate and extract the keystrokes from it (see Figure 2). Assume that the signal contains an N characters long word. Note that each keystroke produces two separate sound segments, as noted in [1, 14], generated by the press of the key button and its release. Let PRESS_i (RELEASE_i) denote the i'th key press (release) in the signal. The output of the signal processing stage consists of the two arrays of signal segments, PRESS and RELEASE.

2.2 Keystroke Processing

A basic capability we need is a method to calculate the similarity between each pair of keystrokes. What we demonstrate is that a good similarity metric not only tells us how similar two keystrokes sound, but also lets us deduce information about the keys' physical proximity on the keyboard. Specifically, for a metric sim, if $sim(K_i, K_j) > sim(K_i, K_k)$ then with a significant amount of confidence we can say that K_i and K_j are positioned more closely on the keyboard than K_i and K_k. Without this property, it would not have been possible to employ our method on such short signals.

There are several possible methods of calculating a similarity metric between two acoustic signals. As part of our study we tested three of these metrics (see Section 3.3). Somewhat surprisingly, we found that the best performance was obtained using the simple cross-correlation metric rather than *fft* or Cepstrum.

Type	Notation	Meaning
EQ	$=$	$K_i = K_j$ means that the i'th keystroke and the j'th keystroke stem from the same key on the keyboard.
ADJ	$\simeq$	$K_i \simeq K_j$ means that the j'th keystroke stems from a key that is adjacent to the key which the i'th keystroke stems from. For example, Q $\simeq$ W but not Q $\simeq$ E since E is located two positions away from Q on a QWERTY keyboard.
NEAR	$\sim$	$K_i \sim K_j$ means that K_i and K_j are at most two keys apart on the keyboard, e.g., keys *NEAR* G include R, D, N, J, etc.
DIST	$\nsim$	Distant keys are those that are not *NEAR* to each other.

Table 1: The four constraint types

Once we have a similarity metric, we measure the similarity of each PRESS_i to every other press, and the same is done for every RELEASE_i. This produces two $N \times N$ matrices of key-to-key similarities. We then combine the two matrices into a single $N \times N$ similarity matrix M_{ij}. We evaluated 5 possible methods for combining the PRESS similarities with the RELEASE similarities. We found that using an unweighted average outperforms the other possibilities we examined, in being resilient and sustaining good performance across different keyboards.

2.3 Constraint Formulation

The similarity matrix M is used to formulate constraints on the recorded word. A constraint is a binary operator expressing a relation on a pair of keystrokes. For example, the constraint $K_i = K_j$ means that the i'th and j'th keystrokes stem from the same key on the keyboard. Note that the constraint does not state what the actual key is, but only that the examined text complies with the given condition. In this manner we define four types of constraints that are listed in Table 1.

A given word produces a specific set of constraints—but the opposite does not necessarily hold; a specific set of constraints may be true of several words. Consider the word "help". Under ideal conditions, this word produces the following constraints: $K_1 \nsim K_2$, $K_1 \nsim K_3$, $K_1 \nsim K_4$, $K_2 \nsim K_3$, $K_2 \nsim K_4$, $K_3 \simeq K_4$. However, the same constraints are also true for the words "iraq", "nose", "path" and more. The full specification of the key relations can be found in appendix A.

2.4 Constraint Evaluation

We use the similarity matrix M to infer as many correct constraints as possible, and use them to postulate on the value of the text. Assuming that we know the length of the text word, N, and that all the constraints that we inferred are correct, we can evaluate the constraints: Go over all possible dictionary words of length N and output all those that match the constraints. This search can be made very efficient using suitable data structures.

However, inferring constraints from the similarity matrix is inherently inaccurate. Any inference policy will either fail to infer all the possible constraints, or produce false ones, or both. It is important to understand that even a single false constraint is enough to cause us to discard the correct word.

Our first method of reducing errors is the policy we use to infer constraints. The success of such a policy is measured by two metrics (for each constraint type):

- *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.
- *Recall* measures the fraction of true constraints with respect to the total number of possible constraints in that category.

There is always a tension between being these two metrics, forcing us to balance between coming up with as many of the true constraints as possible while being as precise as possible.

Our preferred policy of formulating constraints is called the ***BestFriendsPickPolicy***. We found that it performs well in both the *Precision* and *Recall* of the produced constraints, per constraint type (*EQ*, *ADJ*, *NEAR*, *DIST*). The *BestFriendsPickPolicy* is specified in Section 4.1.

2.5 Dealing with False Constraints

The next method we use to mitigate the false constraints that are inferred is by using constraint combinations. The main idea is to select many different subsets ("combinations") of constraints and evaluate each combination. If enough constraints are correct, then many combinations will be consistent with the correct word.

However, if implemented naively, this method can be extremely inefficient computationally. Let C denote the set of constraints extracted out of the similarity matrix, using the constraint inference policy. For this given C, there are $2^{|C|}$ possible constraint combinations. Our method may produce a few dozens of constraints for a 7-13 character word: we clearly face a combinatorial explosion in the number of possible combinations

To overcome the combinatorial explosion, we *randomly* choose a relatively small collection of the possible combinations. We have empirically found that about 1000 combinations usually suffice. Section 5.3 details how we choose the collection, and evaluate the effectiveness of our choice.

Having chosen the combinations of constraints, each combination c is evaluated against the possible words in the dictionary. This yields a list of possible words L_c that all conform with the constraints of combination c.

2.6 Outcome Prioritization

Our next goal is to produce a unified list U of candidate words, prioritized based on the L_c lists produced for the various constraint combinations. For each word w in the dictionary we count the number of combinations c for which $w \in L_c$, and sort the words in decreasing order.

Recall that any erroneous constraint will necessarily preclude the correct word from appearing in the L_c list of any

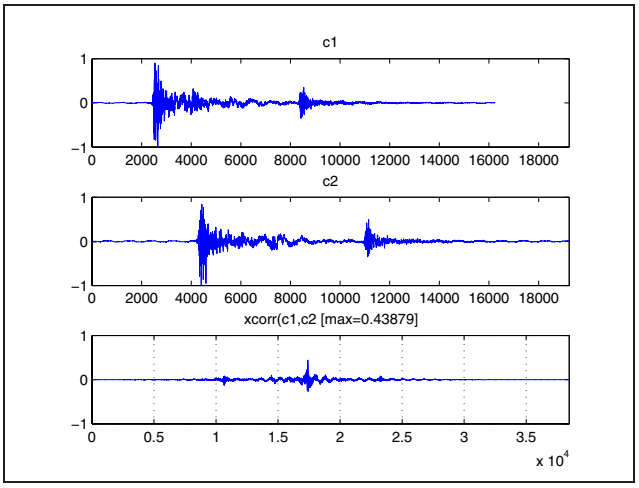

Figure 3: Two recordings of the key `C`, with their cross-correlation function.

combination c that includes this constraint. Therefore, such an L_c will include only false choices—that are randomly distributed. On the other hand, a correct combination of constraints c will include the correct word in its L_c. Therefore, after trying enough combinations, the correct word will appear near the top of the sorted unified list U.

3. SIGNAL PROCESSING

3.1 Recording

We recorded the keyboard acoustic emanations using a cheap omni-directional clip microphone, at a 44,100 Hz sampling rate. We used three different keyboards: an IBM KB-8923, a Genius K295 and an old VISION keyboard (model unknown).

3.2 Feature Extraction

The purpose of the feature extraction process is to be able to locate and compare the different keystrokes in the given input signal (recall Figure 2). This entails identifying the start and end of each keystroke, and then splitting it into the press and release segments.

We break the signal into windows of 2 milliseconds (100 samples), then sum the *fft* coefficients obtained on each window. This gives us an indication of the amount of energy contained in each window. We normalize the energy bins to values between 0 and 1, and using them, we compute the difference between each window and its predecessor, which gives us a "delta vector". Each element of the vector is an indication of a rise in energy in that time frame.

We now go through this delta vector and look for surges of energy that can account for a key press. Once we have homed on a specific peak we look for a fall followed by a resurgence of energy that fits the description of a key-release phase. The two peaks (press and release) are checked to appear within a boundary of 100 ms. If the energy in the whole press is found to be too weak, then this segment is ignored and we move on to the next peak. Figure 2 (bottom) shows the computed energy peaks.

3.3 Measuring Similarity between Keystrokes

Once we identify where each key press and release begins and ends, we can measure the similarity between each of them. There are several possible measures of similarity, so before proceeding let us first describe the properties of a good measure.

Let K_i denote the recording of some key α_i on the keyboard. A similarity measure $sim(K_i, K_j)$ is a function with real output between 0 and 1 with the following properties:

- **Adjacency**. We would like $sim(K_i, K_j) > sim(K_i, K_k)$ to be true if α_i and α_j are physically closer to one another on the keyboard than α_i and α_k are. This criterion is crucial to the success of our method.
- **Symmetry**. $sim(K_i, K_j) = sim(K_j, K_i)$.
- **Reflexivity**. A signal K_i should obviously come out most similar to itself. Ideally $sim(K_i, K_i) = 1$.

Note that **transitivity** is not a requirement, otherwise we would end up with the two farthest keys on the keyboard being similar to one another.

A "good" similarity measure should also have two more properties: **a) universality**, i.e., that it performs well across different keyboards, and **b) computational efficiency**.

There are several known methods for measuring the similarity between two signals, using two major approaches: **a)** methods that rely on the **shape** of the signals, viewing the signal in its time-domain representation and, **b)** methods that concentrate on the **spectral** content, looking at the frequency-domain representation of the signal. We evaluated the following candidates: (1) The simple cross-correlation function (time domain); (2) *fft* coefficients (frequency domain); and (3) Cepstrum Coefficients (quefrency[1] domain). Other possible measures which we did not try include: Wavelets Analysis, Zero-Crossing/Wave Fluctuation rates, EMD[2] [8], and more.

3.3.1 A Similarity Measure based on the Cross Correlation Function

One of the basic primitives in signal processing is the cross-correlation function [9]. The cross-correlation function operates on signals in their time-domain representation, and is computed as follows: given two digitized signals $x[\cdot]$ and $y[\cdot]$, let

$$CC[x, y, t] = \sum_k x[k] \cdot y[t + k].$$

This is, in effect, a sliding dot-product of the two signals, where signal y is shifted by t samples over x. The cross-correlation is maximized at the offset t where the two signals are most similar. We define $sim^{xcorr}(x, y) = max_t(CC(x, y, t))$ to be the similarity measure between two given signals. Figure 3 shows a plot of two signals and their cross-correlation function, clearly demonstrating the peak at the point where the shapes of the two signals match the most.

For convenience, the signals and the outcome of their cross-correlations are normalized. Thus, the similarity of a signal to itself (its auto-correlation) comes out exactly 1, as required.

[1]This terminology was chosen by the Cepstrum inventors [11] to denote a signal in a hybrid time-frequency domain.

[2]EMD is the Earth Movers Distance measure used in music research, by [8]

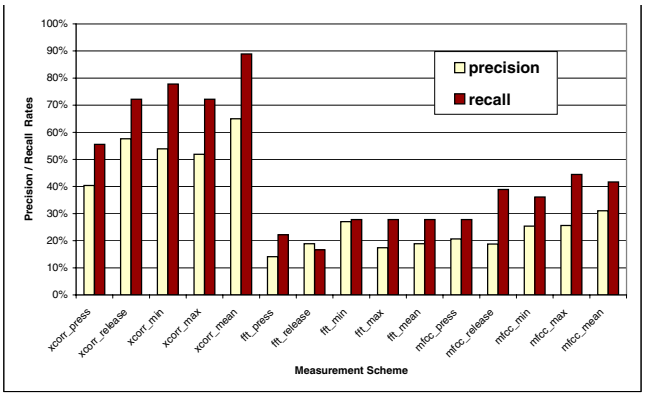

Figure 4: *Precision/Recall* rates for the various measurement schemes.

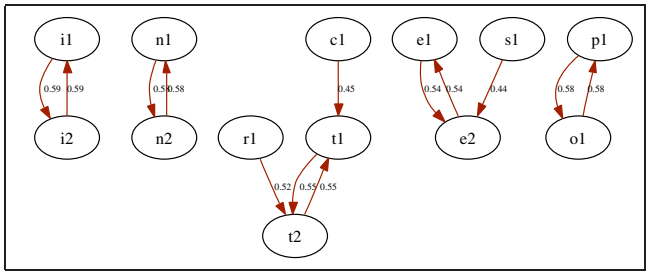

Figure 5: Top *xcorr* values for the word *interceptions*. Each key is represented by a node. An arrow from K_i to K_j indicates that K_j is the top similarity rank for K_i. The numbers on the arrows are the values of the similarity measure. Note that four out of the five best-friends loops are correct (letters *e,i,n,t*) but the fifth loop (for letters *o,p*) is erroneous.

If the two signals are M and N samples long, then computing the correlation requires $O(MN)$ operations of multiplication and addition. Since we are working on signals that are supposed to be more or less aligned in phase due to the procedure we employ in earlier stages (recall section 3.2), then there is no need to slide both signals all the way from start to end but rather only for short intervals around their start and end. In our prototype computation, we used the built-in Matlab function `xcorr` to compute the CC function.

3.3.2 A Similarity Measure based on FFT Coefficients

In essence, this method computes the Euclidean distance between the spectrum of the signals. Specifically:

1. Truncate each K_i to some fixed length L. In our case of dealing with key presses and releases, this L would normally be 2 milliseconds worth of sampling. If K_i is shorter than L, then pad it with 0.

2. Compute *fft* on each K_i. We ignore the phase information and use only the coefficients that represent the energy for each frequency.

3. Group the *fft* coefficients in equally spaced bands. Each band will be represented by the sum of the coefficients belonging to it. Let B_i denote the vector of bands for K_i.

4. Compute $M_{i,j}$ by calculating the Euclidean distance between B_i and B_j. Note that $M_{i,i} = 0$, so complement each value to 1 (0 becomes 1, 0.1 becomes 0.9, etc.)

To compute the *fft* coefficients we use Matlab's `fft` function.

3.3.3 A Similarity Measure based on Cepstrum Coefficients

This is done very much like in the section above describing the usage of the *fft* coefficients, only that here we use the Mel Frequency Cepstral Coefficients instead of the bands. This is the method used by [14]. We used the *mfcc* function provided by the Auditory Toolbox package [10].

3.3.4 Press vs. Release

So far we have listed several methods of extracting and comparing features of the given signal. Using these methods we are able to measure the similarity between all the press segments, and separately, between all the release segments. In our experiments we also considered how best to combine the press and release similarities into a single matrix M.

For each of the three measurement methods mentioned above (*xcorr*, *fft*, *mfcc*), we tested five different schemes to combine the press and the release similarities. Let sp_{ij} denote the press similarity and let sr_{ij} denote the release similarity, between K_i and K_j. The schemes we tested are: **1)** press only: $M_{ij} = sp_{ij}$; **2)** release only: $M_{ij} = sr_{ij}$; **3)** *min*: $M_{ij} = \min(sp_{ij}, sr_{ij})$; **4)** *max*: $M_{ij} = \max(sp_{ij}, sr_{ij})$; **5)** *mean*: $M_{ij} = [sp_{ij} + sr_{ij}]/2$;

min, *max* and *mean* are defined as follows: Let $sim^{press}(K_i, K_j)$ denote the similarity measure between the presses of α_i and α_j and $sim^{release}(K_i, K_j)$ is the similarity measure between the releases of α_i and α_j.

We define the similarity between K_i and K_j based on the *min* scheme to be:

$$min(sim^{press}(K_i, K_j), sim^{release}(K_i, K_j)).$$

The *max* scheme is

$$max(sim^{press}(K_i, K_j), sim^{release}(K_i, K_j))$$

and the *mean* scheme is

$$[(sim^{press}(K_i, K_j) + sim^{release}(K_i, K_j))]/2.$$

3.3.5 Selecting the Similarity Measure

We used the two quality criteria: *Precision* and *Recall* as defined in Section 2.4. Figure 4 shows a comparison of the *Precision* and *Recall* rates for all the different schemes and measurement methods we discussed. It is clear that the best scheme to use is the one using the `xcorr` function with the *mean* scheme, namely ***xcorr_mean***.

We have found that the *xcorr_mean* scheme performs well over different keyboards, word lengths and constraint types. We used *xcorr_mean* throughout the remainder of this paper.

	1	2	3	4	...	N−1	N
1	*EQ*	*EQ*	*ADJ*	*NEAR*			
2	*EQ*	*ADJ*	*NEAR*				
3	*ADJ*	*NEAR*					
4	*NEAR*						
:							
N − 1							*DIST*
N						*DIST*	*DIST*

Table 2: Constraint formulation rules.

4. CONSTRAINTS AND COMBINATIONS

4.1 Constraint Formulation

We now describe the method used to extract constraints of the various types, out of a given key-similarities matrix $M_{i,j}$ obtained in previous stages of the attack. Naturally, there are many ways to do this. Some of the methods we examined produce constraints with a very high *Precision* rate but with a low recall value, and vice versa. After much experimentation we arrived at a method that achieves a balance between the two, and performs well on different keyboards. We call it the ***BestFriendsPickPolicy***. We omit the details of the less successful alternatives.

Let $rank(i,j)$ denote the position of keystroke j in row M_{i*} of the similarity matrix (excluding M_{ii}), sorted in decreasing order. In other words, if $rank(i,j) = 1$ then K_j is the keystroke most similar to K_i.

We now introduce the notion of keystroke friends. Assume that $rank(i,j) = 1$. As discussed in Section 3.3, this indicates that K_i and K_j may lie physically close to one another on the keyboard.

Let us now look at i's rank in j's row, $rank(j,i)$. We say that i and j are *best friends* if i is also j's top rank, i.e., $rank(j,i) = 1$. If K_i and K_j are best friends, then it is reasonable for us to assume that they are derived from the same key α and we can infer the *EQ* constraint $K_i = K_j$ (see Figure 5).

The *BestFriendsPickPolicy* uses a generalization of this notion of friends, to infer keystroke constraints. In general, for a given pair of keystrokes K_i and K_j, we use $rank(i,j)$ and $rank(j,i)$ as indexes to a cell in Table 2. For example, if $rank(i,j) = 3$ and $rank(j,i) = 2$ then we infer a *NEAR* constraint $K_i \sim K_j$. On the other hand, if $rank(i,j) = N$ and $rank(j,i) = N$ then they are very dis-similar and we infer a *DIST* constraint $K_i \nsim K_j$.

Figure 6 demonstrates the performance of the *BestFriendsPickPolicy*. As we can see, the policy produces *Precision* and *Recall* rates that are roughly consistent over different keyboards. The figure clearly shows that the *Precision*/*Recall* rates are at their best for the *EQ* constraint, while the inferred *DIST* constraints have high *Precision* but low *Recall*.

Note that we do not use a predetermined threshold to classify keystrokes. Our experience shows that each keyboard and every recording results in different similarity values, thus calibrating a threshold may be non-trivial and keyboard-dependent, and may possibly require significant training data.

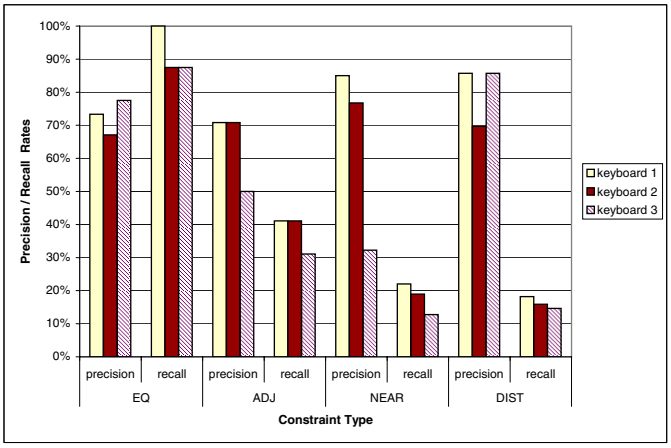

Figure 6: *BestFriendsPickPolicy* performance over 27 words.

5. THE KEY-CONSTRAINTS SATISFACTION ALGORITHM

Once we inferred some constraints we need to evaluate them against the dictionary to extract the words that are consistent with them.

We begin by describing the algorithm for evaluating a single constraint, and based on that, we specify the algorithm for evaluating multiple combinations of constraints.

5.1 Evaluating a Single Constraint

Let N denote the number of letters in the word we are trying to discover. Let Σ denote all the possible letters in the alphabet and let $\square$ denote a constraint type, $\square \in \{=, \simeq, \sim, \nsim\}$. Define $\Sigma_l^{\square} = \{\lambda | \lambda \in \Sigma, (l,\lambda) \in \square\}$ to be the set of letters that match letter l under the the given constraint type $\square$. For a full specification of the relations, see Appendix A.

For a given constraint $\xi_m = K_i \square K_j$, we run through all the possible values $l \in \Sigma$ for K_i and collect out of the dictionary words of length N whose i'th letter is l and j'th letter is in $\Sigma_l^{\square}$. By pre-processing the dictionary into a suitable search data structure, this step can be implemented very efficiently. The outcome of the evaluation of the constraint is the list of unique words collected above. Let $EVAL(\xi_i)$ denote the outcome of the evaluation of a single constraint ξ_i. The *EVAL* algorithm is shown in Figure 7.

5.2 Evaluating a Combination of Constraints

Evaluating a single constraint is only our basic building block. The next step is evaluating a combination of constraints $c = \{\xi_m\}$. This can be done in the obvious way:

$$EVAL(c) = \bigcap_m EVAL(\xi_m),$$

using the *EVAL* algorithm of 5.1.

However, our general attack requires us to produce multiple combinations of constraints, and to evaluate each combination. Doing so naively is extremely time consuming. Therefore, we have come up with an efficient way to simultaneously evaluate all these combinations of constraints, using Boolean matrix algebra. We now describe this method:

As before, let $c = \{\xi_m\}$ denote a combination of con-

Input:
1. The number of keys in the given word N.
2. Two indexes, i and j, and a constraint of the form $K_i \Box K_j$ where $\Box \in \{=, \simeq, \sim, \nsim\}$.

Let $Words(N, i, l) =$
$\{w | w \in dictionary, ||w|| = N, w[i] = l\}$

For each possible letter $l \in \Sigma$:
Let $\Sigma_l^{\Box} \leftarrow \{l' | (l, l') \in \Box\}$
Let $Possible_l \leftarrow \bigcup_{l' \in \Sigma_l^{\Box}} \left\{ Words(N, i, l) \bigcap Words(N, j, l') \right\}$
Let $Possible \leftarrow \bigcup_{l \in \Sigma} Possible_l$

Output: the set of words in *Possible*.

Figure 7: The *EVAL* algorithm for evaluating a single constraint on a single word. The set *Words*(N, i, l), of all the words of length N that have letter l in position i, is implemented using the search data structure produced from the pre-processed dictionary.

straints and let w_k denote the k'th word in the dictionary. We construct a Boolean matrix W that encodes the separate evaluation of each of the constraints, i.e., $W_{m,k} = 1$ iff $w_k \in EVAL(\xi_m)$. Let $\vec{c}$ denote the characteristic vector of the combination c, i.e., $\vec{c}_m = 1$ iff constraint ξ_m is included in the combination c, and 0 otherwise.

Using this notation, the evaluation of the combination c is the Boolean multiplication of the $\vec{c}$ with the matrix W, i.e.,

$$EVAL(c) = \vec{c} \cdot W.$$

Note that $\cdot$ is computed as the Boolean product of the two.

In a computing environment such as Matlab, Boolean multiplication can be implemented by using the normal (integer) matrix multiplication operator, which is then divided by the sum of the elements of $\vec{c}$ and truncated so that a value of 1 is produced only if the sum of terms is exactly $|c| = \sum_t \vec{c}_t$. In other words,

$$\vec{c} \cdot W = \left\lfloor \frac{\vec{c} * W}{|c|} \right\rfloor.$$

The advantage of using such matrix notation is that it lets us evaluate *multiple* combinations simultaneously, while running the *EVAL* only once per constraint, as follows. Let $\mathcal{C} = \{\rfloor_{\backslash}$ be a collection of combinations of constraints. Let Γ be the characteristic matrix of the combinations: $\Gamma_{n,m} = 1$ iff constraint ξ_m belongs to combination c_n. Then we can evaluate *all* the combinations with a single Boolean matrix operation:

$$R = \Gamma \cdot W,$$

where $R_{n,k} = 1$ iff the evaluation $EVAL(c_n)$ includes the word w_k.

To prioritize the output, for each word w_k we compute the number of combinations it appeared in: $r(w_k) = \sum_n R_{n,k}$. The final output U is the list of dictionary words sorted in decreasing order of $r(w_k)$.

5.3 Dealing with Errors

Neither the similarity measure, nor the constraint inference policy, is perfect. However, it is important to keep in mind that even a single false constraint of any kind, will preclude us from finding the right word. We deal with these errors by incorporating randomness into our attack.

Instead of relying on *all* the constraints we extracted in earlier stages, we look at multiple combinations of them. Since there may be a huge number of possible combinations, we do not generate all the possible combinations, but rather generate a random sample of combinations. Each combination is constructed by running through the list of constraints $\{\xi_m\}$ and including constraint ξ_m in the combination with probability p. A single combination is expected to include $|\xi| * p$ constraints.

When dealing with single words, 7 characters or more, we typically infer 13-17 constraints. We used $p = 0.2$ in all cases. Finally, in most cases 100-200 combinations were more than enough, with rare cases that required 1000 combinations.

Evaluating this collection of combinations is performed efficiently as described in Section 5.2, using the matrix Γ to represent the whole collection of random combinations.

Note that Γ may have a large number of rows, depending on the number of choose to use. W, on the other hand, has a column per dictionary word: in our case, around 60,000 columns. Thus, the multiplication of Γ with W may yield an enormous matrix. Luckily, the result of the Boolean multiplication is sparse (almost all the entries are 0), despite the fact that Γ itself is not sparse. There are well known advanced data structures that operate and maintain such matrices very efficiently. We used Matlab's `sparse` function for this purpose.

6. PERFORMANCE EVALUATION

6.1 Overall Success Rate

Our attack produces a sorted list U of word candidates. We measure the efficiency of our attack by marking the position of the correct word in U. The position of a word in U is called the ***Rank*** of the word, where a *Rank* of 1 is the most likely candidate. Based on the *Rank*, we calculate the frequency of the correct word appearing in the top 10, top 25, top 50, top 100 and top 500 places over all the tests we conducted.

We tested 27 words with lengths of 7-13 characters (see Table 3 in the appendix for the complete set of words). Each word was recorded on three keyboards (see Section 3.1). Each word recording was processed 7 times using different random choices.

The word corpus we used is an aggregation of the corncob wordlist [4] and the English words file from the SCOWL-6 package [2].

Figure 8 above summarizes the overall success rate of our attack, on the tested words. For example, we can see that in 73% of the tests, the correct word was located within the top 50 candidates.

6.2 Influential Factors

We tested the success of the attack as a function of two major factors:

1. Character repetitions in the word. As we saw in Sec-

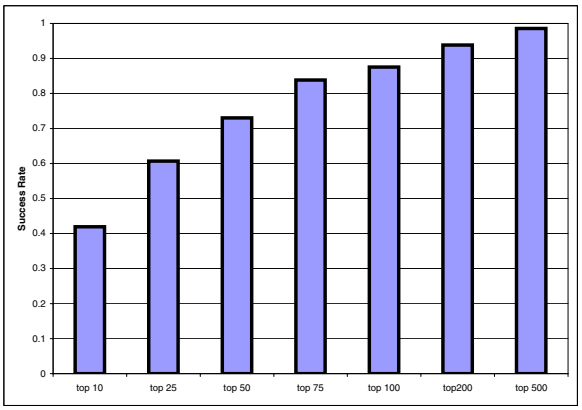

Figure 8: Overall effectiveness of the attack.

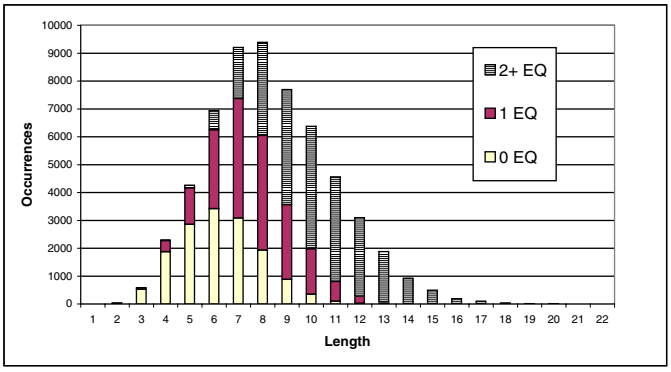

Figure 9: Distribution of word lengths in the word corpus used in the attack. The lower, middle and upper sections of each bar are proportional to amount of words with 0, 1, 2+ *EQ* in them, for that length category.

tion 4.1, *EQ* constraints have the best *Precision* and *Recall* rates among all our constraint types. Naturally, we expect that the attack will work better against words that produce more *EQ* constraints.

2. Word length. The length of the word has two effects. First, longer words produce more constraints, so we expect better results. Second, the distribution of word lengths in the dictionary is non-uniform (see Figure 9), with words of length 7,8 being the most frequent. If there are many possible candidates of length N in the dictionary then we expect the attack success to degrade. However, for length $N \geq 7$, longer words are less frequent, so again, we expect the success of the attack to improve for longer words.

6.2.1 *Character repetitions in the word*

Figure 10 shows that the success rate is strongly influenced by the number of key repetitions in the word. We can see that the success rate grows dramatically with additional *EQ* constraints: the attack finds the correct word in the top 25 at a rate of 90% for words with 2 or more *EQ* constraints.

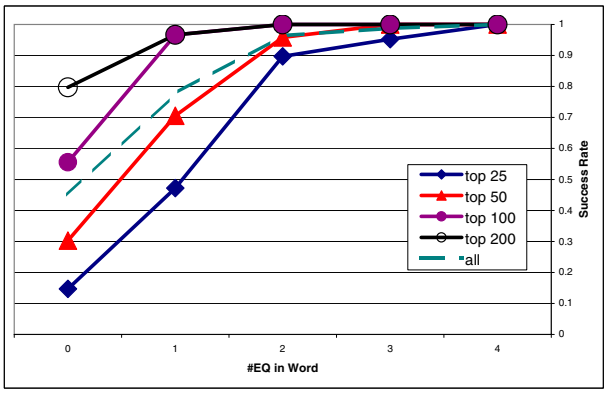

Figure 10: The success rate as a function of the number of repetitions.

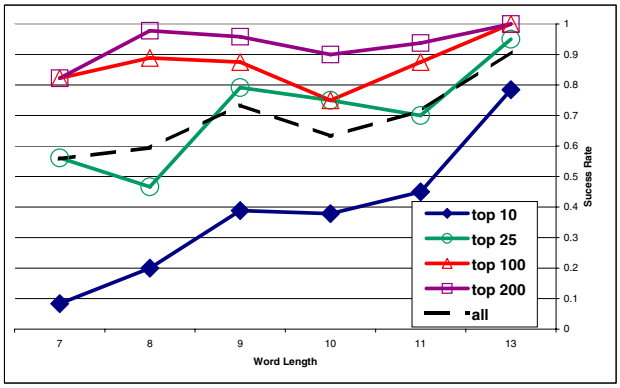

Figure 11: The success rate as a function of the word length.

A closer inspection of the data (omitted) shows that for words of length 7-9 the major improvement occurs at 2 *EQ*s. For words with length 10 and above, 1 *EQ* constraint is usually enough to place the word in the top 10 candidates.

6.2.2 *Word Length*

Figure 11 shows the dependence of the attack's success on the length of the word. The figure clearly shows that, as expected, we have better success against longer words. However, the dependence is less clear-cut than that of the number of repetitions.

7. CONCLUSIONS AND FUTURE WORK

We have presented a procedure that makes it possible to efficiently uncover a word out of audio recordings of keyboard click sounds. The effectiveness of our attack depends mainly on the properties of the hidden text, namely how long it is and how many repetitions are in it. Our methods require no training, work on signals as short as 5 seconds, and are resilient to different different keyboards.

We believe that our attack can be used as an effective

acoustic-based dictionary password cracker—and is another reason *not* to use dictionary words as passwords. However, counter-intuitively, choosing longer passwords *improves* the success of the attack: it seems that to withstand our attack, the ideal password length should be 7-8 characters.

We have some promising preliminary results which show that this attack can also be used as part of an acoustic long-text reconstruction method. The key observation is that the Space key has a very distinct sound and can be identified as a first step, allowing us to identify individual words. The current attack, in combination with inter-word statistics would then let us reconstruct whole sentences and paragraphs.

Clearly, we can integrate inter-keystroke timing information with our approach. We have looked at this direction briefly and it seems viable.

To improve the capabilities of our method for password cracking, we would need to use password dictionaries (rather than English word lists), and we would need to deal with Shift keys (for upper-case letters), punctuation marks, and digits. This seems possible since the Shift key would produce only a PRESS (or only a RELEASE) segment.

We also believe that our signal processing is quite unsophisticated, and developing more accurate models of the keyboard acoustic signals may be a fruitful direction of research for signal processing experts.

Acknowledgments

We would like to thank Miriam Furst-Yust, Zvi Gutterman, and Benny Ben-Ami for many useful discussions and tips. We also thank Roni Rosenfeld, Ron Hecht and Ami Navon for their help with Cepstrum Analysis.

8. REFERENCES

[1] D. Asonov and R. Agrawal. Keyboard acoustic emanations. In *IEEE Symposium on Security and Privacy*, pages 3–11, Oakland, CA, 2004.

[2] K. Atkinson. Scowl - spell checker oriented word lists, 2004. `http://wordlist.sourceforge.net/`.

[3] R. Briol. Emanation: How to keep your data confidential. *Symposium on Electromagnetic Security For Information Protection*, 1991.

[4] CornCob. The corncob list. `http://www.mieliestronk.com/wordlist.html`.

[5] D. Klein. Foiling the cracker: A survey of, and improvements to, password security. In *Proc. UNIX Security Workshop II*, Aug. 1990.

[6] M. G. Kuhn. Compromising emanations: Eavesdropping risks of computer displays. Technical Report UCAM-CL-TR-577, University of Cambridge, Computer Laboratory, Dec. 2003.

[7] J. Loughry and D. A. Umphress. Information leakage from optical emanations. *ACM Trans. Info. Sys. Security*, 5(3):262–289, 2002.

[8] Y. Rubner, C. Tomasi, and L. Guibas. The earth mover's distance as a metric for image retrieval. *International Journal of Computer Vision*, 40(2):99–122, 2000.

[9] Time domain processing: Correlation. `http://www.bores.com/courses/intro/time/2_ave.htm`.

[10] M. Slaney. Auditory toolbox, 1998. `http://rvl4.ecn.purdue.edu/~malcolm/interval/1998-010/`.

[11] S. W. Smith. *The Scientist and Engineers Guide to Digital Sound Processing.* California Technical Publishing, 1997.

[12] D. Song, D. Wagner, and X. Tian. Timing analysis of keystrokes and timing attacks on SSH. In *10th USENIX Security Symposium*, 2001.

[13] Tempest 101. `http://www.tscm.com/TSCM101tempest.html`.

[14] L. Zhuang, F. 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 Press.

APPENDIX

A. DATA TABLES

Word	length
paediatrician	13
interceptions	13
abbreviations	13
impersonating	13
soulsearching	13
hydromagnetic	13
inquisition	11
pomegranate	11
feasibility	11
polytechnic	11
obfuscating	11
difference	10
wristwatch	10
processing	10
unphysical	10
institute	9
extremely	9
sacrament	9
dangerous	9
identity	8
emirates	8
platinum	8
homeland	8
security	8
between	7
spanish	7
nuclear	7

Table 3: List of words used to test the attack.

key	**ADJ**	*NEAR*	*DIST*
Q	Q,W,S,A	Q,W,A,E,S,Z,D,X	B,C,F,G,H,I,J,K,L,M,N,O,P,R,T,U,V,Y
A	A,Q,W,S,Z	A,Q,Z,W,S,X,E,D	B,C,F,G,H,I,J,K,L,M,N,O,P,R,T,U,V,Y
Z	Z,A,S,X	Z,Q,A,W,S,X,E,D,C	B,F,G,H,I,J,K,L,M,N,O,P,R,T,U,V,Y
W	W,Q,A,S,D,E	W,Q,A,Z,S,X,E,D,C,R,F	B,G,H,I,J,K,L,M,N,O,P,T,U,V,Y
S	S,Q,A,Z,X,D,E,W	S,Q,A,Z,W,X,E,D,C,R,F	B,G,H,I,J,K,L,M,N,O,P,T,U,V,Y
X	X,Z,A,S,D,C	X,Q,A,Z,W,S,E,D,C,F,V	B,G,H,I,J,K,L,M,N,O,P,R,T,U,Y
E	E,W,S,D,F,R	E,Q,A,Z,W,S,X,D,C,R,F,V,T,G	B,H,I,J,K,L,M,N,O,P,U,Y
D	D,E,W,S,X,C,F,R	D,Q,A,Z,W,S,X,E,C,R,F,V,T,G	B,H,I,J,K,L,M,N,O,P,U,Y
C	C,X,D,F,V	C,W,S,Z,E,S,X,R,D,F,V,T,G,B	A,H,I,J,K,L,M,N,O,P,Q,U,Y
R	R,E,D,F,G,T	R,W,S,X,E,D,C,F,V,T,G,B,Y,H	A,I,J,K,L,M,N,O,P,Q,U,Z
F	F,R,E,D,C,V,G,T	F,W,S,X,E,D,C,R,V,T,G,Y,H,B	A,I,J,K,L,M,N,O,P,Q,U,Z
V	V,C,D,F,G,B	V,E,D,X,R,F,C,T,G,B,Y,H,N	A,I,J,K,L,M,O,P,Q,S,U,W,Z
T	T,R,F,G,H,Y	T,E,D,C,R,F,V,G,Y,H,B,U,J,N	A,I,K,L,M,O,P,Q,S,W,X,Z
G	G,T,R,F,V,B,H,Y	G,E,D,C,R,F,V,T,B,Y,H,N,U,J	A,I,K,L,M,O,P,Q,S,W,X,Z
B	B,V,G,H,N	R,D,C,T,F,V,G,Y,H,N,U,J,M	A,B,E,I,K,L,O,P,Q,S,W,X,Z
Y	Y,T,G,H,J,U	Y,R,F,V,T,G,B,U,H,I,J,N	A,C,D,E,K,L,M,O,P,Q,S,W,X,Z
H	H,Y,T,G,B,N,J,U	H,R,F,V,T,G,B,Y,U,J,N,I,K,M	A,C,D,E,L,O,P,Q,S,W,X,Z
N	N,B,H,J,M	N,T,F,V,Y,G,B,H,U,J,M,I,K	A,C,D,E,L,O,P,Q,R,S,W,X,Z
U	U,Y,H,J,K,I	U,T,G,B,Y,H,N,I,J,O,K,M,L	A,C,D,E,F,P,Q,R,S,V,W,X,Z
J	J,U,Y,H,N,M,K,I	J,T,G,B,Y,H,U,N,I,K,M,O,L	A,C,D,E,F,P,Q,R,S,V,W,X,Z
M	M,N,J,K	M,Y,H,B,U,J,N,I,K,O,L	A,C,D,E,F,G,P,Q,R,S,T,V,W,X,Z
I	I,U,J,K,L,O	I,Y,H,N,U,J,M,O,K,P,L	A,B,C,D,E,F,G,Q,R,S,T,V,W,X,Z
K	K,I,U,J,M,L,O	K,Y,H,N,U,J,M,I,O,L,P	A,B,C,D,E,F,G,Q,R,S,T,V,W,X,Z
O	O,I,K,L,P	O,U,J,M,I,K,P,L	A,B,C,D,E,F,G,H,N,Q,R,S,T,V,W,X,Y,Z
L	L,O,I,K,P	L,U,J,N,I,K,M,O,P	A,B,C,D,E,F,G,H,Q,R,S,T,V,W,X,Y,Z
P	P,O,L	P,I,J,M,I,K,O,L	A,B,C,D,E,F,G,H,N,Q,R,S,T,U,V,W,X,Y,Z

Table 4: *ADJ*, *NEAR*, *DIST* tables for each of the keys, as used in our algorithms.