Fast Dictionary Attacks on Passwords Using Time-Space Tradeoff

Arvind Narayanan and Vitaly Shmatikov
The University of Texas at Austin
{arvindn,shmat}@cs.utexas.edu

ABSTRACT

Human-memorable passwords are a mainstay of computer security. To decrease vulnerability of passwords to brute-force dictionary attacks, many organizations enforce complicated password-creation rules and require that passwords include numerals and special characters. We demonstrate that as long as passwords remain human-memorable, they are vulnerable to "smart-dictionary" attacks even when the space of potential passwords is large.

Our first insight is that the distribution of letters in easy-to-remember passwords is likely to be similar to the distribution of letters in the users' native language. Using standard Markov modeling techniques from natural language processing, this can be used to dramatically reduce the size of the password space to be searched. Our second contribution is an algorithm for efficient enumeration of the remaining password space. This allows application of time-space tradeoff techniques, limiting memory accesses to a relatively small table of "partial dictionary" sizes and enabling a very fast dictionary attack.

We evaluated our method on a database of real-world user password hashes. Our algorithm successfully recovered 67.6% of the passwords using a 2×10^9 search space. This is a much higher percentage than Oechslin's "rainbow" attack, which is the fastest currently known technique for searching large keyspaces. These results call into question viability of human-memorable character-sequence passwords as an authentication mechanism.

Categories and Subject Descriptors:
K.6.5[**Security and Protection**]: Authentication;
D.4.6[**Security and Protection**]: Authentication;
E.3[**Data Encryption**]: Code breaking

General Terms: Security

Keywords: Passwords, Dictionary Attack, Time-Space Tradeoff, Cryptanalysis, Markov Models

1. INTRODUCTION

It is often said that humans are always the weakest link in the security chain. Social engineering is more frequently successful in penetrating a system than buffer overflow attacks. Similarly, cracking a password is rarely accomplished by breaking the cipher used and more often by exploiting the circumstance that it was generated by a human.

The problem of guessing human-generated passwords has been studied for a long time. Morris and Thompson [27] in their 1979 paper on UNIX password security describe brute-force and dictionary attacks that are all too familiar to the modern reader. The former is based on the observation that short strings are easier for humans to remember, and the latter on the fact that meaningful words are far more memorable than random character sequences.

The state of the art in dictionary attacks has not advanced much since then. Current password crackers such as John the Ripper [29] use essentially the same two techniques, plus a few rules to turn each dictionary word into a set of related words (such as suffixing a digit). The brute-force attack can be made somewhat more efficient by using a time-space tradeoff such as Oechslin's "rainbow" technique [28, 1], which employs precomputation to speed up the process of cracking individual passwords. A common defense is the use of password-creation rules, called *composition rules*, that require passwords to be drawn from certain regular languages, and require passwords to include digits and non-alphanumeric characters. The goal is to increase the size of the search space, making naive dictionary attacks infeasible.

We demonstrate that fast, devastating "smart dictionary attacks" can be staged even on large password spaces that are not vulnerable to brute-force or straightforward dictionary attacks. Our insight is that passwords, even if long and sprinkled with extra characters, must still be human-memorable. Human limitations imply that the entropy of passwords is rather small: an NIST document [7] estimates that user-generated 8-character passwords have between 18 and 30 bits of randomness, even when English dictionary words are filtered out and composition rules are enforced. In a very general sense, our attacks work by rapidly enumerating candidate passwords that can be produced with a small amount of randomness. For instance, the string `asasasasasasasasasasasasasasasas`, at 32 characters, is far beyond the reach of brute-force attacks using current hardware. Intuitively, however, this string is not very random, and should be easily guessable.

Formally, this can be modeled by saying that the *Kol-*

mogorov complexity [23] of this string is low. The Kolmogorov complexity (a.k.a. K-complexity) of a string is defined as the length of the shortest Turing Machine that outputs it and then halts. The sequence of instructions `repeat print "as" 16 times` in any reasonable encoding scheme should have a complexity of around 30 bits, which is in keeping with our intuition that the resulting string is easily guessable.

An obvious approach would be to try to enumerate all strings which have a K-complexity smaller than some threshold, and use the result as the dictionary. There are two problems with this. First, K-complexity is uncomputable. Second, human perception of randomness in different from computational randomness. Subjective randomness has been studied by cognitive scientists Griffiths and Tenenbaum [15, 16], who argue that we need a different way to measure the perceived information content of a string. What is needed is a rigorous multidisciplinary study of human password generation processes, as well as the algorithms for very efficient enumeration of the resulting password spaces. The existence of such algorithms would be a strong indicator of vulnerability to fast dictionary attacks.

We take the first steps in both directions. We posit that phonetic similarity with words in the user's native language is a major contributor to memorability. We capture this property using "Markovian filters" based on Markov models, which is a standard technique in natural language processing [32]. Our other observation is that the way in which humans use non-alphabetic characters in passwords can be modeled by finite automata, which can be considered a generalization of the "rules" used by John the Ripper and other password cracking tools.

Secondly, we present an algorithm to *efficiently* enumerate all strings of a given length that satisfy a Markovian filter, and an algorithm to efficiently enumerate all strings of a given length accepted by a deterministic finite automaton. More precisely, we give algorithms that, given an index i, generate the i^{th} string satisfying each of these properties (without searching through the entire space!). Next, we combine these two algorithms into a hybrid algorithm that produces the i^{th} string satisfying *both* the Markovian filter and the regular language. Finally, we show how to use this algorithm to implement a time-space tradeoff over this search space.

The best-known brute-force attack technique using the time-space tradeoff was proposed by Oechslin [28]. It uses a special data structure called the "rainbow table." Our hybrid attack has the same precomputation time, storage requirement and mean cryptanalysis time as the rainbow attack over a search space of the same size. Yet, compared to the vanilla rainbow attack over the entire fixed-length keyspace (*e.g.*, "all alphanumeric strings of length 8"), our attack has a significantly higher *coverage* since our search space is chosen wisely and consists only of "memorable" strings that have low subjective randomness.

Of course, coverage of a dictionary attack can only be measured by applying it to "real-world" user passwords. Results obtained on artifically generated passwords would not be very convincing, since there is no guarantee that the generation algorithm produces passwords that are similar to those users choose for themselves. We evaluated our techniques on a database of 150 real user passwords provided to us by Passware. Our hybrid attack achieved 67.6% coverage (*i.e.*, more than two thirds of the passwords were successfully recovered), using a search space of size 2×10^9. Detailed results are presented in Section 5. The filters we used can be found in Appendix A. By contrast, Oechslin's attack achieves coverage of only 27.5%[1].

We believe that with a larger search space (around 3×10^{12}, which is what the `passcracking.com` implementation uses), and more comprehensive regular expressions, coverage of up to 90% can be achieved. A more rigorous and larger experiment with the `rainbowcrack` [33] implementation is in progress.

Defenses against dictionary attacks. Salting of password hashes defeats offline dictionary attacks based on precomputation, and thus foils our hybrid attack.

Using an inefficient cipher slows the attacker down by a constant factor, and this is in fact done in the UNIX `crypt()` implementation. This technique, however, can only yield a limited benefit because of the range of platforms that the client may be running. Javascript implementations in some browsers, for example, are extremely slow.

Feldmeier and Karn [10] surveyed methods to improve password security and concluded that the only technique offering a substantial long term improvement is for users to increase the entropy of the passwords they generate. As we show, achieving this is far harder than previously supposed.

There is also a large body of work, subsequent to the above survey, on password-authenticated cryptographic protocols and session key generation from human-memorable passwords [3, 24, 2, 6, 19, 14, 13]. The objective of these protocols is to defeat *offline* dictionary attacks on protocols where participants share a low-entropy secret.

One drawback of password-authenticated key exchange (PAKE) protocols is that they typically rely on unrealistic assumptions such as multiple noncooperating servers or both parties storing the password in plaintext (one exception is the PAK-X protocol of [6]). Storing client passwords on the server is very dangerous in practice, yet even for "provably secure" PAKE protocols, security proofs implicitly assume that the server cannot be compromised.

Furthermore, our attacks apply in a limited sense even to PAKE protocols protocols because our Markovian filters also make *online* dictionary attacks much faster. Thus, our attacks call into our question whether it is ever meaningful for humans to generate their own character-sequence passwords. The situation can only become worse with time because hardware power grows exponentially while human information processing capacity stays constant [25]. Considering that there is a fundamental conflict between memorability and high subjective randomness, our work could have implications for the viability of passwords as an authentication mechanism in the long run.

Organization of the paper. In Section 2, we explain filtering based on Markovian models and deterministic finite automata. In Section 3, we discuss the time-space tradeoff in general and Oechslin's rainbow attack in particular, and show that it works over *any* search space as long as there is an efficient algorithm to compute its i^{th} element. In Sec-

[1]We used the character set consisting of lowercase alphabets and numerals for Oechslin's attack. Simple tricks can improve the coverage, such as running more than one timespace tradeoff with different character sets, and using brute force up to some small length.

tion 4, we present our search space enumeration algorithms for strings satisfying a Markovian filter, strings accepted by a DFA and strings that satisfy *both* conditions. Section 5 consists of our experimental results. Conclusions are in Section 6.

2. FILTERING

In the rest of this paper, we will use words *key* and *password*, *ciphertext* and *password hash*, and *keyspace* and *dictionary* interchangeably. The reason for this is that standard techniques for brute-force cryptanalysis, such as those described in Section 3, typically refer to keys and ciphertexts even when applied to passwords and their hash values.

2.1 Markovian filtering

An alphabetical password generated by a human, even if it is not a dictionary word, is unlikely to be uniformly distributed in the space of alphabet sequences. In fact, if asked to pick a sequence of characters at random, it is likely that an English-speaking user will generate a sequence in which each character is roughly equidistributed with the frequency of its occurrence in English text. Analysis of our password database reveals a significant number of alphabetical passwords which are neither dictionary words, nor random sequences (we used the `openwall.com` dictionary which contains about 4 million words [30]).

Markov models are commonly used in natural language processing, and are at the heart of speech recognition systems [32]. A Markov model defines a probability distribution over sequences of symbols. In other words, it allows sampling character sequences that have certain properties. In fact, Markov models have been used before in the context of passwords. Subsequent to this work, we learned of the "Extensible Multilingual Password Generator" software, which was written by Jon Callas in 1991 and used precisely this technique to *generate* passwords for users. (We are also told that a similar method for generating "random, yet pronounceable" passwords was used at the Los Alamos National Laboratory in the late 1980s.) It should come as no surprise, then, that Markov modeling is very effective at guessing passwords generated by users.

In a *zero-order* Markov model, each character is generated according to the underlying probability distribution and independently of the previously generated characters. In a *first-order* Markov model, each digram (ordered pair) of characters is assigned a probability and each character is generated by looking at the previous character. Mathematically, in the zero-order model,

$$P(\alpha) = \Pi_{x \in \alpha} \nu(x)$$

while in the first-order model,

$$P(x_1 x_2 \ldots x_n) = \nu(x_1) \Pi_{i=1}^{n-1} \nu(x_{i+1}|x_i)$$

where $P(.)$ is the Markovian probability distribution on character sequences, x_i are individual characters, and the ν function is the frequency of individual letters and digrams in English text.

Of course, a dictionary is not a probability distribution, but a set. Therefore, to create a Markovian dictionary, we discretize the probabilities into two levels by applying a threshold θ. This defines the zero-order dictionary

$$\mathcal{D}_{\nu,\theta} = \{\alpha : \Pi_{x \in \alpha} \nu(x) \geq \theta\}$$

and the first-order dictionary

$$\mathcal{D}_{\nu,\theta} = \{x_1 x_2 \ldots x_n : \nu(x_1) \Pi_{i=1}^{n-1} \nu(x_{i+1}|x_i) \geq \theta\}$$

The zero-order model produces words that do not look very natural, but it can already drastically reduce the size of the plausible password space by eliminating the vast majority of character sequences from consideration. Consider 8-character sequences. If θ is chosen so that the dictionary size is $\frac{1}{7}$ of the keyspace (*i.e.*, 86% of sequences are ignored), then a sequence generated according to the model has the probability of 90% of belonging to the dictionary. In other words, 15% of the password space contains 90% of all *plausible* passwords. Other interesting points on the curve are: a dictionary containing $\frac{1}{11}$ of the keyspace has 80% coverage, and a dictionary with $\frac{1}{40}$ of the keyspace has 50% coverage, *i.e.*, only 2.5% of the keyspace needs to be considered to cover half of all possible passwords! The first-order model can do even better. The results are shown in Figure 1.

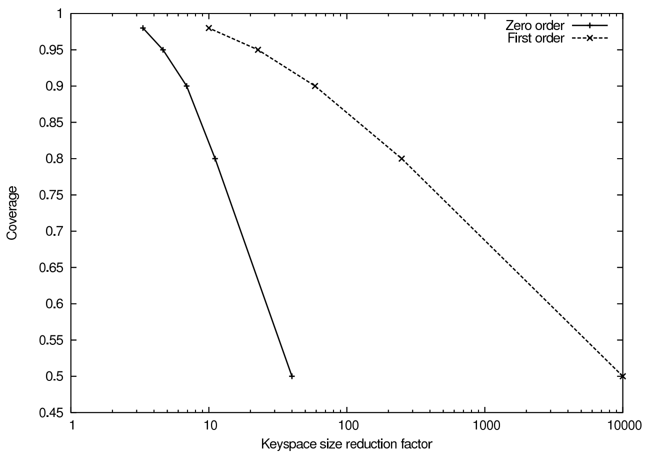

Figure 1: Coverage compression graph

The zero-order model is not obviated by the first-order model. The zero-order model is better for certain commonly used password-generation strategies, such as acronyms consisting of first letters of each word in a sentence.

Needless to say, the distribution of letter frequencies used in keyspace compression via Markovian filtering is language-specific, and the distribution used in this paper applies only to passwords chosen by English-speaking users (presumably, similar distributions can be found for other alphabet-based languages). There are two ways in which the technique can be generalized if the user's native language is not known. First, it is possible to combine the keyspaces for two or more languages (Section 4.6). Second, it may be possible to come up with a distribution that works reasonably well for multiple languages (*e.g.*, all Germanic or all Romance languages). We have not tried doing this and don't know how well it might work.

2.2 Filtering using a finite automaton

A search space consisting of only alphabetic sequences is unlikely to have a good coverage of the plausible password space. Humans often mix upper- and lowercase characters in their passwords, and system-enforced password-creation rules often require them to throw in some numerals and sometimes special characters. Yet, even with these additions, the distribution of resulting passwords is far from random. Below are examples of a few common patterns (this list is by no means definitive):

- In an alphanumeric password, all numerals are likely to be at the end.
- The first character of an alphabetic sequence is far more likely to be capitalized than the others.
- While alphabetic sequences consisting of mostly lowercase characters and a few uppercase characters are common, the converse is not true.

Deterministic finite automata are ideal for expressing such properties. First, we specify a set of common regular expressions ("all lowercase," "one uppercase followed by all lowercase," "uppercase characters followed by numerals," and so on). We define our dictionary to be the set of sequences matching the Markovian filter *and* also accepted by at least one of the finite automata corresponding to the regular expressions. Thus $\mathcal{D}_{\nu,\theta,\langle M_i\rangle} = \{\alpha : \Pi_{x\in\alpha}\nu(x) \geq \theta$, and $\exists i : M_i$ accepts $\alpha\}$. Our complete alphabet consists of 26 lowercase and 26 uppercase characters, 10 numerals and 5 special characters (space, hyphen, underscore, period and comma). We have chosen these five somewhat arbitrarily because we felt they are the ones that occur most commonly in human-generated passwords. It is not possible to consider *all* special characters without knowing what the actual character set is for the application.

This gives us a 67-character alphabet. The associated keyspace of 8-letter sequences is 10^{15}, making brute-force search infeasible. While it is possible to write regular expressions for this 67-character symbol set, the resulting algorithms are not very efficient. Therefore, we give up a little bit of expressivity and group the character set into four categories (lowercase, uppercase, numerals, and special characters, which we will denote, respectively, as a, A, n and s), and consider the input alphabet of our automata to consist of just these four symbols. The regular expressions we have used are listed in Appendix A.

3. TIME-SPACE TRADEOFF

The most basic precomputation technique is to compute and store the hashes of all the passwords in the keyspace ordered by hash, so that cryptanalysis is almost instantaneous. However, this requires storage equal to the keyspace size. Hellman [17] showed how to decrease storage requirements at the expense of attack time in the general context of cryptanalyzing a cipher. This tradeoff works as follows.

Given a fixed plaintext P, define a mapping f on the keyspace $\mathcal{K}$ as $f(k) = R(E_k(P))$ where E is the encryption function and R is a *reduction function* which maps ciphertexts to keys. By iterated applications of f, we create a "chain" of keys. The crucial observation is that by storing *only the first and last elements of a chain*, we can determine if the key corresponding to a given ciphertext belongs to that chain (and also find the key) in time $O(t)$ where t is the length of the chain.

Creating a chain works as follows. Given a starting point k_0, we compute $k_1 = f(k_0), k_2 = f(k_1), \ldots, k_t = f(k_{t-1})$. The keys k_0 and k_t are stored, while the rest are thrown away. When given a ciphertext C to cryptanalyze, we recover the key by computing $k = R(C)$ and $k_i = f^{i-1}(k)$ for $i = 1, 2, \ldots$. Observe that if the key k belongs to the chain, *i.e.*, $k = k_i$ for some i, then $f^{t-i}(k) = k_t$. Thus, after at most t applications of f, we can determine whether or not the chain contains k. Further $i-1$ applications of f suffice to compute k_{i-1} from k_0. Since k_{i-1} satisfies the property that $C = E_{k_{i-1}}(P)$, it is the key we are looking for.

This does not work with certainty because different chains may merge. Therefore, we need to use multiple tables with a different reduction function for each table, with many chains in each table. The storage requirement as well as the cryptanalysis time for this algorithm are $O(|\mathcal{K}|^{2/3})$, where $\mathcal{K}$ is the keyspace.

Rivest [9, p.100] suggested an improvement which greatly speeds up the practical performance of the algorithm by decreasing the number of memory accesses during cryptanalysis. This is done using "distinguished points," which are keys with some special property (such as the first 10 bits being zero). The advantage of requiring that the endpoints of a chain be distinguished points is that during cryptanalysis, a key must be looked up in memory only if it has the distinguished property. Further papers [11, 21, 5, 22, 34] presented optimizations and/or better analyses of the distinguished points algorithm.

A major improvement was made by Oechslin [28] by using "rainbow chains" instead of distinguished points. Rainbow chains use a different reduction function for each point in the chain. It was shown in [28] that rainbow chains achieve the same coverage as distinguished points with the same storage requirement but a significantly faster cryptanalysis time.

Experimental results are impressive. The online implementation of the rainbow attack [1] inverts MD5 hashes of passwords of length up to 8 over the character set `[a-z0-9]` (a keyspace size of 2.8×10^{12}). Its success probability is 0.996 with an amortized cryptanalysis time of under 10 minutes using precomputed tables of size about 48 GB.

3.1 Generic time-space tradeoff using index lookup property

We observe that in both Rivest's and Oechslin's algorithms, the reduction function can be expressed as a mapping from the ciphertext space to $\{0, 1, \ldots |\mathcal{K}|-1\}$, composed with a mapping from $\{0, 1, \ldots |\mathcal{K}|-1\}$ to $\mathcal{K}$. Neither the reduction function, nor any other part of the algorithms makes any assumptions about the keyspace other than its size. In Rivest's attack, the property of being a distinguished point can be computed from the keyspace index rather than the key itself. In the case of the rainbow attack, the mapping from the ciphertext space to keyspace indices is parameterized by the choice of the rainbow table, but the mapping from the keyspace index to the key is the same as in Rivest's attack. Therefore, we can implement either attack over the compressed dictionary of plausible passwords described in Section 2.

This is not trivial, however. The compressed dictionary must have have the property that there exists an *efficient enumeration algorithm* which takes index i as input and outputs the i^{th} element of the dictionary. In the next section, we present such indexing algorithms for zero-order Markovian dictionaries, first-order Markovian dictionaries, deterministic finite automata (DFA), an arbitrary keyspace with some indexable superspace, and, finally, for hybrid Markovian/DFA dictionaries.

4. INDEXING ALGORITHMS

4.1 Zero-order Markovian dictionary

The dictionary we use is a slightly modified version of the zero-order Markovian filter, in which we only consider fixed-length strings. This is because we want to use different thresholds for different lengths. The hybrid algorithm in Section 4.5 will demonstrate how multiple dictionaries can be combined into one. The modified dictionary is $\mathcal{D}_{\nu,\theta,\ell} = \{\alpha : |\alpha| = \ell \text{ and } \Pi_{x\in\alpha}\nu(x) \geq \theta\}$

The key to the algorithm in this section is discretization of the probability distribution. To do this, we first rewrite the filter in an additive rather than multiplicative form: $\mathcal{D}_{\nu,\theta,\ell} = \{\alpha : |\alpha| = \ell \text{ and } \sum_{x\in\alpha}\mu(x) \geq \lambda\}$ where $\mu(x) = \log\nu(x)$ and $\lambda = \log\theta$.

Next, we discretize the values of μ to the nearest multiple of μ_0 for some appropriate μ_0. If we use a larger value for μ_0, we lower our memory requirement, but lose accuracy as well. In our experiments, we have chosen μ_0 such that there are about 1000 “levels” (see below).

Next, we define “partial dictionaries” $\mathcal{D}_{\nu,\theta,\ell,\theta',\ell'}$ as follows. Let α be any string such that $|\alpha| = \ell'$ and $\Pi_{x\in\alpha}\nu(x) = \theta'$. Then $\mathcal{D}_{\nu,\theta,\ell,\theta',\ell'} = \{\beta : \alpha\beta \in \mathcal{D}_{\nu,\theta,\ell}\}$.

Note that $\mathcal{D}_{\nu,\theta,\ell,\theta',\ell'}$ is well-defined because $\Pi_{x\in\alpha\beta}\nu(x) = \Pi_{x\in\alpha}\nu(x)\Pi_{x\in\beta}\nu(x) = \theta'\Pi_{x\in\beta}\nu(x)$. Therefore, it doesn't matter which α we choose. Intuitively, for any string prefix, the partial dictionary contains the list of all possible character sequences which could be appended to this prefix so that the resulting full string satisfies the Markovian property.

We now present a recursive algorithm to compute the size of a partial dictionary

$$|\mathcal{D}_{\nu,threshold,\ total\_length,\ level,\ current\_length}|$$

Observe that this algorithm is executed only once and only during the precomputation stage (rather than once for each key), and, therefore, its efficiency does not affect the cryptanalysis time. Here `mu` refers to the discretized version of the μ function above.

```
partial_size1(current_length, level)
{
    if level >= threshold: return 0
    if total_length = current_length: return 1

    sum = 0
    for each char in alphabet
        sum = sum + partial_size1(current_length+1,
            level+mu(char))
    return sum
}
```

Computation of $\mathcal{D}_{\nu,\theta,\ell,.,\ell'}$ depends on $\mathcal{D}_{\nu,\theta,\ell,.,\ell'+1}$. Thus the partial sizes are computed and stored in a 2-D array of size ℓ times the number of levels, the computation being done in decreasing order of ℓ'.

We are not particularly concerned about the efficiency of this algorithm because it is executed only during precomputation. Note, however, that running time (for computing all partial sizes) is linear in the product of the total length, number of characters and the number of levels.

The following is another recursive algorithm which takes as input an index into the keyspace and returns the corresponding key (this algorithm is executed during the cryptanalysis stage):

```
get_key1(current_length, index, level)
{
    if total_length = current_length: return ""

    sum = 0
    for each char in alphabet
        new_level = level + mu(char)
        // looked up from precomputed array
        size = partial_size1[
            current_length+1][new_level]
        if sum + size > index
            // '|' refers to string concatenation
            return char | get_key1(
                current_length+1,
                index-sum, new_level)
        sum = sum + size

    // control cannot reach here
    print "index larger than keyspace size"; exit
}
```

The `get_key` algorithm uses `partial_size` to determine the first character (this results in a value being looked up in the precomputed table of partial sizes), and then recurs on the index recomputed relative to the first character and the threshold adjusted based on the frequency of the first character.

To index into the entire keyspace, we call `get_key1` with $current\_length = 0$ and $level = 0$.

We note the similarity of the ideas used in this algorithm to the well-known Viterbi algorithm from speech processing[12].

4.2 First-order Markovian dictionary

As in the case of the zero-order model, we define length-restricted dictionaries and their partial versions. After reading a partial string, however, we now need to keep track of the last character because this time we are using digram frequencies.

```
partial_size2(current_length, prev_char, level)
{
    if level >= threshold: return 0
    if total_length = current_length: return 1

    sum = 0
    for each char in alphabet
        if current_length = 0
            new_level = mu(char)
        else
            new_level = level + mu(prev_char, char)
        sum = sum + partial_size2(current_length+1,
            char, new_level)
}

get_key2(current_length, index, prev_char, level)
{
    if total_length = current_length: return ""

    sum = 0
    for char in alphabet
        if current_length = 0
            new_level = mu(char)
        else
```

```
            new_level = level + mu(prev_char, char)
        size = partial_size2(current_length+1,
            char, new_level)
        if sum + size > index
            return char | get_key2(
                current_length+1,
                index-sum, char, new_level)
        sum = sum + size

    // control cannot reach here
    print "index larger than keyspace size"; exit
}
```

4.3 Deterministic finite automaton

This algorithm is similar to the algorithm for zero-order Markovian dictionaries, except that instead of levels and character frequencies we have states and state transitions. The `get_key3` algorithm is very similar to `get_key1` and is omitted.

```
partial_size3(current_length, state)
{
    if current_length = total_length
        if state is an accepting state: return 1
        else: return 0

    sum = 0
    for char in alphabet
        new_state = transition(char, state)
        if new_state is not NULL
            sum = sum + partial_size3(
                current_length+1, new_state)
    return sum
}
```

4.4 Any keyspace

We now describe an indexing algorithm for any keyspace $\mathcal{K}$, which works as long as there is an indexing algorithm for some superspace $\mathcal{K}' \supset \mathcal{K}$ and a testing procedure which, given $\alpha \in \mathcal{K}'$, decides whether $\alpha \in \mathcal{K}$. For instance, we can trivially index into (unfiltered) character sequences of a given length and test if a character sequence satisfies a Markovian filter, and, therefore, we can use this algorithm to index into Markovian dictionaries. The disadvantage is that precomputation involves enumerating $\mathcal{K}'$ via its indexing algorithm which might be prohibitively expensive if $\mathcal{K}$ is sparse in $\mathcal{K}'$. For instance, it is quite reasonable to consider 10-character password sequences with a first-order Markovian filter. This compresses the keyspace by a factor of 10^5, but to use the algorithm below to achieve this would involve iterating over a keyspace larger than 10^{14}. Furthermore, the indexing itself is not very efficient. On the other hand, it provides a good starting point because further keyspace-specific optimizations may be possible.

Given a parameter t, the algorithm divides the space $\mathcal{K}'$ into bins of size t, precomputes the number of members of $\mathcal{K}$ in each bin and stores them. When it gets an index, it quickly figures out which bin it falls into, iterates over all keys in $\mathcal{K}'$ in that bin and tests each one for membership.

Let $|\mathcal{K}'| = mt$.

```
compute_bins(t)
{
    count=0
    for i = 0 to m-1
        for j = i*t to i*t+(t-1)
            if the j'th key of K' belongs to K
                count = count+1
        bin[i] = count
}
```

For each i, this computes the cumulative counts for the first i bins.

```
get_key(index)
{
    i = binary_search(bin[], index)
    // i.e., bin[i] < index <= bin[i+1]

    count=0
    for j = i*t to i*t+(t-1)
        key = the j'th key of K'
            if key belongs to K
                count++
            if count = index - bin[i]
                return key
}
```

This algorithm requires $O(|\mathcal{K}'|)$ precomputation time, $O(\frac{|\mathcal{K}'|}{t})$ storage and $O(t + \log \frac{|\mathcal{K}'|}{t})$ indexing time. Observe that this algorithm is very similar to the algorithm for finding the nth prime [4].

4.5 Hybrid Markovian/DFA dictionary

Let $\mathcal{A}$ be the set of characters, and consider the combined dictionary $\mathcal{D}_{\nu,\theta,\ell_1,M,\ell_2} = \{\alpha : |\alpha| = \ell, \sum_{x\in\alpha, x\in\mathcal{A}} = \ell', \Pi_{x\in\alpha,x\in\mathcal{A}}\nu(x) \geq \theta, \text{ and } M \text{ accepts } \alpha\}$.

As mentioned earlier, our finite automaton works over the symbol set $\{A, a, n, s\}$. All lowercase characters are represented by `a`, all uppercase characters by `A`, all numerals by n and all special characters by s in the input to the automaton.

```
get_key5(index)
{
    count1 = partial_size1(0, 0)
    count2 = partial_size2(0, initial_state)

    index1 = index/count2  // quotient is truncated
    index2 = index - index1 * count2

    key1 = get_key1(0, index1, 0)
    // wlog we assume that key1 consists of
    // lowercase characters
    key2 = get_key2(0, index2, initial_state)

    key = ""
    pos = 1
    for char in key2:
        if char is 'a'
            append key1[pos] to key
            pos = pos+1
        if char is 'A'
            append uppercase(key1[pos]) to key
            pos = pos+1
        if char is neither 'a' nor 'A'
            append char to key

    return key
}
```

Essentially, this algorithm looks at the positions in the output of `get_key2` where alphabet characters are expected, and substitutes the string returned by `get_key1`. Combining an automaton with a first-order Markovian filter works similarly.

4.6 Multiple keyspaces

Finally, we show how multiple disjoint keyspaces can be combined into a single space.

```
get_key6(K1, K2, ... Kn, index)
{
    sum = 0
    for i = 1 to n
        if sum + size(Ki) > index
            return get_key(Ki, index-sum)
        sum = sum + size(Ki)

    // this cannot be reached
    print "index larger than sum of keyspace sizes"
}
```

4.7 Possible optimizations

In this section, we describe some optimizations that have not yet been implemented. The main criterion for the hybrid attack is that indexing should take less time than the hashing algorithm. So we want the hybrid algorithm to use about $50-100$ table lookups and the table must fit into the cache. The automaton algorithm is very fast because its input alphabet is very small; `get_key6` can be slow when there is a large number of keyspaces, but it can accelerated by precomputation of the cumulative sums and binary search for the appropriate keyspace. Our main concern, then, are the Markovian filters. We can speed up `get_key1` by reordering the characters so that the more frequent ones come first, reducing the average number of iterations to 6 (from 13, if the characters are ordered alphabetically). This reduces table lookups to less than 50 for 8-character strings. The same strategy works for `get_key2`, too.

5. EXPERIMENTS

Our first experiment involves measuring the coverage of Oechslin's rainbow attack vs. our hybrid attack. We used a database containing 142 real user passwords kindly provided by Passware (operator of `http://LostPassword.com`), which we believe to be a representative source. We used the search space of 6-character alphanumeric sequences (lowercase characters only) for the rainbow attack, which gives a keyspace size of $36^6 = 2.17 \times 10^9$. To model common password patterns, we created a set of around 70 regular expressions, which are listed in Appendix A.

The following table compares the number of passwords recoverd by the Rainbow attack vs. our attack.

Category	Count	Rainbow	Hybrid
Length at most 5	63	29	63
Length 6	21	10	17
Length 7	18	0	10
Length 8, A* or a*	9	0	6
Others	31	0	0
Total	142	39(27.5%)	96(67.6%)
only length ≥ 6	79	10(12.7%)	33(41.8%)

The effect of the probabilistic nature of the time-space tradeoff has been neglected, since the probability can be arbitrarily increased by increasing table size (and the dependence is the same for both attacks).

This experiment validates our basic hypothesis, but further experiments are needed. Passwords in our database may not have been representative of typical user passwords due to the way the database was compiled by Passware. We had access to the passwords when creating our regular expressions, although we did take the utmost care to write the expressions without using specific knowledge about the passwords in the database. A better experiment would involve a database containing only hashes of passwords, *e.g.*, the contents of an `/etc/passwd` file obtained from a system with a large, diverse user base. We are currently planing such an experiment, which would also involve larger storage for precomputed values, enabling search of larger keyspaces.

6. CONCLUSIONS

There are a variety of attacks against passwords, of which dictionary-based attacks are only one subclass. The simplest to deploy are social engineering attacks such as impersonation, bribery, phishing and login spoofing. Other attacks that directly exploit human vulnerabilities include shoulder surfing and dumpster diving. Password-based authentication systems appear particularly susceptible to protocol weaknesses, which can be exploited by keystroke logging, "Google hacking," wiretapping and side-channel attacks based on timing and acoustic emanations. Among dictionary-based attacks, it is worth mentioning that the United States Secret Service recently reported [20] success with custom dictionaries built from victim-specific information gleaned from analyzing their hard drives, including their documents, email messages, web browser cache and contents of visited websites.

Defending against offline and online dictionary attacks on human-memorable passwords is a difficult task. Possible techniques include graphical passwords [18, 36, 8], reverse Turing tests [31, 35], and hardening passwords with biometric information [26]. These techniques, however, require substantial changes in the authentication infrastructure. An interesting question for future research is whether human-memorable passwords drawn from non-textual spaces (*e.g.*, faces or geometric images) are vulnerable to attacks such as ours, based on filtering out unlikely candidates and very efficient enumeration of the remaining ones. Investigating this question will only be possible after one of the proposed methods is widely adopted and there is a significant body of such passwords chosen by real users.

Acknowledgement. We are grateful to Dmitry Sumin of Passware for providing the password material for our experiments.

7. REFERENCES

[1] MD5 online cracking using rainbow tables. `http://www.passcracking.com/`.

[2] M. Bellare, D. Pointcheval, and P. Rogaway. Authenticated key exchange secure against dictionary attacks. In *Proc. EUROCRYPT '00*, volume 1807 of *LNCS*, pages 139–155. Springer, 2000.

[3] S. Bellovin and M. Merritt. Encrypted key exchange: password-based protocols secure against dictionary attacks. In *Proc. IEEE Security and Privacy*

Symposium, pages 72–84. IEEE Computer Society, 1992.

[4] A. Booker. The Nth prime algorithm. `http://primes.utm.edu/nthprime/algorithm.php`, 2005.

[5] J. Borst, B. Preneel, and J. Vandewalle. On the time-memory tradeoff between exhaustive key search and table precomputation. In *Proc. 19th Symposium on Information Theory in the Benelux*, pages 111–118, 1998.

[6] V. Boyko, P. MacKenzie, and S. Patel. Provably secure password-authenticated key exchange using Diffie-Hellman. In *Proc. EUROCRYPT '00*, volume 1807 of *LNCS*, pages 156–171. Springer, 2000.

[7] W. Burr, D. Dodson, and W. Polk. Electronic authentication guideline. NIST Special Publication 800-63, 2004.

[8] D. Davis, F. Monrose, and M. Reiter. On user choice in graphic password schemes. In *Proc. 13th USENIX Security Symposium*, pages 151–164. USENIX, 2004.

[9] D. Denning. *Cryptography and Data Security*. Addison-Wesley, 1982.

[10] D. C. Feldmeier and P. R. Karn. UNIX password security - ten years later. In *Proc. CRYPTO '89*, volume 435 of *LNCS*, pages 44–63. Springer, 1989.

[11] A. Fiat and M. Naor. Rigorous time/space tradeoffs for inverting functions. In *Proc. STOC '91*, pages 534–541. ACM, 1991.

[12] G. D. Forney. The Viterbi algorithm. *Proceedings of the IEEE*, 61(3):268–278, 1973.

[13] C. Gentry, P. MacKenzie, and Z. Ramzan. Password authenticated key exchange using hidden smooth subgroups. In *these proceedings*, 2005.

[14] O. Goldreich and Y. Lindell. Session-key generation using human random passwords. In *Proc. CRYPTO '01*, volume 2139 of *LNCS*, pages 408–432. Springer, 2001.

[15] T. L. Griffiths and J. B. Tenenbaum. Probability, algorithmic complexity, and subjective randomness. In *Proceedings of the 25th Annual Conference of the Cognitive Science Society*, 2003.

[16] T. L. Griffiths and J. B. Tenenbaum. From algorithmic to subjective randomness. In *Advances in Neural Information Processing Systems 16*, 2004.

[17] M. Hellman. A cryptanalytic time-memory tradeoff. *IEEE Transactions on Information Theory*, 26:401–406, 1980.

[18] I. Jermyn, A. Mayer, F. Monrose, M. Reiter, and A. Rubin. The design and analysis of graphical passwords. In *Proc. 8th USENIX Security Symposium*, pages 135–150. USENIX, 1999.

[19] J. Katz, R. Ostrovsky, and M. Yung. Efficient password-authenticated key exchange using human-memorable passwords. In *Proc. EUROCRYPT '01*, volume 2045 of *LNCS*, pages 475–494. Springer, 2001.

[20] B. Kerbs. DNA key to decoding human factor. The Washington Post, March 28, 2005. `http://www.washingtonpost.com/wp-dyn/articles/A6098-2005Mar28.html`.

[21] K. Kusuda and T. Matsumoto. Optimization of time-memory trade-off cryptanalysis and its application to DES, FEAL-32 and Skipjack. *IEICE Transactions on Fundamentals*, E79-A(1):35–48, 1996.

[22] K. Kusuda and T. Matsumoto. Achieving higher success probability in time-memory trade-off cryptanalysis without increasing memory size. *TIEICE: IEICE Transactions on Communications/Electronics/Information and Systems*, 1999.

[23] R. Li and P. Vitanyi. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer, 1997.

[24] S. Lucks. Open key exchange: how to defeat dictionary attacks without encrypting public keys. In *Proc. Security Protocols Workshop*, volume 1361 of *LNCS*. Springer, 1997.

[25] G. A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review*, 63:81–97, 1956.

[26] F. Monrose, M. Reiter, and S. Wetzel. Password hardening based on keystroke dynamics. *International Journal of Information Security*, 1(2):69–93, 2002.

[27] R. Morris and K. Thomson. Password security: A case history. In *Communications of the ACM, Vol.22, No.11*, pages 594–597, 1979.

[28] P. Oechslin. Making a faster cryptanalytic time-memory trade-off. In *Proc. CRYPTO '03*, volume 2729 of *LNCS*, pages 617–630. Springer, 2003.

[29] Openwall Project. John the Ripper password cracker. `http://www.openwall.com/john/`, 2005.

[30] Openwall Project. Wordlists collection. `http://www.openwall.com/wordlists/`, 2005.

[31] B. Pinkas and T. Sander. Securing passwords against dictionary attacks. In *Proc. 9th ACM Conference on Computer and Communications Security (CCS)*, pages 161–170. ACM, 2002.

[32] L. R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[33] Zhu Shuanglei. Project RainbowCrack. `http://www.antsight.com/zsl/rainbowcrack/`, 2005.

[34] F. Standaert, G. Rouvroy, J.J. Quisquater, and J. Legat. A time/memory tradeoff using distinguished points: new analysis and FPGA results. In *Proc. CHES 2002*, volume 2523 of *LNCS*, pages 593–609. Springer, 2002.

[35] S. Stubblebine and P. van Oorschot. Addressing online dictionary attacks with login histories and humans-in-the-loop. In *Proc. Financial Cryptography*, volume 3110 of *LNCS*, pages 39–53. Springer, 2004.

[36] J. Thorpe and P. van Oorschot. Graphical dictionary and the memorable space of graphical passwords. In *Proc. 13th USENIX Security Symposium*, pages 135–150. USENIX, 2004.

APPENDIX

A. REGULAR EXPRESSIONS FOR COMMON PASSWORD PATTERNS

To simplify the regular expressions as well as the indexing algorithms, we specify length restrictions separately. Each regular expression has the input alphabet $\{A, a, n, s\}$, representing uppercase, lowercase, numeral and special characters, respectively. For each regular expression, we specify 4 numbers, which represent how many times the corresponding symbol type is permitted to occur in any accepted string. We also specify whether Markovian filtering is to be used, and if so, whether it should be zero- or first-order. Finally, we specify the threshold (recall that with Markovian filtering, we will only consider strings whose Markovian weight is above the threshold). The threshold is specified by stating what fraction of the keyspace should belong to the dictionary. The last column in the table is the size of the keyspace. It is not part of the keyspace specification. The size of the combined keyspace is the sum of the entries in the last column.

Determining the threshold for each regular expression is somewhat subjective, but we followed these general rules: no keyspace can have a size more than 10^8; the dictionary size should be 10% of the keyspace (based on Markovian filtering) unless the number of characters is 4 or less; in the latter case it should be 30% of the keyspace.

Regexes with 5 characters or less are not shown because they contribute too little to the keyspace size. We use the first-order Markov model whenever there are at least 6 alphabetical characters, all of them contiguous.

Regex	A	a	n	s	Fraction	Markov	Size
a*	0	8	0	0	0.000478	1	10^8
a*	0	7	0	0	0.0124	1	10^8
a*	0	6	0	0	0.1	1	3.09×10^7
A*	8	0	0	0	0.000478	1	10^8
A*	7	0	0	0	0.0124	1	10^8
A*	6	0	0	0	0.1	1	3.09×10^7
[Aa]*	1	6	0	0	0.00178	1	10^8
[Aa]*	1	5	0	0	0.0540	1	10^8
n*	0	0	8	0	1	-	10^8
n*	0	0	7	0	1	-	10^7
n*	0	0	6	0	1	-	10^6
a*n*	0	6	1	0	0.0324	1	10^8
a*n*	0	5	1	0	0.1	0	1.18×10^7
a*n*	0	4	2	0	0.3	0	1.37×10^7
a*n*	0	3	3	0	1	-	1.75×10^7
a*n*	0	2	4	0	1	-	6.76×10^6
a*n*	0	1	5	0	1	-	2.6×10^6
a*n*	0	1	6	0	1	-	2.6×10^7
A*n*	6	0	1	0	0.0324	1	10^8
A*n*	5	0	1	0	0.1	0	1.18×10^7
A*n*	4	0	2	0	0.3	0	1.37×10^7
A*n*	3	0	3	0	1	-	1.75×10^7
A*n*	2	0	4	0	1	-	6.76×10^6
A*n*	1	0	5	0	1	-	2.6×10^6
A*n*	1	0	6	0	1	-	2.6×10^7
a*A	1	6	0	0	0.0124	1	10^8
a*A	1	5	0	0	0.1	1	3.09×10^7
Aa*	1	6	0	0	0.0124	1	10^8
Aa*	1	5	0	0	0.1	1	3.09×10^7
Aa*n	1	5	1	0	0.0324	1	10^8
Aa*n	1	4	1	0	0.1	0	1.18×10^7
[An]*	5	0	1	0	0.1	0	7.13×10^7
[An]*	4	0	2	0	0.1	0	6.85×10^7
[An]*	3	0	3	0	0.284	0	10^8
[An]*	2	0	4	0	0.3	0	3.03×10^7
[An]*	1	0	5	0	1	0	1.56×10^7
[an]*	0	5	1	0	0.1	0	7.13×10^7
[an]*	0	4	2	0	0.1	0	6.85×10^7
[an]*	0	3	3	0	0.284	0	10^8
[an]*	0	2	4	0	0.3	0	3.03×10^7
[an]*	0	1	5	0	1	0	1.56×10^7
[As]*	6	0	0	1	0.0108	0	10^8
[As]*	5	0	0	1	0.1	0	2.97×10^7
[As]*	4	0	0	2	0.1	0	1.7×10^7
[As]*	3	0	0	3	1	-	4.38×10^7
[As]*	2	0	0	4	1	-	6.34×10^6
[As]*	1	0	0	5	1	-	4.88×10^5
[as]*	0	6	0	1	0.0108	0	10^8
[as]*	0	5	0	1	0.1	0	2.97×10^7
[as]*	0	4	0	2	0.1	0	1.7×10^7
[as]*	0	3	0	3	1	-	4.38×10^7
[as]*	0	2	0	4	1	-	6.34×10^6
[as]*	0	1	0	5	1	-	4.88×10^5