Bypassing Key-loggers

Secure password entry when input systems are compromised

Mathew Kuthur James
David Cheriton School of
Computer Science
University of Waterloo
Waterloo, Ontario, Canada
m2kuthur@uwaterloo.ca

ABSTRACT

This project explores three different schemes to enter passwords via a keyboard compromised by a key-logger. It also explores methods to use information gathered from a key-logger to attack the schemes presented, along with methods to optimize usability.

2 Introduction

The project aims to develop a set of tools which enable someone to safely enter a password in a situation where every input they give to the system is visible to an attacker. For instance, when entering a password on a system which could be infected by a key-logger or when the movements of the hand on the keyboard are visible to the public. The threat model assumes that communication made from the system directed to the user is done via a safe channel.

The alphabet from which the password is constructed is Σ, with | Σ| referring to the number of characters in the alphabet. For a password string P, |P| refers to the length of the password.

2 Threat Model

The attacker is assumed to have access to a key-logger installed in the input device where the password is entered. The key-logger records the keystrokes along with the timestamp for each keystroke. The attacker has access to multiple successful login attempts using the same password.

We assume that the output screen is not seen by the attacker and that the software running on the system is not compromised (except for the key-logger itself). Since capturing the screen produces a large amount of data, a potential screen-capture device would have to either keep streaming the data over a high-bandwidth channel or have enough local storage space to store the data. Furthermore, any attack that relies on reading information from the screen can be frustrated by rendering data in a manner that makes optical character recognition prohibitively computationally expensive.

We assume that the user does not have access to a separate trusted device (since if they did, they could simply use this device to log in instead of the compromised system). We also assume that the user does not have access to computational abilities beyond that of a typical human (for instance, the user cannot compute the SHA-256 hash of a string).

3 Shuffled String

3.1 Description

Given a password P, a string S is constructed, where the characters of P are woven into S alongside randomly chosen characters from the alphabet. The alphabet is the set of valid hexadecimal digits. A successful login requires the entry of S by the user.

The user is shown S with blank spaces where the characters of P are supposed to be inserted. The threat model assumes that the screen is not captured, so the

attacker does not know which characters in S belong to P.

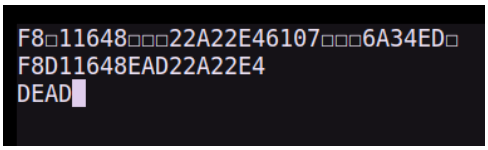

Figure1:**P='DEADFEED', S='F8D11648EAD22A22E46107FEE6A34EDD'**

An attacker with access to the system can determine the $|P|$. A brute force attack requires the trial of $|\Sigma|^{|P|}$ possible passwords.

An attacker with a single sample of a successful input and no other information will know that P is a non-contiguous substring of S. Each character in S may or may not be part of P. The number of combinations to try thus becomes $2^{|S|}$. The entropy of the system is the negative binary logarithm of the probability of a successful password entry, which is:

$$-lb(\frac{1}{2^{|S|}}) = |S|$$

The entropy can, independently of P, be made arbitrarily high by increasing the amount of chaff in S.

The scheme requires the user to do the following in addition to what the user would do when entering a password as is:

1. The user must keep track of whether the current character being entered is part of the password or the chaff.

2. The user must identify which character of the chaff is to be entered (note that allowing the user to generate the chaff on the fly is less secure than automatic generation, since humans tend to perform poorly at simulating true randomness.

3. The user must identify which character of the password is to be entered, by keeping track of the part of the password already entered.

Good UI design minimizes the mental effort required to perform the password entry. The number of times the user switches from "chaff-entry mode" (where the user copies characters from the screen) to "password-entry-mode" (where the user copies characters from their own memory) is determined by the distribution of the password within the chaff.

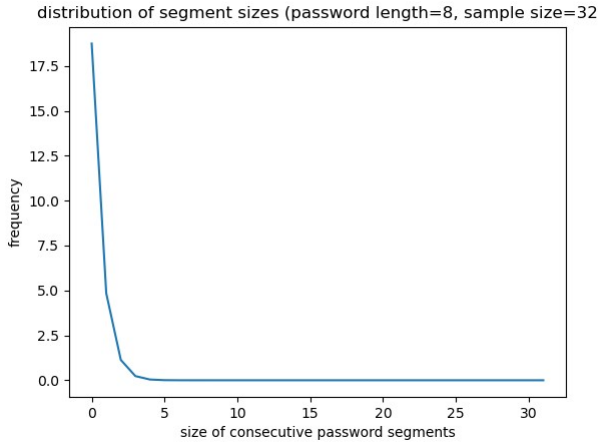

Figure 2: **distribution of segment sizes**

When S is generated via the insertion of chaff characters in random positions of the password, we find that for small passwords, most occurrences of password characters are in segments of size 1. Larger segments are very rare, with points of transition between the chaff and the password tracking closely with the size of the password itself.

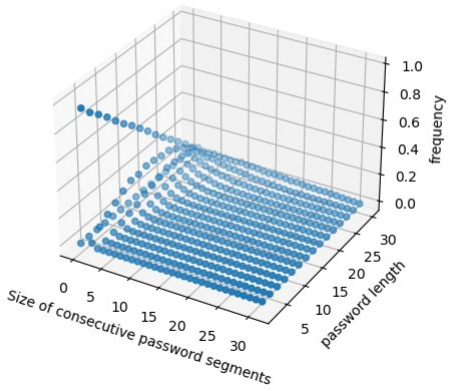

Figure 3: **distribution of password segments**

3.2 Frequency Attack

Assuming that the attacker has access multiple samples with random chaff and no change in the password, frequency analysis of the samples reveals information about the password.

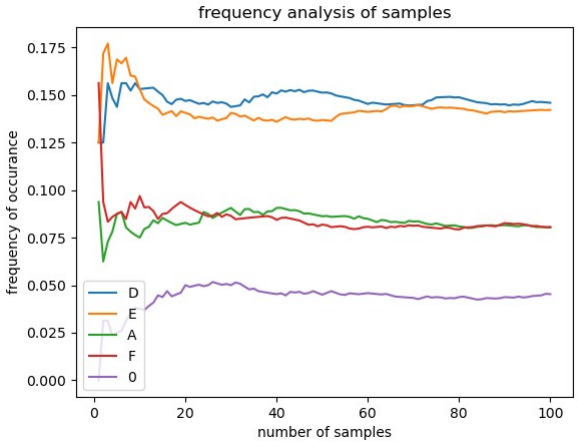

Figure 4: **frequency analysis for P="DEADFEED"**

In the figure above, the character `0` is plotted as a stand-in for a character not found in the password but found in the alphabet. After around 10 samples, it becomes clear which characters are present in the password. At 20 samples, the relative frequency of the characters allows educated guesses about the frequency of occurrence of each character in the password. In the best case (where every character in the password is unique), the number of possible permutations is !|P|, which lowers the entropy of the password.

Once the constituent characters are identified as `D`,`E`, `A` and `F`, further information about the structure of the password can be revealed by looking at frequencies of character pairs like `DE`, `AD` and `FE`. This technique only works for cases where the average size of a continuous password segment crosses 2, which occurs when the password forms a large percentage of the sample.

A possible defense against this technique is to pick the chaff in such a way that the final distribution of characters in S is such that no character is privileged. However, doing so would require storing information about the characters in P (presumably on a server, which computes the chaff and sends it to the client during every login attempt). However, this makes the scheme vulnerable to data leaks from the entity which stores information about P.

3.3 String Analysis Attack

Let S be two input samples captured by the key-logger. Assume that the password is P. P must be a non-contiguous substring of S (for instance, "AX" and "AY a non-contiguous substring of "ABXY"). Given a single sample S, the number of possible passwords is 2^|S|. If |S| is sufficiently large, the brute-force attack becomes infeasible.

Imagine that the attacker has access to two samples S1 and S2. Let n be the number of non-contiguous substrings between S1 and S2. Though n can, in theory, be equal to 2^|S1| (in the case where S1 and S2 are the same), n in practice is often much smaller (since the characters of S1 and S2 are decided by a random process). Given two samples, the effective entropy of the system is reduced to lb(n).

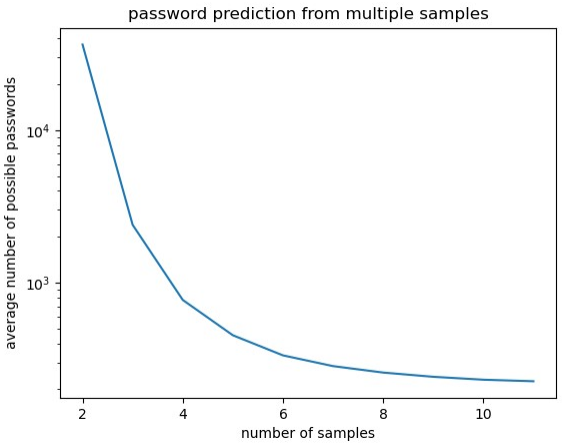

Figure 5: **password prediction from multiple samples**

Let f(S1,S2) = {X: X is a common non-contiguous substring of S1 and S2}. Let H be a set of strings. Let g(S,H) = union over S_ in H: f(S,S_). Via repeated application of g on the samples gathered, the set of possible passwords decreases and converges to a minimum value. The figure shows the results when this process is applied to a large set of possible passwords. One can, in figure 5, observe a sharp drop in the number of possible passwords (and thus effective entropy) with just a small number of samples.

A possible improvement on this attack is to focus solely on the largest common non-contiguous substring between the samples. Though the password is not guaranteed to be present as a non-contiguous substring in the longest common non-contiguous substrings (a trivial counterexample is P="AB", S1="WXYZAB", S2="XYZAWB", longest common non-contiguous substring="XYZ"), the password is likely to be captured by such a process for small sets of samples.

In the special case of exactly two samples S1 and S2, if m is the length of the longest common non-contiguous substring between S1 and S2, then |P| must be less than or equal to m (since P is a common non-contiguous substring between S1 and S2). Thus, finding m for a pair of samples S1 and S2 results in information about the password length. In practice, the sample space for the password is cut down significantly from just two samples. With m slots, each slot taking one character in the alphabet Σ, producing a total number of permutations = $|\Sigma|^{|P|}$, the entropy is reduced to:

$$lb(|\Sigma|^m) = m * lb(|\Sigma|)$$

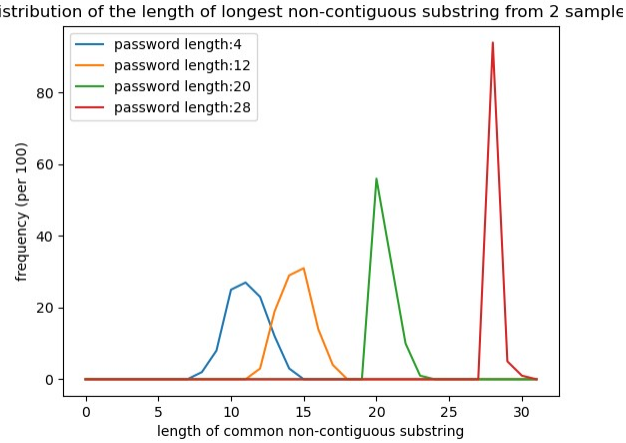

Figure 6

The figure above shows how m often results in a value very close to |P|, with the effect being more pronounced for large passwords.

3.4 Algorithms for the longest non-continuous substring

The use of dynamic programming with memoization enables one to find m in O(|S1|*|S2|) time. The brute force approach (of checking each possible substring) takes O(2^|S1| * |S2|) time - O(|S2|) time for checking each of the 2^|S1| nc subs of S1. Let f(S1,S2) compute the length of the largest non-contiguous substring common to S1 and S2.

```
f(S1, S2):
    if |S1| or |S2| is 0:
        return 0
    assume S1 = S.x
    if x is not in S2:
        return find_m(S,S2)
    assume S2 = PxQ, where x is not present in Q
    return max(f(S,S2),f(S,P)+1)
```

The simplest version of the algorithm consists only of recursion. The base case is when either S1 or S2 is a zero-length string. In each recursive call, the length of at least one of the arguments is reduced by at least 1, which guarantees termination.

In the base case, the largest common non-contiguous substring is the empty string, whose length is zero.

If the base case doesn't hold, S1 and S2 must have non-zero length. Assume that S1 = Sx, where x is a single character appended to the end of the string S. Assume that the largest common non-contiguous substring is T. The character x is either present or not present in S2.

If x is not present in S2, x cannot be present in T either. Thus the function can be evaluated by dropping all instances of x in S1 and S2.

$$f(S1, S2) = f(Sx, S2) = f(S, S2)$$

If x is present in S2, assume that S2 = PxQ, where P and Q are strings such that x is not present in Q. In other words, P contains every instance of x in S2 except the last. T may or may not end with x. If T does not end with x and T is contained in Sx, then T must be contained in S. So for this case,

$$f(S1, S2) = f(Sx, S2) = f(S, S2)$$

If T does end with x, then those parts of S1 and S2 past the last occurrence of x can be eliminated.

To combine the two cases of the ending of T,

$$f(S1, S2) = max(f(S, S2), f(S, P) + 1)$$

To extend the algorithm to include dynamic programming and memoization, consider that each recursive call to f is evaluated on prefixes of the original arguments given to f. Constructing a table and computing f for each substring of S1 and S2 step by step allows the re-use of solutions from memory. For instance, consider f("DEAD","XEYA"):

	D	DE	DEA	DEAD
X	0	0	0	0
XE	0	1	1	1
XEY	0	1	1	1
XEYA	0	1	2	2

The algorithm is extended to computing common non-contiguous substrings by storing the set of strings in place of their length. The time complexity of this implementation depends on the specific complexities of the implementations of the set data structure.

```
F(S1,S2):
    if |S1| or |S2| is 0:
        return {""}
    assume S1 = S.x
    if x is not in S2:
        return f(S,S2)
    assume S2 = PxQ, where x is not present in Q
    let W = f(S,S2)
    let V = {Tx : T ∈ f(S,P)}
    return W∪V
```

	D	DE	DEA	DEAD
X	{""}	{""}	{""}	{""}
XE	{""}	{"E",""}	{"E",""}	{"E",""}
XEY	{""}	{"E",""}	{"E",""}	{"E",""}
XEYA	{""}	{"E",""}	{"EA","E",""}	{"EA","E",""}

4 Iterated choice

Since the attacks on the previous scheme rely on the user typing the characters present in the password, these can be defended against by introducing an alternate scheme which never requires the typing of the characters in the password at any time.

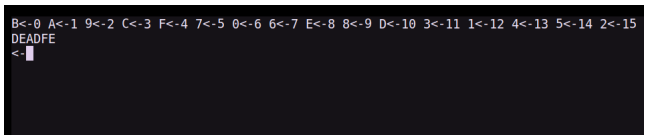

Figure 7: **Iterated choice scheme for P="DEADFEED"**

This scheme prevents any character in the password from being input, by creating a random bijective function from the set of whole numbers to the characters in the alphabet. The user enters a whole number, and the function is applied to it to get the character. To avoid revealing information about the positions of repeated characters, different random functions are chosen after each character is input.

This scheme suffers poor usability, since the user needs to compute the inverse of the function at each step, by searching for the character to input. Improving the usability is the primary motivation for the introduction of the next scheme.

5 State Machine

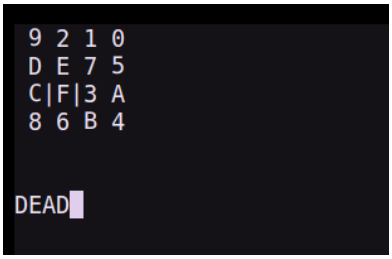

Figure 8: **State machine scheme for P="DEADFEED"**

5.1 Description

In this scheme of password entry, the user never directly types any of the password characters on the keyboard. Instead, the password entry is done by a state machine with one state for each alphabet in the language. The user controls the state machine by giving the inputs for its transitions. The state machine has the following characteristics:

1) The initial state is chosen at random.

2) Every state is reachable.

3) Transitions between different states write no output.

4) The state associated with the character x has a transition to itself, with x as the output.

A simple implementation of such a state machine is a grid which allows transitions along four directions (triggered by `wsad`, with `enter` for the self-transition). The figure shows a state machine with 16 states, with its current state at `F`. Triggering the self-transition writes `F` to the output.

Consider a password where a character appear multiple times, such as `E` in "DEADFEED". When the user is at the second-last `E`, the user triggers the self-transition twice by hitting `enter` twice to generate the two adjacent `E`s. This reveals to the attacker that the password has the same character in the second and third last positions.

Furthermore, the sequence of transitions taken from the first `E` to the second `E`, when simulated on a state

machine with the same structure, reveals that the final and initial states are the same. This reveals to the attacker that the second character is the same as the third last character.

To prevent revealing details about the structure of the password, the state machine needs to be randomized after every output is made. The characters associated with each state is randomly reassigned.

5.2 **Usability**

A simple measure of usability is the number of transitions that need to be taken per character of the password.

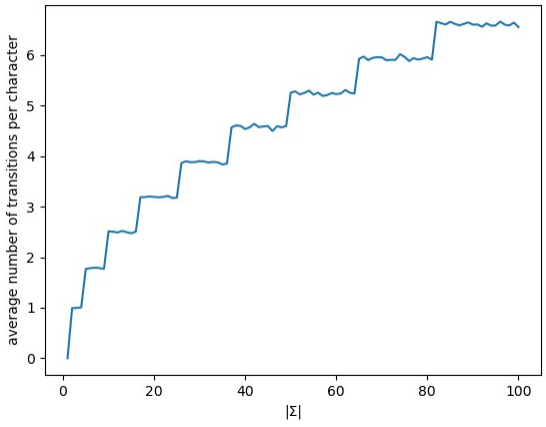

Figure 9: **Variation of the average number of transitions per character with the size of the alphabet**

When varying the size of the language, the size of the square grid (N) varies as:

$$N = \left\lceil \sqrt{|\Sigma|} \right\rceil$$

The dependence of the average number of transitions per character on N is linear.

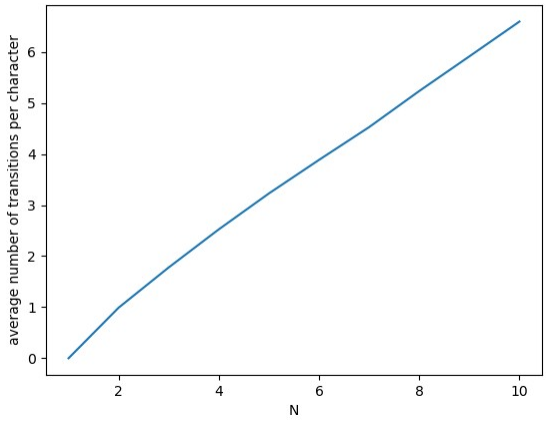

Figure 10: **Variation of the average number of transitions per character with N**

Let the entropy of the system be E, and the length of the password be |P|. Then:

$$E = lb(|\Sigma|^{|P|}) = |P| * lb(|\Sigma|)$$

Since the entropy varies with both |Σ| and the password length, minimizing the average number of transitions requires for a given entropy requires the minimization of N*|P| under the constraint that |P|*lb(|Σ|) = E, where N depends on |Σ|.

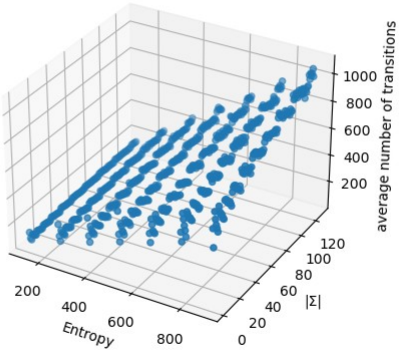

Figure 11: **Variation of the average number of state transitions**

This minimum point occurs at small sizes of $|\Sigma|$, which suggests that smaller alphabets are optimal for UX. However, this result assumes that the time taken to enter a password is proportional to the number of state transitions, which may not hold in practice.

6 Contributions

This project makes the following contributions:

1) Implementation of prototypes for alternate password entry schemes.

2) Description of a dynamic programming algorithm to compute the length of the longest non-contiguous substring of two strings, running in polynomial time.

3) Description of a dynamic programming algorithm to compute all non-contiguous substring of two strings.

4) A description of an attack that uses the aforementioned algorithms to gain information about passwords.

5) An analysis of the effectiveness of the aforementioned attack.

6) An analysis of the effectiveness of frequency analysis as a possible attack.

7) Analysis of the usability of the alternate password entry schemes.

The prototype is hosted at the following repository: https://github.com/MathewKJ2048/CS858-fall-project

The data generated is done via scripts hosted in the same location.