Popularity *is* everything

A new approach to protecting passwords from statistical-guessing attacks

Stuart Schechter
Microsoft Research

Cormac Herley
Microsoft Research

Michael Mitzenmacher
Harvard University

ABSTRACT

We propose to strengthen user-selected passwords against statistical-guessing attacks by allowing users of Internet-scale systems to choose any password they want—so long as it's not already too popular with other users. We create an oracle to identify undesirably popular passwords using an existing data structure known as a *count-min sketch*, which we populate with existing users' passwords and update with each new user password. Unlike most applications of probabilistic data structures, which seek to achieve only a *maximum* acceptable rate false-positives, we set a *minimum* acceptable false-positive rate to confound attackers who might query the oracle or even obtain a copy of it.

1. INTRODUCTION

User-selected passwords are subject to statistical guessing attacks, a form of dictionary attack in which an attacker sorts the password dictionary by presumed, or previously-observed, popularity and guesses the most popular passwords first. To defend against a statistical guessing attack it is important to

1. limit the number of guesses that the attacker can issue against each account and

2. minimize the cumulative fraction of accounts that use the most popular passwords.

The latter requires influencing user behavior and is the motivation behind myriad password-selection policies, password-strength meters, and user guidance on increasing password obscurity. Existing tools to influence user selection of passwords take a circuitous path to this goal. Password-composition policies are criticized for their usability [2, 7] and have significant unintended consequences. Faced with a requirement to choose longer passwords, users may be more likely to rely on dictionary words or other lower entropy strings [16].

When administrators respond by requiring the use of a special character to increase entropy users may simply take dictionary words and replace letters with a predictable special character ('password'→'$p@ssword$'), achieving little added entropy. If this is forbidden, users may instead map special characters to a relatively small set of concepts: '#'→{lb,hash}, '*'→{star,all}, '^'→{hat,top,up}, and so on. In some cases, this may even reduce entropy. Password policies have become a cat and mouse game where administrators impose stricter and stricter requirements and users, trying to achieve memorability, often confound their efforts by using all-too-similar strategies.

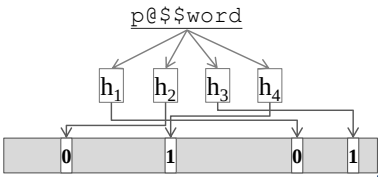
Password-strength meters provide guidance based on rules similar to those used to construct password policies, but the threat model under which they provide this 'strength' is unclear. Thus, most online strength meters will deem a string of 32 random lowercase letters a 'weak' password, yet Windows Live ID will report that "`@Aaaaaa`" is a strong password and Yahoo will report that "`P@ssword`" is a strong password.

We propose taking the direct approach to discouraging the use of dangerously-popular passwords: notifying users when the passwords they've chosen, or their existing password, is already undesirably popular among a large, existing, user base. We use a probabilistic data structure, the *count-min sketch* [13] (related to a Bloom filter), to efficiently track password popularity within the user base. We select a sketch large enough to limit the maximum frequency of false positives, as these force users to unnecessarily choose a new password. We also find it useful to ensure a *minimum* rate of false positives to confound attackers who might query the data structure or even obtain a copy of it.

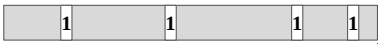
Replacing password creation rules with popularity limits has the potential to increase both security and usability. Since no passwords are allowed to become too common, attackers are deprived of the popular passwords they require to compromise a significant fraction of accounts using online guessing. We conjecture that usability also increases. System designers no longer need to create increasingly complex password-selection rules with no guarantee that they will result in truly strong passwords. Users needn't read, learn, or interpret these rules. Instead, users are only inconvenienced when their password choice is one that would

add(s) for($i \in [1,k]$)
$t\,[h_i(s)] \leftarrow 1$

query(s) return $\bigwedge_{i=1}^{k} t\,[h_i(s)]$



(a) Before add(`p@$$word`), isMember(`p@ssword`) is *false*



(b) After add(`p@$$word`), isMember(`p@ssword`) is *true*

Figure 1: A **Bloom filter** representing a set of passwords. A set of k hash functions map the string to k pseudorandom positions within a table t. Here $k = 4$. To add a string to the set, write a 1 into the table at these positions. To test for membership, verify that the values at *every* one of these k positions is set to 1.

lead to a quantifiably unacceptable level of vulnerability to a statistical guessing attack. Users who choose common passwords discover that password-construction techniques they think would yield unique passwords are actually quite common.

Our proposal is not without precedent. Twitter, in responding to an online password guessing attack that exploited their failure to lock out guessers [1], now forbids 390 of the most common passwords. It would appear that Twitter decided that this inconveniences their users less than the introduction of cumbersome password policies.

2. BUILDING A POPULARITY ORACLE

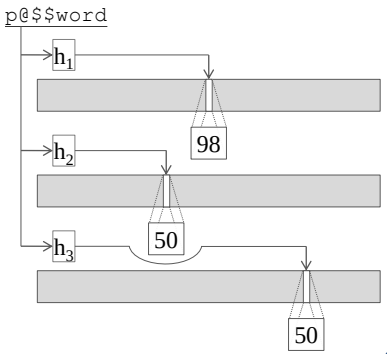
We propose building an oracle that, when queried with a password, replies whether the password is too popular. This oracle leverages statistics on existing passwords, such as those available to the Internet-scale authentication systems used by AOL, Facebook, Google, Microsoft, and Yahoo, each of which have hundreds of millions of user accounts.

One could build such an oracle as a table mapping hashed passwords to occurrence counts, similar to a hashed password file. Unlike hashed password files, the collection of aggregate statistics would require abandoning the use of entry-specific salts before hashing. If the table were revealed, a dictionary attack against all entries would yield the exact frequency of each broken password.

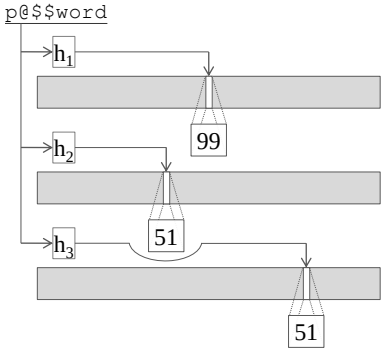
Bloom filters provide an attractive alternative to lists of hashed passwords because the hashes used are small and not unique to any one password. A Bloom filter stores a representation of a set of strings by using a

count(s) return $\min\left(t_1\,[h_1(s)], t_2\,[h_2(s)], \ldots t_k\,[h_k(s)]\right)$

add(s) for $i \in [1,k]$
$t_i\,[h_i(s)] \leftarrow t_i\,[h_i(s)] + 1$



(a) Before add(`p@ssword`), count(`p@$$word`) = 50



(b) After add(`p@ssword`), count(`p@$$word`) = 51

Figure 2: A **count-min sketch** representing the number of times each password occurs in a password database. A set of k hash functions map the string to k pseudorandom positions in k tables. To record an occurrence, increment the counter at each position. To count the estimated occurrences of a string s, find the minimum value stored in the table at these positions. (If the count-min sketch in this illustration had implemented a conservative add operation, the counter associated with h_1 would not have been incremented.)

table t of bits (all initially 0) and k hash functions. A string is mapped to k pseudorandom bit positions in the table via the hash functions. To add a string to the set represented by the Bloom filter, a 1 is written at each of these positions, as illustrated in Figure 1. Membership in the set is tested by verifying that the bits at each of these positions are all set to 1, as is true in Figure 1b but not Figure 1a. False positives can occur if a string is not in the set but the bits in all of the string's k positions are set to 1 by strings that are in the set.

In 1991, Spafford proposed using a Bloom filter [4] to compactly store a collection of dictionary words that should be forbidden for use as passwords [9]. Bloom filters achieve compactness by allowing false positives—

they only approximate the true dictionary. Spafford was motivated to accept false positives because, two decades ago, storing a large dictionary of potential passwords was costly. The binary nature of the Bloom filter employed by Spafford only allowed tests for set membership; it could not be used for any measure of popularity beyond whether a password had been seen before.

To create our popularity oracle we employ a descendant of the Bloom filter, called a *count-min sketch* [13][1] which uses k tables of numeric counters rather than one table of bits. Each of the k hash functions maps strings to a separate table, as illustrated in Figure 2. The `add`(s) operation increments one counter within each table; the position of the counter within table i is specified by $h_i(s)$, the hash function for that table. The `count`(s) operation returns the minimum value of all the counters associated with the string s (one from each table), which is an upper bound on the number of times s has been added to the sketch. Note that it is possible to remove one or more instances of s from a count-min sketch by decrementing the corresponding counters.

Like the Bloom filter, the count-min sketch saves space at the cost of accuracy: count(s) may exceed the actual number of occurrences of string s, as each of the counters associated with s might be incremented by strings other than s. The likelihood of large errors in the count estimates can be made suitably small by choosing the number of hash functions and the size of the filter appropriately; see [13] for a full analysis. To reduce estimate errors, a count-min sketch may implement a *conservative add*: only increment the counter or counters associated with string s that contain the minimum value returned by count(s). Conservative adds preserve the invariant that if a string is inserted m times, all counters associated with that string will have an occurrence count of at least m [5]. However, if conservative adds are used then strings can no longer be removed by decrementing counters.

To quickly populate the oracle with password data, each existing account is associated with a Boolean field (initially *false*) which indicates whether the account's password has been added to the sketch. If the field is *false* when a user logs in, the password she used is added to the sketch and the field is set to *true*.

We consider a password dangerously popular if it occurs at a rate (frequency) that exceeds a threshold r—the *fractional popularity threshold*. After observing N password instances (calling `add`(s) N times) these undesirably popular passwords will have been observed at least $d = rN$ times—the *integer popularity threshold*. This threshold grows with the number of observations.

[1]See also the equivalent *parallel multistage filter* of [5] or the closely related spectral Bloom filter of [20].

For example, assume we want to flag passwords that occur with probability greater than $r = \frac{1}{1{,}000{,}000}$ (the fractional popularity threshold) and have observed N= 100,000,000 password instances. The integer popularity threshold d is equal to $\frac{100{,}000{,}000}{1{,}000{,}000} = 100$, and will grow as more password instances are observed.

Once enough passwords have been collected to bootstrap the oracle with sufficiently reliable data, we flag existing accounts with undesirably popular passwords. We may ask or require users of these accounts to choose new passwords when they next login.

Forbidding passwords with fractional popularity greater than r would limit an attacker who was able to issue G guesses against each account, and who knew the G most popular passwords, to compromise a fraction of at most rG of accounts. For example, if no password had been allowed with popularity greater than $\frac{1}{1{,}000{,}000}$, an attacker using the most popular password can compromise only 0.0001% accounts. In contrast, an attacker who can identify and exploit a target's *single* most-popular password could compromise 0.22% of MySpace accounts [22] or 0.9% RockYou accounts [14].

The downside of using an approximate count-min sketch, as opposed to an exact oracle, is that approximations occasionally lead to false positives. That is, a password will be hashed to counters that all lie above the threshold, even though the password is not actually popular. False positives will force users to choose new passwords unnecessarily. However, we will show in the following section that a modest number of false positives is actually a *desirable* feature of the count-min sketch.

3. PREVENTING MISUSE

It should be no surprise that attackers, as well as defenders, can benefit from password popularity oracles. Indeed, the success of statistical guessing attacks will increase if attackers can refine the order of their dictionaries based on actual popularity statistics.

3.1 Threat model

We assume that attackers not only have the ability to query the oracle online, such as by entering a proposed new password, but that they have access to a perfect copy of the count-min sketch.

We also assume the adversary has access to existing password-popularity statistics and can use them to build a *cracking dictionary*, ordered by the observed popularity of each password. These data sets include compromised passwords that have been semi-publicly released: phished MySpace passwords and 32 million plaintext RockYou passwords. Lists of the most popular passwords from these data sets have been published widely [14, 22]. Attackers can also feed successfully-guessed or compromised passwords from their own attacks back into their statistical popularity models.

For online attacks, we further assume that attackers can leverage millions of IP addresses (such as by using a botnet) to limit the efficacy of IP-based lockout mechanisms. Finally, we assume users have access to millions of valid usernames, or a dense enough space of likely-valid usernames, in order to avoid triggering account-based lockout mechanisms. In other words, they can use only the most popular passwords in order to target millions of different accounts.

3.2 Online attacks

Even if we enforce a no-popular-passwords policy for our own systems, an attacker could still leverage the oracle to identify the most popular user-selected passwords in order to target others' systems. Many sites will continue to rely on rules-based password-selection policies even if we were to allow them to query the oracle. One way to avoid collecting statistics about the most popular user-selected passwords is add passwords to the oracle only when they meet our policy; once a password reaches the popularity threshold it will no longer be added. However, if we were to stop counting once the password was deemed popular, there would be brief periods when all passwords would temporarily fall below the popularity threshold; the integer threshold grows with the number of observations and when it is incremented even the highest `count` values would fall one below it.

Better still is to stop incrementing the count-min sketch shortly above the popularity counting threshold. This *counting limit* must exceed the integer popularity threshold by at least a small margin so that counters associated with popularity don't fall below the threshold because of random variations in the rate at which users select them.

By restricting the `add` function so that no table entry is incremented beyond the counting limit, all passwords that reach this limit will appear equally popular. As attackers already have approximate password popularity estimates from existing data sets, and the more popular passwords will likely exceed the limit, the adversary does not benefit from having a copy of the sketch.

3.3 Offline attacks

An attacker who compromises a password file, and uses a cracking dictionary to perform an offline dictionary attack, can leverage a popularity oracle to identify the most popular entries in the dictionary. The attacker can then skip (or delay using) passwords that the oracle reports as unpopular, speeding up the rate at which passwords are cracked.

Against this threat comes help from an unlikely hero—the very false positives that are the bane of most applications of count-min sketches, Bloom filters, and other probabilistic data structures. While most applications grow probabilistic data structures to be *large* enough to bound the *maximum false positive rate*, we also require that the count-min sketch be *small* enough to guarantee a *minimum false positive rate*.

Assume, for example, that the count-min sketch has a false positive rate of 1%, so that 1% of all possible password strings yield a false positive. User-selected passwords are more likely to be popular than random ones. If only 10% of user-selected passwords are popular, fewer than 10% of the warnings issued to users that their passwords are too popular will be false. This seems reasonable given that impact of a false positive is that the user will need to select another password.

On the other hand, consider an attacker with a cracking dictionary sorted with the best popularity information available before obtaining access to the oracle. The attacker uses the oracle to ensure that unpopular passwords are only guessed after the popular ones have been exhausted. For every 99 unpopular password strings the attacker may filter out, one will appear to be popular and remain. The maximum-possible improvement in cracking performance is therefore a factor of 100. While this is non-negligible, it is smaller than the gain achievable through hardware optimization or through parallelization (*e.g.* using botnets). It does not change the recommended action in the event password database exposure: all user passwords should be reset.

Attackers who try to grow their cracking dictionaries by testing random passwords against the oracle will receive orders of magnitude more false positives than truly popular passwords.

While others have leveraged false positives inherent in Bloom filters to protect the privacy of set membership [3, 17], we are not aware of prior work that seeks to guarantee a minimum false-positive rate for a counting filter, such as a count-min sketch. It is thus not surprising that theoretical work has not considered how to size such data structures to guarantee a minimum false positive rate. We can approximate the correct sizing by building sketches using distributions (*e.g.* Zipf) that approximate the popularity of passwords. We can then choose a filter size that errs on the small side, knowing that if the false-positive rate is too high we can extend the sketch with an additional table and start again.

3.4 Intersection attacks

Attackers can overcome false positives by testing passwords against two or more popularity oracles. If false positives are independent, the likelihood that an unpopular password would be reported popular by each successive oracle would decrease rapidly as a product of their false positive rates.

Since the number of organizations that can leverage hundreds of millions of passwords to build a popularity oracle is small, they may be able to share a single oracle.

Updates could be issued in bulk to protect the privacy of individual users across organizations.

Even if there exists only one oracle (one count-min sketch), there is still the danger that it would become necessary to create a new sketch in the future. For example, substantial growth in the number of total and unique passwords observed could cause an unacceptable increase in the false positive rate, causing too many users asked to choose new passwords unnecessarily.

To reduce the false positive rate by some factor b, we simply create a new count-min sketch with a size chosen to make b the false positive rate. We can populate this sketch in a matter of days by entering existing passwords (once per account) at the time users log in. This new sketch would have its own integer popularity threshold and counting limit based on the number of passwords it has observed (the number of `add` operations). Once the new sketch is populated, a password will now be deemed popular only if both sketches would have deemed it popular. In other words, the set of popular passwords is the intersection of the sets represented by the count-min sketches.

The offline adversary benefits from the fact that the false positive rate has been reduced by a factor of b. However, since the set of passwords now deemed popular is a strict subset of those deemed popular by the original set, he learns nothing by taking the intersection.

4. ATTACK-DETECTION SKETCHES

A successful defense against statistical guessing attacks requires not only the avoidance of popular passwords, but also mechanisms to limit the number of guesses an attacker can issue. Limiting guesses is especially important if users aren't forced to avoid popular passwords, or if users with accounts that predate the no-popular-passwords policy have not been forced select less popular passwords. To protect accounts, a per-IP or per-account limit of guesses may be imposed, above which the system may reduce the rate at which guesses can be made (*e.g.* exponential back-off), require a CAPTCHA, or even locked the user out. Lower limits provide for better defense, but decreased usability.

In prior work on personal authentication (security) questions, another form of knowledge-based authentication that, like passwords, is subject to statistical guessing attacks, Schechter *et al.* observed that users who require multiple responses to get the correct answer are unlikely to guess only popular answers. They proposed varying the permitted number of guesses inversely with the popularity of values guessed [21]. We could use the oracle to gauge popularity for this purpose, but we had expressly designed it to avoid tracking popularity above the popularity threshold. Rather than risk extending the oracle in a manner that might aid attackers, we introduce a new *attack-detection* count-min sketch.

The attack-detection sketch stores only those passwords used in *unsuccessful* login attempts, and does not store actual account passwords. Attack-detection sketches may be short-lived and may be created only at times when an unusually high incidence of failed login attempts indicates an attack is likely underway. Should attackers choose a small set of popular passwords, these passwords will occur with high frequency in the sketch. We can thus reduce the number of login attempts permitted in proportion to the popularity of the incorrect passwords guessed. Attackers can respond by growing the set of passwords they use in their attacks, but this will require them to include ones that are progressively less popular, reducing the efficacy of each guess.

In the extreme case in which an attacker uses a small set of passwords he suspects to be the most popular (*e.g.* 5), we might even trigger a CAPTCHA before the *first* login attempt that uses this password can be checked against the password database. Legitimate users have chosen these passwords would now face a CAPTCHA on every login during the attack. We consider this a feature; if a user has chosen a password that is among a small set being targeted by attackers, having to face a CAPTCHA on every login provides an extra incentive to change it.

5. RELATED WORK

Checking to prevent unacceptable password choices has a long history. Klein [6] suggests "a publicly available proactive password checker, which will enable users ... to check *a priori* whether the new password is safe." He proposes a combination of a dictionary check and other rules to screen for common weak password choices. Spafford [9] describes OPUS, a Bloom filter to compress a dictionary of forbidden passwords. OPUS stores a $250,000$ word dictionary in 350 kBytes with a 0.5% false positive rate. The OPUS system also supported password aging: password choices are added to the filter, so attempts to re-use old passwords will be refused. A related work uses the OPUS system to gather information on actual user password choices without the risk of leak [8]. Manber and Wu [23] describe an approach based on Bloom filters that allows checking both exact and approximate dictionary membership. Passwords that are a single insertion, deletion, or substitution from a dictionary word will be be refused. Bergadano *et al.* [12] describe a decision tree approach which achieves greater dictionary compression. However, the system does not allow incremental additions of new dictionary words. Instead, retraining must be performed when additions are made.

Despite persistent and creative efforts to nudge users toward better practices [11], password strength remains a problem [10]. Yan [15] points out that any mismatch

between the dictionary used by the checker and the attacker can result in weak passwords beng accepted. He suggests augmenting the dictionary checks used in proactive password checking with entropy checks as well. Pinkas and Sander [19] propose the use of CAPTCHA's to slow down dictionary attacks when limiting the number of attempts is undesirable. Van Oorschot and Stubblebine [18] extend this work and show that using login history can greatly reduce the number of CAPTCHA's presented to users (as opposed to attackers).

6. DISCUSSION AND FUTURE WORK

Verifying the hypothesized usability benefits of a no-popular-passwords policy would require a password oracle to be populated with a large database of passwords and the use of this oracle in a large usability study. We have not undertaken such a study; all known lists of popular passwords are stolen goods.

Our proposed approach of building popularity oracles from count-min sketches can be used in other knowledge-based authentication schemes, such as the 'secret' security questions. These questions often have very popular answers that render them vulnerable to statistical guessing attacks [21].

If we can sufficiently limit the benefit to attackers of obtaining the sketch, we could publicly distribute a compressed copy of the sketch. To compress the sketch, we turn each counter into a single bit containing 1 if above the popularity threshold and 0 otherwise. This more compact version could still be queried, but not added to. The compressed sketch could then be embedded in operating systems and devices to help users choose better passwords even when offline.

One open question is how to design an oracle that may be queried online without revealing the queried password, or information from which it could be derived. Another is reduce the risks to integrity when multiple parties to contribute to the oracle, as malicious parties may try to prevent popular passwords from being detected.

7. CONCLUSION

We have proposed replacing today's complex password policies with a simple one: allow any password the user desires, so long as it is not an attractive target for a statistical guessing attack. To enable this policy we describe how use a count-min sketch to build a password-popularity oracle. We confound attackers who would use this oracle to identify the very most popular passwords by ensuring that the information they desire is not tracked by the count-min sketch. We require that the count-min sketch have a *minimum* false-positive rate, limiting the advantage attackers can obtain by filtering cracking dictionaries (lists of potentially-popular passwords) through the oracle.

8. REFERENCES

[1] Wired: Weak Password Brings 'Happiness' to Twitter Hacker. `http://blog.wired.com/27bstroke6/2009/01/professed-twitt.html`.

[2] Adams, A., and Sasse, M. A. Users Are Not the Enemy. *Commun. ACM 42*, 12 (1999).

[3] Bawa, M., Bayardo, R., Agrawal, R., and Vaidya, J. Privacy-preserving indexing of documents on the network. *The VLDB Journal 18*, 4 (2009), 837–856.

[4] B.H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Comm. of the ACM* (1970).

[5] C. Estan and G. Varghese. New directions in traffic measurement and accounting: focusing on the elephants, ignoring the mice . *ACM Transactions on Computer Systems* (2003).

[6] D. V. Klein. Foiling the Cracker: A Survey of, and Improvements to, Password Security. *Proc. of the 2nd USENIX Security Workshop* (1990).

[7] D.A. Norman. The Way I See It: When security gets in the way. *Interactions 16*, 6 (2009), 60–63.

[8] EH Spafford. Observing reusable password choices. *Proceedings of the 3rd UNIX Security Symposium* (1992).

[9] EH Spafford. OPUS: Preventing weak password choices. *Computers & Security* (1992).

[10] Florêncio, D., and Herley, C. A Large-Scale Study of Web Password Habits. *WWW 2007, Banff.*.

[11] Forget, A., Chiasson, S., van Oorschot, P. C., and Biddle, R. Improving text passwords through persuasion. In *SOUPS '08: Proceedings of the 4th symposium on Usable privacy and security* (2008).

[12] G. Bergadano, B. Crispo and G. Ruffo. Proactive Password Checking with Decision Trees. *Proc. CCS* (1997).

[13] G. Cormode and S. Muthukrishnan. An Improved Data Stream Summary: The Count-Min Sketch and Its Applications. *Journal of Algorithms* (2005).

[14] Imperva. Consumer Password Worst Practices. `http://www.imperva.com/docs/WP_Consumer_Password_Worst_Practices.pdf`.

[15] J.J. Yan. A Note on Proactive Password Checking. *NSPW* (2001).

[16] L. St. Clair and L. Johansen and W. Enck and M. Pirretti and P. Traynor and P. McDaniel and T. Jaeger. Password Exhaustion: Predicting the End of Password Usefulness. In *Proc. of 2nd Intl Conf. on Information Systems Security (ICISS)* (2006).

[17] Parekh, J., Wang, K., and Stolfo, S. Privacy-preserving payload-based correlation for accurate malicious traffic detection. In *Proceedings of the 2006 SIGCOMM workshop on Large-scale attack defense* (2006), ACM, p. 106.

[18] P.C. van Oorschot, S. Stubblebine. On Countering Online Dictionary Attacks with Login Histories and Humans-in-the-Loop. *ACM TISSEC vol.9 issue 3* (2006).

[19] Pinkas, B., and Sander, T. Securing Passwords Against Dictionary Attacks. *ACM CCS* (2002).

[20] S. Cohen and Y. Matias. Spectral Bloom Filters . *Proc. ACM SIGMOD Intl Conf. on Management of Data* (2003).

[21] Schechter, S. E., Brush, A. J. B., and Egelman, S. It's No Secret: Measuring the Security and Reliability of Authentication via "Secret" Questions. In *IEEE Symposium on Security and Privacy* (2009), pp. 375–390.

[22] Schneier, B. MySpace Passwords Aren't So Dumb. *Wired, Dec.* (2006). `http://www.wired.com/politics/security/commentary/securitymatters/2006/12/72300`.

[23] U. Manber, S. Wu. An Algorithm for Approximate Membership Checking with Application to Password Security. *Information Processing Letters* (1994).