Probabilistic Passphrase Cracking

Luc Gommans
Radically Open Security

February 12, 2018

Abstract

Passphrases are an alternative to passwords that are supposed to be easier to remember for users. For passwords, cracking tools are widely available, helping us understand their strengths and weaknesses. For passphrase cracking, there is currently no common approach which is universally seen as the most effective way.

In this work, we attempted to find and implement the most effective method. We compared Markov chains, n-grams and a hybrid dictionary attack. The latter was found to be most effective, both in terms of efficiency and the number of results. Excluding the time required to build the dictionary from English Wikipedia and Wikiquote articles, it took only 10 minutes to crack over 2.3 million unique passphrases by using a hybrid dictionary attack with hashcat. Future users could download such a dictionary without having to compile it manually from various sources.

1 Introduction

Requirements for secure passwords are getting more and more stringent to stay ahead of the increase in computational capacity of commonly accessible systems. Using passphrases, users can approach the problem differently: instead of a complex sequence of characters, form a string of random, common words. The most famous example is 'correct horse battery staple': four words with no coherence, but many people in the field can recall this phrase without ever trying to memorize it[**own**]. Studies also show that users have fewer difficulties when memorizing a passphrase than when memorizing equally-strong random passwords[**behavioral-analysis**][**pwd-memorability**].

Password cracking tools, which help us understand the strength (or weakness) of passwords, are widely available. For passphrases, there is no commonly used approach using a single tool or dataset, probably because there are very few of them.

In the next section, we will explore the state of the field. Section 3, Problem Statement, describes our expected contribution. In order to obtain a set of passphrases to work with, we reproduced Labrande's previous work in section 4. Next, we explore possible probabilistic methods in section 5. Details about the implementation of the chosen method are set out in section 6 and an analysis of its performance in section 7. Finally, we describe our conclusions in section 8 and possible future work in section 9.

2 Related Work

Sparell and Simovits[**sparell-simovits**] created a passphrase cracking program by using Markov chains on a word level. Markov chains are used in many applications, such as next word prediction in keyboards on smartphones. Similarly, one can use them to approximate the next most likely word in a passphrase given a few starting words (which could be a dictionary walk). In Sparell and Simovits' work, the LinkedIn hash dump of 2012 was used as one of the test sets. This contains mostly passwords (not phrases), but is still frequently used due to its large size of 64 million hashes. Of these, 21 000 phrases of between 10 and 20 characters in length were cracked.

In previous work, we[**own**] generated passphrases from quotes of various well-known persons and song lyrics. Using this method on the LinkedIn set, we recovered 92 000 passphrases between 15 and 67 characters. A relatively large amount of time was spent on collecting this data, which we think could be optimized in future work. We also looked into whether users incorporate personal

data, such as lyrics specifically from a band they enjoy, but we have not found this to be the common case.

Labrande[**crackmeimfamous**] attempted to crack hashes from the Korelogic dump (139 million MD5 entries) using a compilation of different public sources. Around 4.3 million passwords were cracked, of which 200 000 are 16 characters or longer. Public sources include quotes from Wikiquote and text written in bold and italics on Wikipedia.

Bonneau and Shutova[**payphrase-properties**] examined ,,patterns of human choice in a passphrase-level authentication system deployed by Amazon". They ,,tested the availability of [over] 100,000 possible phrases at Amazon's registration page, which prohibits using any phrase already registered by another user." This system allows passphrases as short as two words and offered no explanation to users regarding what might be secure. Its interface merely asks for a 'phrase' (which is generally a string of *logical* words) and even suggests two-word phrases to use. The conclusion drawn by this paper is not surprising, given these circumstances: ,,users aren't able to choose phrases made of completely random words, but are influenced by the probability of a phrase occurring in natural language. Examining the surprisingly weak distribution of phrases in natural language, we can conclude that even 4-word phrases probably provide less than 30 bits of security which is insufficient against offline attack." We consider this low entropy estimation to be specific for Amazon PayPhrase.

Weir[**weir**] examined probabilistic methods for password cracking, specifically Markov chains and probabilistic context-free grammar. Such techniques might also be applied to passphrases. Besides examining different methods, they also created an implementation using probabilistic context-free grammar. This method cracked more passwords than John the Ripper in dictionary mode with the default word-mangling rules. It appears that this method is well-suited for password cracking.

3 Problem Statement

Our main research question is: *How can software efficiently generate likely passphrases, to be used in passphrase cracking?*

To achieve this, we will use a probabilistic method which generates realistic phrases. Which specific method to use, is part of this research. We want to use a probabilistic method because they can account for linguistic properties of phrases. It also allows users to download a model, rather than a large list of phrases to use directly. In section 5 we compare different methods.

Our implementation will be used in combination with traditional password cracking tools such as Hashcat or John the Ripper, because those have fast implementations of hash functions and other helpful features. One such feature is rule sets, which we will use for generating variants of phrases to be tried. These rule languages can perform simple manipulations such as lowercasing each word, removing spaces, or appending an exclamation mark.

Probabilistic methods require a dataset to be trained on. Because very few systems use passphrases exclusively, there are no lists of commonly used passphrases like there are for passwords. We will attempt to reproduce Labrande's research[**crackmeimfamous**] in order to create such a set. The result of this attempt is described in the next section.

4 Passphrase Dataset

In order to train a probabilistic algorithm, a dataset is needed to train on. Labrande described cracking over 4 million entries from the Korelogic[**korelogic**] MD5 dataset, 200 000 of which are 16 or more characters. This was done by downloading all articles of Wikipedia and Wikiquote in various languages, plus a few other, smaller sources. Page titles, list item entries, bold text, and italic text were used as phrases. Finally, they used rules in John the Ripper to test variants of each phrase. They also used rules that mangle the phrase into a password, for example by taking only the first letter of each word in the phrase.

We classify this attack as a hybrid dictionary attack, because of the similarity to traditional hybrid dictionary attacks on passwords. In suchs attacks, a large list of words is used together with mangling rules to generate possible permutations. In Labrande's variant, phrases are used instead of words, using a smart method of selecting likely phrases.

Since we are mainly interested in English passphrases, we downloaded only the English Wikipedia and Wikiquote articles. For a set of hashes, we used the LinkedIn as well as the Korelogic dump. This also provides a way of comparing results with Sparell and Simovits'[**sparell-simovits**] Markov chain-based method, which they tested on the LinkedIn set. We processed the data in a similar manner: extracting text in italics and bold, page titles, and list items on Wikiquote. We used only a few phrase mangling rules, which are described in section 6. Since we are, at this point, only interested in passphrases, we did not include many of the more elaborate rules: rules such as taking the first letter of each word or appending random numbers, were not used. Such random variations would defeat the purpose of a passphrase for the user, which is to be a set of common words with no punctuation or other random tweaks to remember.

The results of this effort are described in section 7.1 (evaluation of the hybrid dictionary attack).

Additionally, a large plain text password dump was downloaded, specifically the 'breach compilation' of 1.4 billion passwords. This proved to mainly consist of very insecure passwords and did not contain more than a few passphrases.

Note that the downloaded dumps did not contain usernames, and any cracked hash values are kept confidential.

5 Probabilistic Method Selection

Possible probabilistic approaches are Markov chains, weighted context-free grammar and n-grams. Such methods have been applied to password cracking in the past, and might be applied to passphrases as well. Markov chains is one example which has also been applied to passphrases[**sparell-simovits**] with good results.

So far, probabilistic context-free grammar has not been applied to passphrase cracking. It proved to be a viable method in Weir's[**weir**] password cracking work. For passphrases, it could be adapted to operate on whole words rather than individual characters. Based on texts and previously cracked passphrases, rules could be derived which indicate for example how likely it is for a noun to follow a verb.

In order to test whether this is a viable method, we attempted to classify the type of word in cracked passphrases with a python program which used the WordNet 3.0 database[**wordnet**]. Because not all passphrases separate words by spaces, the program would try to find words in phrases. The string 'correcthorsebattery' would result in the identified words 'correct', 'horse' and 'battery'. However, the WordNet database does not contain any conjugations or plurals. In the string 'hadadrink', it would not find the word 'had' (it only contains 'to have'). The first known word would be 'dad', after which it would recognize 'ink' as the next word. Incorrect identification of words following an incorrect identification was extremely common. Particularly in long words, it might recognize many small words incorrectly.

We evaluated the possibility of adding conjugations and plurals in some way, at least for the most common words, but this quickly proved to be too large a task to fit within the scope of this project.

While analyzing the dataset, it was found that it contained many geographical names, brands and people's names. Additionally, many of the phrases have the

ring of a catchphrase to them or form a common saying, for example "eye for an eye" or "give me a break". While a probabilistic context-free grammar model might be able to approximate those using large lists of names (and other words like verbs, adjectives, etc.), n-grams capture parts of phrases and will inherently become biased towards common phrases and constructions.

N-grams automatically include names and common constructions because they are subsets of sentences, and implementations can be very fast due to the simplicity of the algorithm. The downside is that it has no memory: if it learns 'there are' and 'are there' are common constructions, it will generate 'are there are', 'there are there'. Markov chains have the same issue, but context-free grammar does maintain more context (despite its name). We chose to implement a proof of concept using n-grams because it is a novel technique to generate passphrases and it seems fit for the purpose.

6 N-gram implementation

,,[An] *n-gram* is a contiguous sequence of n items from a given sample of text or speech."[**wiki-ngram**] In our case, each item is a word. N-grams of the order two and three and also called *bigrams* and *trigrams*, respectively.

A program was implemented in Python which splits a text on spaces and counts each possible subsentence of n words. For example, in the sentence 'lay of the land', trigrams are 'lay of the' and 'of the land'. The number of times an n-gram occurs is counted. Another Python program uses these n-grams to generate phrases, using the most frequently occurring n-grams first. The source code for these programs can be found in our git repository[**git**]. The output of the second program can be directly used in tools such as Hashcat.

The generator can make arbitrarily long phrases, as long as it can find a possible continuation in the n-gram file (which is typically infinite because of repetitions). The maximum length is therefore configurable. A logical value is around eight words: this does not output too many illogical repetitions while still generating fairly long phrases.

A bigrams file was initially generated from the passphrases that were cracked in section 4. This yielded only 761 bigrams and 331 trigrams because most users stripped their passphrases of spaces. We attempted to use the program mentioned in section 5 to find concatenated words within phrases, splitting e.g. 'batterystaple' into 'battery' and 'staple'. Again we found that the identification errors were too great to be of any use, even when using an additional dictionary[**wamerican**] which contains conjugations. (This dictionary was not used initially because it does not have word type classifications.)

In order to supplement the system with more possible sentence continuations, we used English Wikipedia articles. This yielded around 29 million bigrams and 47 million trigrams. Those were combined with the n-grams from the cracked passphrases, prioritizing the latter. This way, n-grams from Wikipedia are only used supplementarily and n-grams from previously cracked passphrases are tried first.

Finally, mangling rules were used in Hashcat to create simple variants for each phrase. Hashcat can use multiple rule sets which are then combined. If, for example, if in set 1 there are rules for both lowercasing and uppercasing a phrase and in set 2 there are rules for removing spaces, the resulting phrases for 'A b' will be 'a b', 'A B', 'ab' and 'AB'. In our previous work, we learned what kind of alterations users make to their phrases. Based on that information, we created three sets of rules:

1.
   - use as-is
   - lowercase all characters
   - capitalize the first letter, lowercase the rest
   - capitalize the first letter of each word, lowercase the rest (titlecase)
2.
   - use as-is
   - append a period
   - append an exclamation mark
   - append a question mark
3.
   - use as-is
   - remove all spaces

These rules create 32 variants for each input.

The system used for all tests is a laptop with an Intel i7-3630QM CPU and 8GiB RAM.

7 Evaluation

In this section the performance of different passphrase cracking methods is evaluated, considering both the number of entries they manage to crack and their efficiency.

Because we reproduced Labrande's hybrid dictionary attack (with slight alterations) and cracked a set of hashes identical to those of Sparell and Simovits'[**sparell-simovits**], the performance of all methods can be compared directly.

7.1 Hybrid dictionary

Using the hybrid dictionary attack as described by Labrande (with slight alterations), we cracked 2.3 million hashes from the Korelogic MD5 set. 146 742 were 16 characters or more. 311 815 included at least one space, meaning they are probably passphrase-like even if they are shorter than sixteen characters.

This is only half as many as Labrande cracked: 4.2 million and 200 000 of 16 characters or more (there are no statistics on the number of cracked entries with spaces). This difference could be explained by the fact that we were strictly after English passphrases and did not use a large number of mangling rules, and because we did not include other languages. Labrande used the English, German, French, Italian and Spanish versions of Wikipedia and Wikiquote.

Note that in previous work[**own**], we found that only about half the users left spaces intact in passphrases. This number is probably even lower in a set where users were not specifically queried about passphrases. The number of passphrase-like entries probably more than twice as high as the number of entries we found with spaces.

Of the LinkedIn set, we were able to crack 1.3 million entries, of which 13 361 are 16 characters or more and 26 345 cracked passwords include a space.

7.1.1 Efficiency

Labrande's hybrid dictionary attack took ,,a few weeks on an outdated laptop with a 2GHz processor," using John the Ripper. When reproducing this attack, the process took around 10 minutes in Hashcat. Our dictionary was smaller (26 million instead of 65 million entries) and used fewer mangling rules, which could explain the difference. Despite those limits, we still cracked half as many entries as Labrande. Despite the more than 2000-fold decrease in time, it still yielded good results.

The storage requirements of this method are relatively large: our phrases file (with 26 million entries) is 690MiB. Since this was generated from Wikipedia, all Wikipedia and Wikiquote articles also had to be downloaded, which total 26GiB as download and 135GiB uncompressed.

Memory usage is not relevant for this attack, as each line can be read off of disk and is immediately passed onto Hashcat.

7.2 Markov chains

Sparell and Simovits cracked 25 000 entries from the LinkedIn set, of which 384 are 16 characters or longer.

7.2.1 Efficiency

Sparell and Simovits described that generating phrases is the dominant part of the time consumption for their cracking process. They therefore wrote the output of their phrase generator to a file before inputting it into Hashcat, classifying files by different sources and passphrase lengths. Generation times differ from 16 minutes for a 2.6 million 10-character phrases file to 960 hours (40 days) for 8.5 billion 20-character phrases. The numbers in terms of phrases per second vary greatly, ranging from 22 500 to 2 459 phrases per second.

The size of the source files, which form the Markov model, is not described; nor is the amount of RAM required for this method.

7.3 N-grams

Using the passphrase generator with trigrams, it was able to crack 835 562 entries of the Korelogic MD5 set, of which 32 911 are 16 characters or longer, and 70 198 contain a space. Of the LinkedIn set, it was able to crack 482 014 entries, of which 3 782 are 16 characters or longer and 14 903 contain a space.

Using bigrams, it was able to crack 342 145 entries of the Korelogic set, of which 9 148 are 16 characters or longer and 1 023 contain a space. Of the LinkedIn set, it was able to crack 190 117 entries, of which 1 468 are 16 characters or longer and 1 449 contain a space.

Both attacks produced a great number of short results, such as 'thefault' or 'wanderers'. After going through all phrases shorter than or equal to n words, the number of results dropped significantly.

7.3.1 Efficiency

The n-gram phrase generator first loads the data, which takes a few minutes and scales linearly with the n-gram file size. While running, it generates two to four million phrases per second on a single CPU core (the median is 3.3 million phrases per second). This saturated Hashcat's ability to process input: hashcat

processes around ten million SHA-1 hashes per second, but applies a few rules. Since rules easily generate many more than three variants per phrase, the speed of our program is sufficient despite running on only a single core and being written in Python, an interpreted language.

Memory usage for the generation process depends on the number of n-grams loaded: for 22 million trigrams (464MiB), it uses 5.3GiB of RAM. This is quite intensive, but required for fast lookups. Bigrams are much more efficient: because there are fewer combinations to be made, the input files are much smaller. The complete bigrams file contains 2.9 million entries (47MiB) and uses 800MiB RAM.

A file with trigrams of 464MiB is not much smaller than our Wikipedia and Wikiquote phrase dictionary of 690MiB. Bigrams are more efficient, using 47MiB.

7.4 Overview

Table 1 gives an overview of the performance of the different methods. The column headers 'Dict-1' and 'Dict-2' indicate the hybrid dictionary attack, Labrande's and ours, respectively. The rows labeled '16+' indicate the number of results which are 16 characters or longer.

Note that for our hybrid dictionary attack, the number of generated phrases per second is estimated at 10 million. Contrarily to all other methods, this method is IO-bound instead of CPU-bound. The amount of RAM required is considered negligible, as it only requires IO buffers.

	Dict-1	**Dict-2**	**Markov**	**3-grams**
LinkedIn	n/a	1 300 000	25 000	482 014
LinkedIn 16+	n/a	13 361	384	3 782
Korelogic	4 200 000	2 300 000	n/a	835 562
Korelogic 16+	200 000	146 742	n/a	32 911
Phrases/second	n/a	10 000 000	22 500	3 300 000
RAM	n/a	negligible	n/a	5.3GiB
Storage	n/a	690MiB	n/a	464MiB

Table 1: Performance overview

8 Conclusions

We have explored and compared different methods for passphrase cracking, confirmed the results of Labrande's hybrid dictionary attack, and implemented a new method based on n-grams. Our conclusion is that the most effective attack at this time is the hybrid dictionary attack, even when considering the trade-off between storage requirements and practicality versus the number of results obtained. While the dictionary does need a relatively large amount of storage, particularly when generating or updating it, it is extremely efficient in terms of RAM and CPU.

N-grams are less effective than expected. The most effective part of the cracking process were the phrases of n words or fewer, corresponding to the order of the n-gram. This means that literal hits were much more common than hits of combined n-grams, indirectly confirming again how powerful the dictionary attack is. It also indicates that passphrases are generally not normal, full sentences: despite training the model on both previously cracked passphrases and all English Wikipedia articles, sentences generated from those n-grams do not yield many results.

Markov chains, as described by Sparell and Simovits, yielded the fewest results of the three methods we compared. Generating the passphrases was also slower than by using n-grams or a dictionary, probably because the algorithm for Markov chains is more complex.

The source code for our software is available on Github[**git**] under GPL version 3.0.

9 Future work

We have not been able to explore probabilistic context-free grammar fully due to the difficulties in classifying words. Weir was successful in this with passwords because he focused on a character level. For passphrases, which are composed of words instead of characters, this method is more complex. It is recommended to look into other, possibly commercial options for a word classification database. It is also recommended to avoid using previously cracked passphrases as training set: many are stripped of spaces, making it very difficult to programmatically split words for classification.

Our n-gram attack can be improved in various ways. It might be more effective to store which n-grams were the start of a sentence. In our version, it would take an n-gram and append other, matching n-grams to complete the phrase.

However, if the starting n-gram is not a logical way to start a sentence, all generated continuations are wasted. It would also be more memory- and storage-efficient to use a language with less memory overhead, and to identify words by tokens instead of indexing literal n-grams.

Finally, we are curious how neural networks would perform for this purpose. Training a model is typically computationally intensive, but general purpose frameworks are available which make it easy to run such computations on a GPU and generate fairly large models on modest hardware.