Count-Min Sketches for Estimating Password Frequency within Hamming Distance Two

Leah South and Douglas Stebila

School of Mathematical Sciences, Queensland University of Technology, Brisbane, Queensland, Australia
`leah.south@connect.qut.edu.au`, `stebila@qut.edu.au`

Abstract. The *count-min sketch* is a useful data structure for recording and estimating the frequency of string occurrences, such as passwords, in sub-linear space with high accuracy. However, it cannot be used to draw conclusions on groups of strings that are similar, for example close in Hamming distance. This paper introduces a variant of the count-min sketch which allows for estimating counts within a specified Hamming distance of the queried string. This variant can be used to prevent users from choosing popular passwords, like the original sketch, but it also allows for a more efficient method of analysing password statistics.

Keywords: count-min sketch, Bloom filter, password frequency, approximate string matching

1 Introduction

The use of passwords for identity verification is widespread. There is a long line of research on analyzing the security and guessability of password [8, 3]. In large online systems that we see on the Internet today, an important characteristic that affects the overall security of the system is that passwords within the system should not be too *popular*. In an ideal setting, of course, users would create a unique password that is hard to guess, and not popular, so that only that user and no one else could access their account. However, users tend to choose passwords that are easy to remember and familiar to them, such as dictionary words, or perhaps strings associated with the system in question. This tendency means that certain passwords are used with higher frequency, making them popular.

If an attacker knew the distribution of passwords, they could use its statistics and guess the most popular passwords first. This is known as the *statistical guessing technique* [9]. When there is a high percentage of popular passwords, the attacker can compromise a high percentage of accounts. For example, the 2009 breach of RockYou.com's 32 million account password database showed that the most popular password (`123456`) was used by 0.9% of all accounts, and the next 4 most popular passwords (`12345`, `123456789`, `password`, and `iloveyou`) were used by another 0.8% of all accounts. Clearly, the system was not screening passwords for popularity. As a result, a statistical guessing attack would lead to millions of accounts being compromised. In order to prevent a successful statistical guessing attack such as this, it is common to limit the number of guesses; more recently, it has been proposed [9] to limit the popularity of passwords: when users try to set their password to a string that is used in a percentage of accounts above some threshold, it is rejected and the user is required to choose another password.

In order to keep track of password popularity, some sort of system which counts passwords must be used. Online sites with a large number of users are best suited for systems which restrict popular passwords, as such sites are at high risk of *trawling attacks*, in which attackers aim to guess the passwords to many accounts without targeting any single account.

How can we store password information in a way that allows us to calculate frequency when users attempt to register a password? The simplest technique for calculating frequency during password registration would be to store a separate table of passwords along with their frequency. This is undesirable both for efficiency reasons (since the size of the table grows linearly in the number of distinct passwords) and for security reasons (since it immediately provides an attacker with the full distribution of passwords). Best-practice recommendations for storing passwords for login involve storing salted password hashes for each account; the set of such passwords does not admit statistical analysis since, by design, the hash of the same password under different salts yields different, seemingly independent, outputs, thus yielding no information about the frequency with which a password is used.

1.1 Bloom Filters and Count-Min Sketches

The *Bloom filter* [2] can be used [10] to store in sub-linear space a table representing a dictionary of prohibited passwords. The system is setup as follows. A $w \times h$ table T of bits is used, along with h independent hash functions $\mathsf{hash}_1, \ldots, \mathsf{hash}_h : \{\mathsf{A}-\mathsf{Z}, \mathsf{a}-\mathsf{z}, \mathsf{0}-\mathsf{9}, \ldots\}^* \to \{1, \ldots, w\}$. Each word x in the dictionary of prohibited passwords is hashed under each hash function hash_k, and the entry $T_{\mathsf{hash}_k(x),k}$ is set to 1. When a proposed password y is to be tested for membership in the list of prohibited passwords, if all of the values $T_{\mathsf{hash}_k(y),k}$ are equal to 1, then y is deemed to be prohibited, but if at least one of those table entries is zero, then y is not prohibited. There are no *false negatives*, meaning that it is impossible for a password that is prohibited to not be recognized as such, but there may be *false positives*, meaning that some passwords that are not prohibited may, due to collisions on all rows, still be identified by the table as prohibited. Assuming the hash functions are independent random functions, the false positive rate $(1-(1-\frac{h}{w})^N)^h$, where N is the number of prohibited passwords originally added to the table [10].

The *count-min sketch* [5] enhances the Bloom filter by storing a table of integers, not bits. The $\mathsf{update}(x, c)$ function records that string x has been used c more times by adding c to each table entry $T_{\mathsf{hash}_k(x),k}$. The $\mathsf{estimate}(x)$ function returns an estimate on the number of times that string x has been used by computing $\min\{T_{\mathsf{hash}_k(x),k} : 1 \leq k \leq h\}$. The use of count-min sketches for recording password frequency was proposed by Schechter, Herley, and Mitzenmacher [9], who propose preventing users from registering with passwords whose current popularity is above a certain threshold. As with the Bloom filter, false negatives cannot occur, meaning that for any password x it is impossible for the $\mathsf{estimate}(x)$ to return a value lower than the sum of the c over all calls of $\mathsf{update}(x, c)$. However, false positives may still occur, meaning that $\mathsf{estimate}(x)$ may over-estimate the frequency of password x, due to collisions across all hash functions. For a single hash, the expected error due to collisions is N/w, where N is the total sum of the counts of all passwords; this is because the total count of N will spread approximately evenly across all w columns in the table. By the Markov inequality, the error of the count min-sketch (the minimum across all hash functions) is at most $2N/w$ with probability at least $1 - (\frac{1}{2})^h$.

1.2 Contributions

When used for passwords, the Bloom filter and the count-min sketch can be useful in prohibiting certain passwords or limiting the frequency with which any password is registered. However, they cannot accommodate any relation between 'similar' passwords. For example, a user who tries to register the password `password` and finds that it is prohibited may try again with `p@ssword` or `passw∅rd`. An attacker, knowing this prevalence for 'leetspeak', may also make use of these similarities in an attack strategy that targets 'almost popular' passwords. Since the Bloom filter and count-min sketch use independent random passwords, they lose any semantic connection between such similar strings.

In this work, we explore a variant of the count-min sketch that allows one to compute estimates not only for the given string but for all strings within a fixed Hamming distance of the given string. This technique allows a system to detect frequently used passwords at the registration phase. Our technical approach is to introduce 'wildcard' characters and then record and estimate based on all strings that can be constructed with wildcard characters within the required Hamming distance. This technique imposes a computational overhead of $\binom{\ell}{d}$, where ℓ is the length of the password in question and d is the maximum Hamming distance. This compares favourably with the naive technique which would have a computational overhead of $\binom{\ell\alpha}{d}$, where α is the size of the alphabet. We provide false positive error rates as well, and compared with the naive method our technique provides better accuracy for the same size table for a wide range of password lengths. Our technique can naturally be extended to higher Hamming distances.

2 Related Work

While the Bloom filter and count-min sketch are widely used, there are several other methods to store estimate counts, some of which may be useful in obtaining statistics on groups of similar passwords.

There have been some systems in the past which have stored cleartext passwords alongside their counts. This method is mostly outdated due to the lack of security it provides. If attackers gain access to password databases such as these, a highly successful statistical guessing attack can be carried out because the actual passwords are visible and there are no false positives - the exact counts are known. Using hash functions is preferable to this method.

Decision trees, as suggested by Bergadano, Crispo and Ruffo, can be used to determine password membership. These decision trees consist of nodes for particular attributes, arcs for values of these attributes and leaves for classification of whether or not the password has been used before in that database [1]. When using decision trees, there is a training phase in which the nodes, arcs and leaves are chosen to produce the best results. For each update to the database, the training phase must be repeated. Due to this, decision trees can be used for determining membership but are not as practical when frequent updates may be involved.

There has also been some work on the topic of determining popularity of similar words. This includes work which allows for checking membership of words within Levenshtein distance 1, that is words which have only one insertion, deletion or substitution. Manber and Wu [7] suggested an approach similar to the original Bloom filter, with the main difference being that the membership of words within Levenshtein distance 1 are checked in the estimation stage. It is proposed that if any password within distance 1 appears to be positive, the password cannot be used [7]. As with all Bloom filter-based techniques, this only records the binary data of whether two similar passwords have been used, not counts on how frequently similar passwords are used.

3 Construction

The adaptation of the original count-min sketch that is introduced in this paper allows for the popularity estimations of words within Hamming distance zero to two, that is words that differ by up to two substitutions. Like the count-min sketch, this adaptation consists of three main parts: *hashing* the passwords, *updating* the table and *estimating* the counts. In Appendix A, we provide a worked example of each of the three operations—hashing, updating, and estimating.

3.1 Hash

We represent passwords x as integers X. Any canonical representation of a word as an integer will suffice. For expository convenience, we can imagine a mapping of an ℓ-letter word $x = x_1 \dots x_\ell$ where each character x_i is coded as a two-digit integer $X_i \in \{00, \dots, 98\}$; the integer 99 is reserved to represent a wildcard character $*$. The vector $\langle X_1, \dots, X_\ell \rangle$ is then viewed as an integer $X = \sum_{i=1}^{\ell} 100^{i-1} X_i$. This suffices to encode for example all passwords that can be typed using characters on a standard US keyboard.

As in [6], each hash function hash_k is a Carter–Wegman 2-universal hash function [4] of the form

$$\mathsf{hash}_k(x) = (((a_k X + b_k) \bmod p_k) \bmod w) + 1 \ ,$$

where X is the canonical integer representation of the word x, p_k is a prime number much larger than w (recall w is the width of the table), say $p_k = 2^{31} - 1$ or $p_i = 2^{61} - 1$, and a_k and b_k are chosen uniformly at random from $\{0, \ldots, p_k - 1\}$. We can simplify to using the same prime $p = p_1 = \cdots = p_h$ across all hash functions, but all a_k and b_k need to be chosen independently to ensure negligible probability of collisions; otherwise the benefits of using multiple hash functions will be eliminated.

We define the vector-wise hash function $\mathsf{hash}(x)$ where the kth entry of $\mathsf{hash}(x)$ consists of $\mathsf{hash}_k(x)$ with a_k and b_k.

3.2 Update

The function $\mathsf{update}(T, x, a, b, c) \rightarrow T'$ updates the sketch based on the previous table T, the password x to be updated, the values or vectors of a and b for the hash functions and the amount of times c that the password is being added. The update algorithm in this work differs greatly to that in the traditional count-min sketch.

Firstly, the count for the actual password is updated. Like the original count-min sketch, this is done by finding the hash of the password for each hash function then updating the count at these positions in the table. Next, the updates are done for words within distance 1. To do this, a 'wildcard character' is introduced. This wildcard character, which for our canonical encoding above is denoted by 99, is used to group similar passwords together: for example if the word `abcd`, or $\langle \mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d} \rangle$, is represented in integer form as $\langle 1, 2, 3, 4 \rangle$, then $\langle 99, 2, 3, 4 \rangle$ represents `*bcd`, all four letter words ending in `bcd`. A wildcard character indicates that any character could go in its place. After the count of the actual password is updated, all wildcard passwords within Hamming distance 1 are updated by creating $\ell = \text{length}(x)$ variations of the password, each with a different character from the initial word replaced with 99, then hashing these variations and updating their positions in the table. Next, the wildcard words within Hamming distance 2 of the password x are hashed and their positions in the table are updated.

In Algorithm 1, update, using the wildcard character means that ℓ additional updates are done for adding wildcard words within distance 1. The alternative, naive technique, would be to search all $\alpha\ell$ passwords within distance 1 during the estimate algorithm, where α is the size of the alphabet. When the password length $\ell \ll \alpha$, the proposed technique is much more efficient than the naive technique.

3.3 Estimate

The function $\mathsf{estimate}(T, x, a, b)$ is used to obtain estimates of the count of passwords at *exactly* Hamming distances 0, 1, and 2 of x and is specified in full in Algorithm 2.[1]

[1] Note that we have $\mathsf{estimate}$ return estimates for counts of passwords at exactly, rather than within, the specified Hamming distance for clarity of exposition; estimates

Algorithm 1 $\mathsf{update}(T, x, a, b, c)$

```
1: $T' \leftarrow T$
2: $H \leftarrow \mathsf{hash}(x, a, b)$
3: for $k = 1$ to $h$ do
4:     $T'_{H_k,k} \leftarrow T'_{H_k,k} + c$
5: for $i = 1$ to $\ell = \text{length}(x)$ do
6:     $x' \leftarrow x$
7:     $x'_i \leftarrow 99$
8:     $H' \leftarrow \mathsf{hash}(x', a, b)$
9:     for $k = 1$ to $h$ do
10:        $T'_{H'_k,k} \leftarrow T'_{H'_k,k} + c$
11: for $i = 1$ to $\ell - 1$ do
12:     for $j = i + 1$ to $\ell$ do
13:         $x'' \leftarrow x$
14:         $x''_i \leftarrow 99$
15:         $x''_j \leftarrow 99$
16:         $H'' \leftarrow \mathsf{hash}(x'', a, b)$
17:         for $k = 1$ to $h$ do
18:             $T'_{H''_k,k} \leftarrow T'_{H''_k,k} + c$
19: return $T'$
```

At distance 0. When the specified Hamming distance d is zero, that is, when the count of the actual word is desired, the estimate process is very similar to the original count-min sketch. The h different hashes of the password and the counts at all h positions in the table are found. These counts can then be put into a vector est^0 of length h, where each element represents the count at a different hash function. The resulting estimate $\overline{\mathsf{est}}^0$ is the minimum of these counts.

At distance 1. At a maximum Hamming distance of one, the process is slightly more complex. First, estimates est^1_i, $i = 1, \ldots, \ell$, for the frequency of words that may differ from x in the ith character; each of these estimates can be found using the technique estimating the (distance 0) occurrences of the wildcard word. In other words, est^1_i is the (distance 0) estimate for the wildcard "word" x' where the ith character of x has been replaced with the special wildcard character. Once estimates est^1_i for the frequency of words that may differ from x in the ith character have been found, we can find an estimate $\overline{\mathsf{est}}^1$ for the number of occurrences of words at distance 1 from x by summing est^1_i for all i, then subtract $\ell\overline{\mathsf{est}}^0$.

The reason for the subtraction is as follows. As explained in the update section, when a single password x is added into the database, the counts for the exact password and the counts for all $\ell = \text{length}(x)$ passwords within distance 1 are all increased. In the estimation stage, the counts for all passwords with one wildcard character, or within distance 1, are summed together. This means that the count

within the specified distance can be found by summing the estimates exactly at all distances less than or equal to.

for the exact password has been included ℓ times in the calculation, when it should only have been included once. To overcome this problem, the count of that exact password multiplied by the length of the password is subtracted.

At distance 2. The frequency estimate $\overline{\mathsf{est}}^2$ for passwords at distance 2 is found in a similar manner to that of distance 1. For each of the $\binom{\ell}{2}$ wildcard passwords of x that may differ in the ith and jth characters, we compute (distance 0) estimates $\mathsf{est}^2_{i,j}$ of the wildcard "word" x'' where the ith and jth characters of x have been replaced with the special wildcard character. We then sum these estimates of words within distance 2. However, again we have overcounted. The words at precisely distance 1 have been counted $\ell - 1$ times. Similarly, the word at precisely distance 0 has been counted $\binom{\ell}{2}$ times.

Algorithm 2 $\mathsf{estimate}(T, x, a, b)$

```
1: $H \leftarrow \mathsf{hash}(x, a, b)$
2: for $k = 1$ to $h$ do
3:     $\mathsf{est}^0_k \leftarrow T_{H_k,k}$
4: $\overline{\mathsf{est}}^0 \leftarrow \min_k\{\mathsf{est}^0_k\}$
5: for $i = 1$ to $\ell = \mathrm{length}(x)$ do
6:     $x' \leftarrow x$
7:     $x'_i \leftarrow 99$
8:     $H' \leftarrow \mathsf{hash}(x', a, b)$
9:     for $k = 1$ to $h$ do
10:        $\mathsf{est}^1_{i,k} \leftarrow T_{H'_k,k}$
11:    $\mathsf{est}^1_i \leftarrow \min_k\{\mathsf{est}^1_{i,k}\}$
12: $\overline{\mathsf{est}}^1 \leftarrow \left(\sum_i \mathsf{est}^1_i\right) - \ell\,\overline{\mathsf{est}}^0$
13: for $i = 1$ to $\ell - 1$ do
14:    for $j = i + 1$ to $\ell$ do
15:        $x'' \leftarrow x$
16:        $x''_i \leftarrow 99$
17:        $x''_j \leftarrow 99$
18:        $H'' \leftarrow \mathsf{hash}(x'', a, b)$
19:        for $k = 1$ to $h$ do
20:            $\mathsf{est}^2_{i,j,k} \leftarrow T_{H''_k,k}$
21:        $\mathsf{est}^2_{i,j} \leftarrow \min_k\{\mathsf{est}^2_{i,j,k}\}$
22: $\overline{\mathsf{est}}^2 \leftarrow \left(\sum_{i,j} \mathsf{est}^2_{i,j}\right) - (\ell - 1)\,\overline{\mathsf{est}}^1 - \binom{\ell}{2}\,\overline{\mathsf{est}}^0$
23: return $\langle \overline{\mathsf{est}}^0, \overline{\mathsf{est}}^1, \overline{\mathsf{est}}^2 \rangle$
```

4 Analysis of Construction

4.1 Error

Like the original count-min sketch, collisions can occur in this adaptation. These collisions result in some error, causing either slight overestimations of counts or false positives.

The error in estimating exact words can be expressed fairly simply. In determining this error relative to the original sketch, it is important to note that the total count for this method is not the same as what it would be in the original count-min sketch. In this adaptation of the sketch, the total count across the sketch is $(1 + \ell + \binom{\ell}{2})$ times larger than the actual total of passwords used. This additional count is the result of the update stage, in which the password itself, all ℓ passwords within distance 1 and all $\binom{\ell}{2}$ combinations of passwords within distance 2 are added into the sketch. Let $\hat{N} = (1 + \ell + \binom{\ell}{2})N$ denote the total count on the new sketch, where N is the total number of (non-distinct) passwords entered. In general, hash outputs are expected to spread approximately evenly across all columns of the table, making the average count per hash the quotient of the total sum of counts and the width w of the table: $\hat{N}/w$. The expected maximum error is therefore $\hat{N}/w$, which may occur, for example, if an estimate is done on a password which is not present but happens to have the same hash across all h hash functions as other passwords which have a count of $\hat{N}/w$.

The expected maximum error for estimating counts of passwords at distance 1 can be derived from the formula for distance 1 in Algorithm 2:

$$\mathrm{error}(\overline{\mathsf{est}}^1) \leq \mathrm{error}\left(\sum_i \mathsf{est}_i^1\right) + \mathrm{error}(\ell\, \overline{\mathsf{est}}^0) \ .$$

Since each term of $\sum_i \mathsf{est}_i^1$ is calculated as distance-0 estimate, the error of each term is the same as the error in a distance-0 estimate, and thus

$$\mathrm{error}(\overline{\mathsf{est}}^1) \leq \ell\, \mathrm{error}(\overline{\mathsf{est}}^0) + \mathrm{error}(\ell\, \overline{\mathsf{est}}^0) \leq 2\ell\, \mathrm{error}(\overline{\mathsf{est}}^0) = 2\ell \frac{\hat{N}}{w} \ .$$

For passwords at distance 2, there is another increase in error. Based on the estimate for words at distance 2 in Algorithm 2, the expected maximum error is

$$\mathrm{error}(\overline{\mathsf{est}}^2) \leq \mathrm{error}\left(\sum_{i,j} \mathsf{est}_{i,j}^2\right) + \mathrm{error}((\ell - 1)\, \overline{\mathsf{est}}^1) + \mathrm{error}\left(\binom{\ell}{2} \overline{\mathsf{est}}^0\right) \ .$$

Since each term of $\sum_{i,j} \mathsf{est}_{i,j}^2$ is calculated as distance-0 estimate, the error of each term is the same as the error in a distance-0 estimate, and thus

$$\mathrm{error}(\overline{\mathsf{est}}^2) \leq \binom{\ell}{2} \mathrm{error}(\overline{\mathsf{est}}^0) + \mathrm{error}((\ell - 1)\, \overline{\mathsf{est}}^1) + \mathrm{error}\left(\binom{\ell}{2} \overline{\mathsf{est}}^0\right) \ .$$

Simplifying and substituting, we get

$$\begin{aligned}
\text{error}(\overline{\mathsf{est}}^2) &\le \binom{\ell}{2}\text{error}(\overline{\mathsf{est}}^0) + (\ell-1)\,\text{error}(\overline{\mathsf{est}}^1) + \binom{\ell}{2}\text{error}(\overline{\mathsf{est}}^0) \\
&\le \left(\binom{\ell}{2} + (\ell-1)(2\ell) + \binom{\ell}{2}\right)\text{error}(\overline{\mathsf{est}}^0) \\
&= \left(2\binom{\ell}{2} + 2\ell(\ell-1)\right)\text{error}(\overline{\mathsf{est}}^0) \\
&= \left(2\binom{\ell}{2} + 4\binom{\ell}{2}\right)\text{error}(\overline{\mathsf{est}}^0) \\
&= 6\binom{\ell}{2}\text{error}(\overline{\mathsf{est}}^0) = 6\binom{\ell}{2}\frac{\hat{N}}{w}\ .
\end{aligned}$$

4.2 Comparison with Naive Method

The purpose of the adaptation is to provide a more effective method of estimating counts of passwords within specified Hamming distances. While the count-min sketch can be used to do this, this *naive method* using the original sketch—incrementing the entry for every one of the $\alpha\binom{\ell}{d}$ words within distance d in an alphabet of size α—would be less efficient and have a higher error rate.

In this section, a comparison of efficiency and error is carried out. All of the following graphs have the same response variable: the expected average error as a multiple of $\frac{N}{w}$ where N is the total number of passwords entered (also the total count in the original sketch) and w is the width of the table.

Estimates at distance 0. When estimating how many times a specific word appears in a database (i.e. estimating passwords at distance zero), the original count-min sketch is more effective. The expected maximum error in estimating counts is $\frac{N}{w}$ in the original method and $\frac{\hat{N}}{w}$, or $(1+\ell+\binom{\ell}{2})N/w$, in this variation. Therefore the expected maximum error for estimating counts at distance zero will always be $1+\ell+\binom{\ell}{2}$ times larger in this variation. However, this increase in error is not a major problem since this sketch was not designed for estimating exact word counts.

Estimates at distance 1. The benefits of using this modified sketch are more evident when estimating counts at distance 1 of a password. In order to estimate counts at distance 1 of a specified password using the original sketch, multiple estimates would have to be done. More specifically, the number of estimates would be $\alpha\ell$, where α is the size of the alphabet and ℓ is the length of the password. Since the error in estimating each individual password is $\frac{N}{w}$, the expected maximum error in estimating $\alpha\ell$ passwords is $\alpha\ell N/w$. With the new method, the expected maximum error in estimating words at distance 1 is $2\ell\hat{N}/w = 2\ell(1+\ell+\binom{\ell}{2})N/w$. A comparison of these errors can be seen in Figure 1, where the size of the alphabet has been set to $\alpha = 72$ (26 uppercase and 26 lowercase letters, 10 numbers, and 10 special characters). From this graph, it can be seen that the error for the proposed method is lower until the length of the password reaches nine characters.

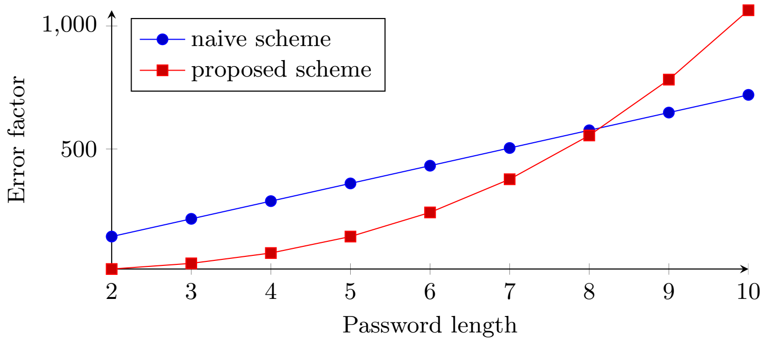

Fig. 1. Error as a multiple of N/w for passwords at distance **1**, using a count-min sketch supporting maximum distance of **2**, compared with naive scheme.

Note that this calculation above assumes that the `update` stage is as described in Section 3.2, allowing for estimates within distance 2, even though we are currently estimating distance 1. If the sketch is *not* going to be used to estimate counts at distance 2 at all, then it is possible to remove the distance 2 section. Updates for passwords at distance 2 would then not be included, making the expected maximum error for words at distance 1 equal to $2\ell(1+\ell)N/w$. A graph comparing the expected maximum error using the traditional method and this variation in which only distance 1 is included can be seen below, in Figure 2. For this graph, the size of the alphabet is again $\alpha = 72$. The proposed method has a lower error than the naive method until the length of the password reaches 36 characters.

Estimates at distance 2. For passwords at distance 2, $\binom{\ell}{2}\alpha$ estimates would have to be done using the original sketch. This makes the error for the original sketch $\alpha\binom{\ell}{2}N/w$ whereas our proposed technique has an error of $6\binom{\ell}{2}(1+\ell+\binom{\ell}{2})N/w$. When the size of the alphabet is $\alpha = 72$, the resulting differences in error can be seen in Figure 3: the error when using our proposed scheme is better than the naive method when the length of the word is less than 24 characters.

5 Example Parameter Instantiation

By using the error estimates found in previous sections, it is possible to estimate the size of the table required in order to obtain certain levels of accuracy. To do this, the Markov inequality can be applied to all expected maximum errors. The results are as follows:

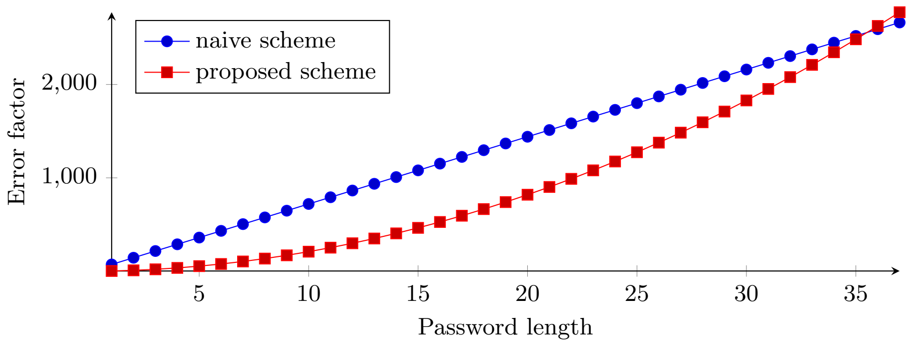

Fig. 2. Error as a multiple of N/w for passwords at distance **1**, using a count-min sketch supporting maximum distance of **1**, compared with naive scheme.

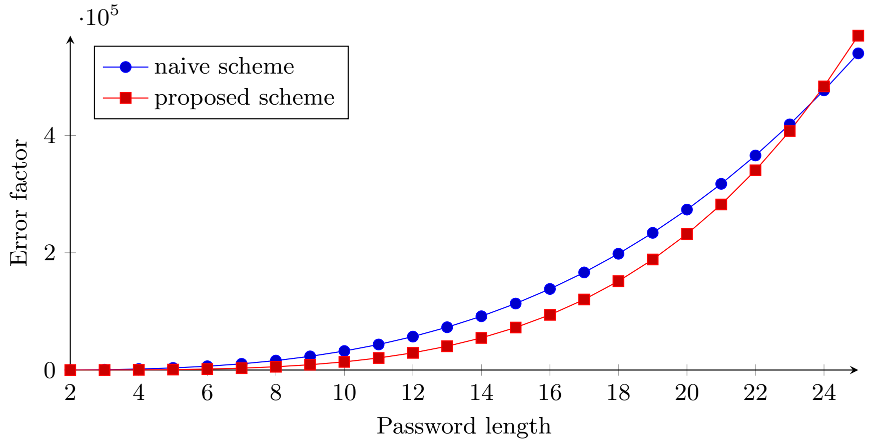

Fig. 3. Error as a multiple of N/w for passwords at distance 2, compared with naive scheme.

- When estimating the number of times a specific word appears, there is an error of at most $2(1+\ell+\binom{\ell}{2})N/w$ with probability of at least $1-(\frac{1}{2})^h$, where h is the number of hash functions.
- There is an error of at most $4\ell(1+\ell+\binom{\ell}{2})N/w$ with probability of at least $1-(\frac{1}{2})^h$ when estimating words within distance 1.
- There is an error of at most $12\binom{\ell}{2}(1+\ell+\binom{\ell}{2})N/w$ with probability of at least $1-(\frac{1}{2})^h$ when estimating words within distance 2.

The following exact parameter calculations are done for distances one and two, but not for Hamming distance zero because, if the user desires a certain error rate for estimating exact words, it is preferable to use the original count-min sketch.

If the purpose of this sketch was to estimate passwords within distance 1 with an error of at most 1% with probability of at least 99.9%, then the width w of the table and the number h of hash functions would have to be as follows:

$$\frac{4\ell(1+\ell+\binom{\ell}{2})}{w} = \frac{1}{100} \implies w = 100\cdot 4\ell\left(1+\ell+\binom{\ell}{2}\right)$$

$$1-\left(\frac{1}{2}\right)^h = \frac{999}{1000} \implies h \approx 6.64$$

The number of hash functions would have to be 7 or more and the width of the table would depend on the length of the password. If the length of the passwords is 6 characters, then the width of the table would have to be at least 48,400. This size would be smaller if the table only included passwords within distance zero to one, as suggested previously.

Similarly, if the sketch was needed to estimate passwords within distance 2 with an error of at most 1% with probability of at least 99.9%, then the number of hash functions would still be at least 7 but the width of the table would be:

$$\frac{12\binom{\ell}{2}(1+\ell+\binom{\ell}{2})}{w} = \frac{1}{100} \implies w = 100\cdot 12\binom{\ell}{2}\left(1+\ell+\binom{\ell}{2}\right)$$

For 6-character passwords, this would make the width 338,800. While this width seems large, it is possible that the false positive (or error) rate could be higher when estimating words within specified distances.

6 Conclusion

When no restrictions are placed on passwords choices, users tend to choose popular passwords. This leaves systems vulnerable to statistical guessing attacks. By limiting the percentage of popular passwords, these attacks are not as successful. In order to do this, efficient tools must be available to track password usage. The count-min sketch can be used to estimate the counts of password usage within a system. However, the count-min sketch is not as effective when estimating the counts of passwords within specified Hamming distances. In this paper, we have proposed a variant of the count-min sketch using wildcard characters that can be used to calculate estimates of words that are close in Hamming distance.

As with the original count-min sketch, there will never be false negatives — where the estimate algorithm under-reports usage of the password — but there may be false positives — meaning the estimate algorithm may over-report usage of the password due to collisions. We have calculated the error rate for estimation as the Hamming distance increases, which allows for calculation of sketch size for a given error rate. For a reasonable alphabet size, the error rates in our proposed method are lower than they would be using the naive approach on a standard count-min sketch for a wide range of password lengths, up to 36- and 24-character passwords when estimating passwords within distances 1 and 2 respective.

Our technique is most suited when all passwords in the database have the same length. In future work, it may be desirable to develop another variation in which *edit distance*, such as Levenshtein distance, is used, rather than Hamming distance, to take into account passwords that are close due to deletions or insertions of characters. Other future work may be to consider the impact of using fractional contributions for passwords within a certain distance, where the fraction is either a function of the Hamming distance divided by the password length, or based on some model of textual similarity: for example, in 'leetspeak', `5` is a common substitution for `s`.

References

1. F. Bergadano, B. Crispo, G. Ruffo. Proactive password checking with decision trees. *Computer and Communications Security*, **4**(1):67-77, 1997.
2. B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, **13**(7):422–426, July 1970.
3. J. Bonneau. The science of guessing: analyzing an anonymized corpus of 70 million passwords. In *Proc. 2012 IEEE Symposium on Security and Privacy (S&P)*, 2012.
4. J. Lawrence Carter and M.N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, **18**(2):143–154, 1979.
5. G. Cormode, S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, **55**(1):58-75, April 2005.
6. G. Cormode, S. Muthukrishnan. Approximating data with the count-min sketch. *IEEE Software*, **29**(1):64–69, January/February 2012.
7. U. Manber, S. Wu. An algorithm for approximate membership checking with application to password security. *Information Processing Letters*, **50**(1):191-197, 1994.
8. J.O. Pliam. On the incomparability of entropy and marginal guesswork in brute-force attacks. In *Proc. INDOCRYPT 2000*, LNCS, vol. 1977, pp. 67–79.
9. S. Schechter, C. Herley, M. Mitzenmacher. Popularity is everything: A new approach to protecting passwords from statistical guessing attacks. In *Proc. 5th USENIX Conference on Hot Topics in Security (HotSec)*, 2010.
10. E. Spafford. Preventing weak password choices. *Purdue University Computer Science Technical Reports*, paper 875, report number 91-028, 1991. `http://docs.lib.purdue.edu/cstech/875`

A Example Count-Min Sketch Calculation

Fix the table T to be of width $w = 101$ and height $h = 2$.

A.1 Hash

First we show how the hash values can be calculated for a single password, say `abcd`, under two hash functions.

We encode each character as two-digit integer, say $\texttt{abcd} \mapsto x = \langle 1, 2, 3, 4 \rangle$, then find the integer representation, $X = 01020304$.

Set a common prime $p = 3571$, and for each of the two hash functions hash_k, choose the parameters a_k and b_k at random modulo p; for example, $a = [1151, 2111]$ and $b = [941, 1433]$.

The hashes of `abcd` are as follows:

$$\mathsf{hash}_1(X) = ((1151 \cdot 01020304 + 2111) \mod 3571) \mod 101 = 20$$
$$\mathsf{hash}_2(X) = ((941 \cdot 01020304 + 1433) \mod 3571) \mod 101 = 83 \;.$$

The vector-wise hash is thus $\mathsf{hash}(\texttt{abcd}, a, b) = \langle 20, 83 \rangle$.

A.2 Update

We now show how to update the table T to record the use of a password, first updating just the entries for the password itself, then the entries for passwords within Hamming distance 2.

Suppose the table T is currently as follows:

column		19	20	21		37	38	39		82	83	84	
row 1	⋯	0	0	0	⋯	0	0	0	⋯	0	0	0	⋯
row 2	⋯	0	0	0	⋯	0	0	0	⋯	0	0	0	⋯

Update at distance 0. Now, suppose we call $\mathsf{update}(T, x = \texttt{abcd}, a, b, c = 1) \mapsto T'$, which is meant to increment (since $c = 1$) the use of the password `abcd`. Assume a and b are as in the previous subsection. Then we have that $\mathsf{hash}(\texttt{abcd}) = \langle 20, 83 \rangle$. Thus, we increment the 20th entry of row 1 and the 83rd entry of row 2, to obtain T':

column		19	20	21		37	38	39		82	83	84	
row 1	⋯	0	**1**	0	⋯	0	0	0	⋯	0	0	0	⋯
row 2	⋯	0	0	0	⋯	0	0	0	⋯	0	**1**	0	⋯

Update at distance 1. Next, we increment all four wildcard passwords within distance 1 of `abcd`, namely `*bcd`, `a*cd`, `ab*d`, and `abc*`. This is done by computing the corresponding hashes

$$\mathsf{hash}(\texttt{*bcd}) = \langle 38, 17\rangle \qquad \mathsf{hash}(\texttt{a*cd}) = \langle 37, 47\rangle$$
$$\mathsf{hash}(\texttt{ab*d}) = \langle 37, 63\rangle \qquad \mathsf{hash}(\texttt{abc*}) = \langle 78, 1\rangle$$

and updating all the corresponding entries of the table accordingly. Notice that a few partial collisions occur, for example, both `ab*d` and `a*cd` collide under hash_1, but fortunately do not collide under hash_2.

In our table extract, we only see a few of the 8 updates since not all columns are shown (though we imagine all the updates are applied):

column		19	20	21		37	38	39		82	83	84	
row 1	···	0	1	0	···	**2**	**1**	0	···	0	0	0	···
row 2	···	0	0	0	···	0	0	0	···	0	1	0	···

Update at distance 2. Finally, we increment all $\binom{4}{2} = 6$ wildcard passwords within distance 2 of `abcd`; namely

$$\mathsf{hash}(\texttt{**cd}) = \langle 55, 82\rangle \qquad \mathsf{hash}(\texttt{*b*d}) = \langle 55, 33\rangle$$
$$\mathsf{hash}(\texttt{*bc*}) = \langle 31, 36\rangle \qquad \mathsf{hash}(\texttt{a**d}) = \langle 18, 27\rangle$$
$$\mathsf{hash}(\texttt{a*c*}) = \langle 95, 66\rangle \qquad \mathsf{hash}(\texttt{ab**}) = \langle 95, 82\rangle$$

In our table extract, we again only see a few of the 12 updates:

column		19	20	21		37	38	39		82	83	84	
row 1	···	0	1	0	···	2	1	0	···	0	0	0	···
row 2	···	0	0	0	···	0	0	0	···	**2**	1	0	···

A.3 Estimate

Suppose we now $\mathsf{estimate}(T, x = \texttt{bbcd}, a, b)$ to obtain the estimate $\overline{\mathsf{est}}^1$ of the frequency of passwords at distance 1 of `bbcd`.

Estimate at distance 0. First, we need to compute an estimate $\overline{\mathsf{est}}^0$ for the number of times `bbcd` itself has been used. We do this by computing $\mathsf{hash}_k(\texttt{bbcd})$ and retrieving the corresponding cell from row i, then taking the minimum. In this case, $\mathsf{hash}(\texttt{bbcd}) = \langle 76, 53\rangle$. The 76th entry in the first row and the 53rd entry in the second row are both 0, so this yields a (correct) estimate that the exact password `bbcd` has been seen 0 times before.

Estimate at distance 1. To compute an estimate $\overline{\mathsf{est}}^1$ for the number of times any password at distance 1 of `bbcd` has been used, we first need to compute estimates for the number of times each of the four wildcard passwords `*bcd`, `b*cd`, `bb*d`, and `bbc*` has been used before.

$$\begin{aligned}
\mathsf{hash}(\texttt{*bcd}) &= \langle 38, 17\rangle && \Longrightarrow \mathsf{est}^1_1 = \min\{1,1\} = 1\\
\mathsf{hash}(\texttt{b*cd}) &= \langle 100, 8\rangle && \Longrightarrow \mathsf{est}^1_2 = \min\{0,0\} = 0\\
\mathsf{hash}(\texttt{bb*d}) &= \langle 100, 60\rangle && \Longrightarrow \mathsf{est}^1_3 = \min\{0,0\} = 0\\
\mathsf{hash}(\texttt{bbc*}) &= \langle 76, 63\rangle && \Longrightarrow \mathsf{est}^1_4 = \min\{0,1\} = 0
\end{aligned}$$

Then, we sum these values and subtract the estimate for the number of times the string `bbcd` itself was used:

$$\overline{\mathsf{est}}^1 = \left(\sum_i \mathsf{est}^1_i\right) - \ell\,\overline{\mathsf{est}}^0 = 1 \ .$$