Mechanism for the prevention of password reuse through Anonymized Hashes

Junade Ali Corresp. 1

1 Cloudflare Inc., San Francisco, California, United States

Corresponding Author: Junade Ali
Email address: junade@cloudflare.com

Password authentication is an essential and widespread form of user authentication on the Internet with no other authentication system matching its dominance. When a password on one website is breached, if reused, the stolen password can be used to gain access to multiple other authenticated websites. Even amongst technically educated users, the security issues surrounding password reuse are not well understood and restrictive password composition rules have been unsuccessful in reducing password reuse. In response, the US NIST have published standards outlining that, when setting passwords, authentication systems should validate that user passwords have not already been compromised or breached. We propose a mechanism to allows for clients to anonymously validate whether or not a password has been identified in a compromised database, without needing to download the entire database or send their password to a third-party service. A mechanism is proposed whereby password hash data is generalized such that it holds the *k*-anonymity property. An implementation is constructed to identify to what extent the data should be generalized for it to hold *k*-anonymity and additionally to group password hashes by their generalized anonymous value. The implementation is run on a database of over 320 million leaked passwords and the results of the anonymization process are considered.

Mechanism for the prevention of password reuse through Anonymized Hashes

Junade Ali[1]

[1]**Cloudflare, Inc., San Francisco, California**

Corresponding author:
Junade Ali[1]

Email address: junade@cloudflare.com

ABSTRACT

Password authentication is an essential and widespread form of user authentication on the Internet with no other authentication system matching its dominance. When a password on one website is breached, if reused, the stolen password can be used to gain access to multiple other authenticated websites. Even amongst technically educated users, the security issues surrounding password reuse are not well understood and restrictive password composition rules have been unsuccessful in reducing password reuse. In response, the US NIST have published standards outlining that, when setting passwords, authentication systems should validate that user passwords have not already been compromised or breached. We propose a mechanism to allows for clients to anonymously validate whether or not a password has been identified in a compromised database, without needing to download the entire database or send their password to a third-party service. A mechanism is proposed whereby password hash data is generalized such that it holds the k-anonymity property. An implementation is constructed to identify to what extent the data should be generalized for it to hold k-anonymity and additionally to group password hashes by their generalized anonymous value. The implementation is run on a database of over 320 million leaked passwords and the results of the anonymization process are considered.

INTRODUCTION

Password reuse poses a significant threat to password authentication systems, Das et al. (2014) observed that 43% of users reused passwords across multiple websites and notes that there have been user password breaches from high profile websites including Twitter, Yahoo and LinkedIn. If a password is reused across multiple accounts, an attacker who is able to compromise the password from one service is able to reuse the same password to compromise other user accounts. A users password can become known if it is insecurely stored in an applications database and the contents of that database are disclosed. Another strategy for disclosing a users password involves attempting a brute force attack by automatically attempting common password combinations until a users password is known, websites which do not challenge or rate limit high-frequency log-in attempts make good targets for this kind of attack. In either instance, a website which insecurely protects a users credentials can become the weakest link in compromising a series of accounts (Herley and Van Oorschot, 2012). In it's worst case, password reuse can lead to a domino effect whereby a low value account being compromised can lead to financial disaster for an individual.

In standards published by the United States National Institute of Standards and Technology (Grassi et al., 2017) it is a requirement, when storing or updating passwords, to ensure they do not contain values which are commonly used, expected or compromised. Grassi et al. (2017) cites passwords obtained through previous security breaches and dictionary words amongst the examples of potential sources for such a blacklist.

Lists of disclosed passwords are available in bulk on the Internet; one such list, Hunt (2017), contains over 320 million such passwords in SHA-1 hashed form. Another such list, Hornby (2016), contains just under 1.5 billion passwords in raw form (without hashing). In uncompressed form such datasets are $12.86GB$ and $15GB$ in size, respectively.

Without downloading the entire substantial datasets of leaked passwords, users have to enter their

passwords into third-party websites in order to validate whether their password has or has not been leaked. If such passwords are sent in their original format, it is possible for a third-party website to capture or log such passwords, thereby violating their secrecy and allowing for malicious use. Submitting passwords for validation in a hashed form still contains a number of security problems. Firstly when a password is successfully found to be in a blacklist, it can be possible to correlate a given password to a user based on characteristics of the request such as IP Address or browser signature (Eckersley, 2010). Even where a hash is not found in a database, such as a hash can be stored for later analysis; the stored hash can later be compromised if a later disclosure results in an identical hash or, if the hashing algorithm itself is found to be vulnerable, the plain-text may be identified.

In this paper, an approach is presented to allow clients to anonymously check whether a password is in a database of breached passwords without needing to download the entire database. Password hashes are pre-processed such that they can be queried in ranges, this allows a client to check if their password is in a database of leaked passwords by only submitting the first few characters of the password hash. An algorithm is proposed to discover the maximum length of the hash prefix that can be searched for such that the query holds the property of k-anonymity. An implementation is built using the Go programming language, with the processed data stored in a SQLite database. By running this implementation on a list of over 320 million breached passwords, we are able to evaluate the practicality of using the resulting data set. We find that the result size of any query is small enough for a large variety of practical use-cases, whilst still maintaining anonymity for the original password hash that is queried.

1 RELATED WORK

1.1 Mechanisms for Deterring Password Reuse

Das et al. (2014) discusses how client-side password hashing can be used to generate unique passwords for different websites, thus helping mitigate the risk of password reuse; this work is similar in spirit, however does not itself help to identify when a password has been compromised through a breach. The work also requires the user to install a browser extension themselves instead of allowing a website or an application developer to themselves take steps to help protect users of password reuse.

Jenkins et al. (2014) discusses mechanisms for preventing password reuse and proposes analyzing changes in typing patterns to detect password reuse. The authors propose they are able to detect password reuse with 81.71% accuracy; upon detecting password reuse a just-in-time fear appeal is used to persuade the user to select a different password. They find that 88.41% of users who received the fear appeal later set unique passwords, whilst only 4.45% of users who did not receive a fear appeal would set a unique password. This work demonstrates that it is advantageous to alert users to password reuse, however the proposed mechanism still has a non-trivial margin of error. Further; when a password is auto-filled using a client-side password manager, typing patterns cannot identify password reuse (as the user does not enter in the password themselves).

Campbell et al. (2011) discusses the impact of imposing restrictive password composition rules on the password choices made by users; websites can impose restrictive policies on how passwords can be formed (such as requiring a minimum number of special characters or utilization of upper case characters), the authors consider whether these enforcement policies make positive impact on users password behavior. By performing user research, the paper considers the implication of password composition rules on the resulting password. The research largely discredits the use of password composition rules, finding they do not make any particular difference to human behavior. With specific regard to password reuse, the research does not support the proposition that password composition rules decrease password reuse.

Bonneau et al. (2012) extensively considers a vast quantity of alternative authentication mechanisms to act as alternatives to passwords; despite a wide-ranging search, the authors fail find an alternative to password authentication on the basis that such mechanisms do not retain the full set of benefits that passwords offer. The authors claim that as a result of failing to consider a sufficiently wide range of real-world constraints, many academic proposals have failed to gain traction. Gaw and Felten (2006) finds a gap between the technology available to limit password reuse and it's utilization. Even amongst a well-connected, well-educated technologically savvy demographic their was still trouble understanding the nature of attacks against reused passwords and the sampled users rarely utilized technology such as password managers to aid in limiting password reuse. Further, Gaw and Felten (2006) demonstrated that password reuse becomes a greater problem as users acquire more accounts accounts online, as more accounts can result in greater password reuse.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3322v1 | CC BY 4.0 Open Access | rec: 5 Oct 2017, publ: 5 Oct 2017

1.2 Hash Functions

Given an arbitrary input x and a hash function $h(input)$, $h(x)$ produces an output c such that it is computationally difficult to find x whilst knowing only the details of the function h and the output c (Naor and Yung, 1989).

The Avalanche Effect is a property of ideal hashing algorithms, whereby a tiny change in the input will result in a substantial change in the cipher text (Yang et al., 2017). This property means that a small change in the input of a hash algorithm can result in a completely different output, further, this property helps ensure that the input of a given hash cannot be evaluated from the hash of a similar input.

Menezes et al. (1996) notes Collision Resistance as another such property of hash algorithms. This property requires that it be computational difficult to find 2 distinct inputs, x and x', such that the resulting hashes have identical outputs; i.e. $h(x) = h(x')$. Note that as hash algorithms have a fixed size of output and an arbitrarily sized input, due to the Pigeonhole principle, it is inevitable that such collisions will occur (Paar and Pelzl, 2010).

Hashing algorithms are deterministic; let $y = h(x)$, no matter how many times $h(x)$ is run, the value of y will be the same each time. Lists of commonly used passwords can be hashed and stored in large databases, such databases (Rainbow Tables) can be used to rapidly find plain-text passwords from hashes (Kumar et al., 2013). In order to mitigate this risk, the plain-text can be concatenated with a random salt prior to hashing in order to make pre-computation more difficult; the hash and the salt are then stored for later use. When the hash needs to be regenerated in order to compare a user input to the original hash, the new hash is generated by concatenating the user-input with the same salt. Password hashing algorithms such as BCrypt (Provos and Mazieres, 1999) or PBKDF2 (Visconti et al., 2015) incorporate mechanisms to salt passwords; for example, PBKDF2 will repeatedly hash a password for a number of iterations and each time the hashing process is repeated, the salt is suffixed to the end of the output of the last round of hashing. These password hashing schemes are used as the additional complexity in deriving the resulting hash means additional computational complexity can be required to break a given hash, dependent on the algorithm used. Whilst such algorithms exist, as password leaks have demonstrated, they are frequently insecurely implemented (Ruoti et al., 2016) or not implemented at all with passwords being stored in plain-text (Bauman et al., 2015).

1.3 Anonymization of Data

Ghinita et al. (2007) discusses the use of k-anonymity and l-diversity as a mechanism for preserving the privacy of an individual's record whilst releasing workable datasets. One such utility of exposing such information is for hospitals to be able to release patient information for medical research. The k-anonymity model states that for every record in a released table there should be $k-1$ other records identical to it. k-anonymity is often achieved by generalization (such as truncating a phone number) or suppression (hiding fields entirely). The concept of l-diversity builds upon this by requiring that each equivalence class has at least l "well-represented" values for the sensitive attribute. Machanavajjhala et al. (2007) provides three possible definitions for how an equivalence-class can be well-represented, the simplest definition (Distinct l-diversity) requires there be at least l distinct values for the sensitive field in each equivalence class. Despite the introduction of l-diversity, k-anonymity remains a useful concept and is suitable in cases where the sensitive attribute is omitted or implicit, Ghinita et al. (2007) cites an example of this being a database containing information of convicted people with the crimes omitted.

Ghinita et al. (2007) is of relevance to this work, as it covers the anonymization of one-dimensional identifiers. Whilst this work uses one-dimensional identifiers as quasi-identifiers (that are derived from multi-dimensional data), instead our use-case purely concerns the anonymization of hashes which are themselves a one-dimensional dataset. Xiao and Tao (2006) proposes an l-anonymization mechanism called Anatomy that works by hashing records into different buckets (based on their Sensitive Attribute) and partitions the data by randomly selecting l records from distinct buckets.

2 IMPLEMENTATION

For our implementation; we seek to provide an API service which allows remote web services and clients to validate whether a password is or is not in a database of leaked passwords, without ever needing to send the password itself. In order to achieve this anonymity, a Range Query can be used such that the client will request all passwords in a particular search range. This Range Query will return multiple matching results, from here the client is able to determine if one of the passwords that is returned is identical to the

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3322v1 | CC BY 4.0 Open Access | rec: 5 Oct 2017, publ: 5 Oct 2017

password supplied by the user. An interceptor or the operator of the API service cannot themselves tell if the client queried for a password that was breached, as the only query was for a range of data (that is associated to multiple possible entries).

2.1 Mitigating Background Knowledge Attacks

In anonymizing text data, it is possible to infer attributes about the contents based on the search query. For example; a search request is put to the API service for all leaked passwords starting with "8StJoh" and it is known the requester resides at the address "8 St Johns Road", it is possible to infer the password even if it is not in the database of leaked passwords itself. This is a known restriction in k-anonymity and has been discussed extensively in Machanavajjhala et al. (2007) where l-diversity is proposed in order to help mitigate such Background Knowledge Attacks. As such, it is important to ensure the ranges of passwords queried do not become quasi-identifier attributes themselves.

As a result of the Avalanche Effect, when highly-similar plain-text values are hashed they can result in completely different hash values. This has the benefit of ensuring that when searching for a particular range of hashes, the results do not necessarily have anything which connects the values together. For example; should we query all hashes which start with the hexadecimal value $0FF56$, the resulting hashes do not necessarily have a common meaning in plain-text. As such, we can mitigate a variety of Background Knowledge Attacks by simply processing every plain-text password in the database through a hash algorithm which is known to maintain the Avalanche Effect property (Yang et al., 2015).

Note that unlike when passwords are normally hashed, the aim here is not to ensure that the hashed passwords cannot be restored to their plain-text. These values can be hashed without the need for salting or using password derivation functions like BCrypt (Provos and Mazieres, 1999). The plain-text values of these passwords is already known as they have been previously leaked and uniquely salting the passwords would make it impractical for them to be searched. The purpose of hashing the passwords here is to prevent the plain-text used in the Range Query from becoming a quasi-identifier attribute. It is to ensure that a particular Range Query of passwords doesn't correlate to any underlying meaning in the plain-text.

2.2 Distributing Hashes into Buckets

For hashes to be able to be easily queried by third-party services, they are split into buckets which consist of a set of hashes. Let $S = \{h_1, h_2, h_3, \ldots, h_S\}$ where S represents a set of password hashes and let l represent the minimum amount of hashes in an individual bucket. Our aim is to find $A = \{b_1, b_2, b_3, \ldots, b_n\}$ such that $b_n \subset S$ whilst $\sum_{i=1}^{|A|} b_i = |S|$ and $|b_n| \geq l$.

Friedman et al. (2006) discusses how a decision tree can be used for searching a dataset whilst maintaining k-anonymity. In this instance, as we are anonymizing one-dimensional data, it is possible to implement a tree structure to allow third-parties to search for password hashes so long as any search query will return a multiple hashes.

Take the example of a crude theoretical hashing algorithm that produces it's output as an integer between 1 and 30, with the breached hash values stored in a decision tree of a similar structure to Fig. 1. The service is exposed via an API service; when a consumer wishes to run a query to check whether a given hash is in the database, they will first submit an API call an endpoint which will return the range of hashes available in the database and will state which questions the client needs to answer in order to continue the search. This decision tree will continue until the query achieves it's desired level of accuracy, at which point the complete set of hashes matching that query is returned.

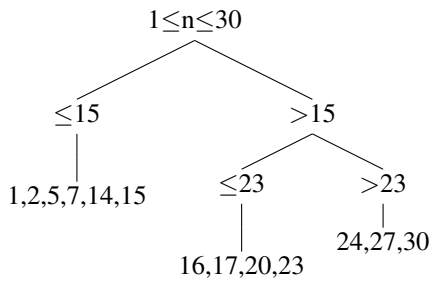

Figure 1. Example Decision Tree

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3322v1 | CC BY 4.0 Open Access | rec: 5 Oct 2017, publ: 5 Oct 2017

As the number of hashes stored in the decision tree increases, in order reduce the amount of values returned by the leaf nodes, the depth of the decision tree must also increase. B-Tree is a simple example of a decision tree; with n items, the longest path of a B-Tree of order d is at most $\log_d n$ nodes (Comer, 1979). Whilst $logn$ difficulty can be considered small for many applications, it is important to note that at each decision point an API call will need to be made. Where Internet connections have high latency (such as in developing countries or for those over mobile connections) multiple API requests can be incredibly slow, especially when performed in a web browser using JavaScript, and therefore this approach was not suitable.

A similar approach could be to provide a map file which flattens a tree structure into a single index file, an example of the data tree Fig. 1 represented in a flat file format is demonstrated in Fig. 2.

1.txt: $1 \leq n \leq 15$
2.txt: $15 \leq n \leq 23$
3.txt: $23 \leq n \leq 30$

Figure 2. Example Flattened Tree

As a greater number of hashes are added, instead of the tree depth increasing, the size of the index file would increase. In other words, we'd simply be trading a large amount of API requests for a single, but significantly larger, index file. Decreasing the index file would require an increase in the order of the tree itself (thereby increasing the amount of hashes stored in an individual leaf). As a result this solution also seemed impractical.

By allowing the client to itself perform the generalization of a password hash, we would eliminate the need for there to be a tree entirely. Instead the client would not need to follow a decision tree and instead would itself know how a given hash should be generalized for anonymity. For this to occur, the client would apply a generalization to the user-supplied password and request all hashes which matched the same generalization.

One mechanism of doing this would be through performing arithmetic operations on the hashes. Firstly the database is pre-processed by calculating the minimum possible numeric value of a hash in the database alongside the maximum numeric value. For the final step in the pre-processing, we calculate the maximum distance between l values in the entirety of the dataset and this becomes our $step$ value. This data is presented in an index file for the API, an example structure is shown in Fig. 3. In order to determine which hashes are then split into which files, a simple formula is used: $file = \lfloor (largestHash - smallestHash)/step \rfloor + 1$.

Start: 5
End: 25
Step: 5

Figure 3. Example Index File

When a client wishes to check if their hash is in a given database, they can generalize their hash by re-running the formula locally, with the appropriate values obtained from the APIs index file. They then perform another API request to obtain all password hashes listed in the same file as the generalization and can locally validate if any of these hashes match the user provided password. Whilst this approach seems promising, there is still a significant drawback. Hashes typically are large numbers, outside the maximum integer value of a variety of programming languages and as such, developers would need to implement Big Integer libraries to perform such calculations.

A simpler alternative to anonymize hash data is to instead truncate the hashes such that the a query is made to an API service on the basis of prefix of a hash. This idea is not completely novel, this concept is briefly discussed in Lu and Tsudik (2010) and Manweiler et al. (2009). Lu and Tsudik (2010) proposes an alternative Domain Name System for the Internet which utilizes Distributed Hash Tables. In essence, clients retrieve all values whose hash value matches a Hash Prefix, thereby achieving

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3322v1 | CC BY 4.0 Open Access | rec: 5 Oct 2017, publ: 5 Oct 2017

anonymity. Manweiler et al. (2009) also discusses the utilization of Hash Prefixes to preserve k-anonymity when sending messages in location-based social networks. Neither of these concepts consider to the anonymization of password hashes and therefore the work in determining the prefix length is not applicable to this use-case.

This approach can be implemented in a relatively simple way. Suppose a user enters the password $test$ into a login form and the service they're logging into is programmed to validate whether their password is in a third-party database of leaked password hashes. Firstly the client will generate a hash (in our example using SHA-1) of $a94a8fe5ccb19ba61c4c0873d391e987982fbbd3$. The client will then truncate the hash to a predetermined number of characters (for example, 5) resulting in a hash prefix of $a94a8$. This hash prefix is then used to query the remote database for all hashes starting with that prefix (for example, by making a HTTP request to $example.com/a94a8.txt$). The entire hash list is then downloaded and each downloaded hash is then compared to see if any match the locally generated hash. If so, the password is known to have been leaked.

Determining Hash Prefix Length

A critical decision to make, prior to anonymizing data, is the length of the Hash Prefix that is used for clients to be able to search the database. For a range of password hashes to hold the property of l-diversity, there must be at least l distinct password hashes which start with the same Hash Prefix. When every Hash Prefix in the entire password database holds the property of l-diversity, the entire database can be said to have l-diversity (Li et al., 2007).

Algorithm 1 demonstrates a simple algorithm to find a Maximum Hash Prefix Length that ensures every Hash Prefix represents at least more than one hash from the entire dataset. This algorithm uses a method defined in Algorithm 2 for determining the similar prefix between two individual hashes. The first step that Algorithm 1 performs is order all hashes in alphabetical order, this allows for rapid checking to find the similarity in the prefix of two hashes as the program only needs to look at two other values instead of searching the entire text to locate another hash with the same prefix. This property helps save

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3322v1 | CC BY 4.0 Open Access | rec: 5 Oct 2017, publ: 5 Oct 2017

memory utilization when dealing with large sets of hashes.

```
Data: passwordHashes
Result: prefixLength
S = {};
prefixLength = 40;
// Default prefix length set to SHA-1 hash length
passwordHashes = sortAlphabetically(passwordHashes);
while passwordHashes as hash do
    S_n = hash;
    if |S| == 3 then
        A = similarPrefix(S_1, S_2);
        B = similarPrefix(S_2, S_3);
        similarity = max(A, B);
        if similarity < prefixLength then
            prefixLength = similarity;
        end
        delete(S_1);
    end
end
```

Algorithm 1: Calculate Truncation to Anonymize Hashes

```
Function SimilarPrefix(a, b):
    for length = 0; length ≤ length(a); length+1 do
        aPrefix = truncate(a, length);
        bPrefix = truncate(b, length);
        if aPrefix == bPrefix then
            similarity = length;
            continue;
        end
        break;
    end
    return similarity;
return
```

Algorithm 2: Similar Prefix Method

Distributing Hashes into Buckets by Prefix

With the Hash Prefix length determined, we use Algorithm 3 to group every password hash together by their respective Hash Prefix. This algorithm takes input with a set of password hashes and the desired truncation length that's required.

```
Data: passwordHashes, truncationLength
Result: results
results = {};
while passwordHashes as hash do
    n = truncate(hash, truncationLength);
    // Hash Prefix is stored in n
    results_n.insert(hash);
    // Hash Prefix (n) stored in results set, with individual hashes
       stored as a subset (in result_n)
end
```

Algorithm 3: Generate Anonymized Hash Information

In Algorithm 3, we create a set *results* which contains Hash Prefix values. Each of these Hash Prefix values act as sets which then contain a subset consisting of the hashes associated to the relevant Hash Prefix.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3322v1 | CC BY 4.0 Open Access | rec: 5 Oct 2017, publ: 5 Oct 2017

Open Questions

We have covered a theoretical approach for performing a Range Query by using a Hash Prefix; however, there still remain a number of open questions. Consider the distribution of the hashes, if password hashes are overwhelmingly more likely to start with one particular Hash Prefix over others, this leaves potential for a number of problems. A Range Query which returns a small amount of hashes for for one prefix may instead return a vast number of hashes for another Hash Prefix. If this is the case, we would have to consider an implementation approach which also utilizes an alternative search mechanism for further filtering (such as a Decision Tree). In order to determine whether distribution is a problem, we can evaluate the implementation on a trial dataset.

EVALUATION

To evaluate the performance of Algorithms 1 and 3, we build an implementation that processes Hunt (2017), a list of over 320 million leaked passwords. The passwords already come hashed using the SHA-1 algorithm, saving a step.

With the hashed passwords collected in a simple text file, the hashes are pre-processed to ensure all the values are unique and additionally to sort the hashes into ascending alphabetical order such that Algorithm 1 can work effectively. The output of this pre-processing is passed into a text file which contains the $320,335,236$ hashes.

An implementation is created for Algorithm 1 which works by starting up one process which reads hashes piped into the program and passes them down a channel. A separate worker process then iterates through the values in the channel and calculates how similar the prefix of any individual hash is to any other hash in the dataset. Any time a given hash has a common prefix which is shorter in length than the previous shortest common prefix, the value is recorded. After the program has finished execution we find out the Maximum Hash Prefix Length of the password hashes. If every password hash was truncated to the Maximum Hash Prefix Length, no truncated hash would be unique.

Note that we concurrently read hashes from the text file into the channel, whilst running the worker to run calculations. As the file containing all the hashes measures $12.86GB$ it is impractical on most computers to load the entire file into memory prior to processing. As a result, we feed the contents of the file into a channel and the worker processes the values as they are loaded, thus reducing the amount of hashes that need to be stored in memory at a particular time. The software to perform the evaluation was written in Go due to the level of ease of introducing concurrency into a program.

With the Hash Prefix Length determined, it is then possible to run Algorithm 3 to actually group together hashes by their prefix. The simplest implementation for this would be to write a text file for each Hash Prefix containing a list of all hashes associated to that prefix. For instance; a file called $19FCF.txt$ would contain all hashes starting with $19FCF$.

At first glance, this seems like an elegant and practical solution. These files could be placed on a web server in static format, then when a third party wishes to run a query on the basis of a given Hash Prefix, they simply send a HTTP query for where the file should exist (e.g. $example.com/19FCF.txt$) and if the file exists, the client checks to see if any of the hashes within the text file match the local hash.

This approach can rapidly become impractical as the number of Hash Prefix increases in size. As hexadecimal contains 16 distinct symbols, the total amount of possible files where the Hash Prefix is 4 characters long is $16^4 = 65,536$. This seems manageable, but when the Hash Prefix is increased to 5 the total amount of possible files becomes $16^5 = 1,048,576$ and when the Hash Prefix is raised to 6 the possible amount of files becomes $16^6 = 16,777,216$. File management tools can struggle to handle such a vast quantity of files and, without adopting a directory structure, file systems can even reach their limit for the number of files in a single directory.

As an alternative, it is possible to create a simple 2 column database structure which connects Hash Prefixes to individual hashes. For the database system, our implementation uses SQLite which acts as an embedded database with all the data stored in a single binary file. As a hash should only need to be stored once throughout the database, a unique property can be applied to the database column. Additionally to accelerate the searching of hashes when querying by the Hash Prefix, an index can be set-up on the Hash Prefix column.

To expose this SQLite database as an Internet API service, a simple web application can be created to receive a Hash Prefix from a HTTP GET request and query the SQLite database for all corresponding

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3322v1 | CC BY 4.0 Open Access | rec: 5 Oct 2017, publ: 5 Oct 2017

hashes which start with that prefix. The hashes can then be returned in plain-text or in a format like JSON. Should performance become a constraint, HTTP caching can be used as necessary.

RESULTS

Upon running Algorithm 1 on Hunt (2017), the ideal Maximum Hash Prefix Length was found to be 5. Comparison of the first 2 hashes found the common Hash Prefix size to be 8, though this incrementally decreased to 7, then 6 and eventually 5 as the dataset was processed further.

We therefore set the Hash Prefix size to 5 when running Algorithm 3. Splitting the hashes into buckets by Hash Prefix would mean a maximum of $16^5 = 1,048,576$ buckets would be utilized, assuming that every possible Hash Prefix would contain at least one hash. In the final dataset we found this to be the case and the amount of distinct Hash Prefix values was equal to the highest possible quantity of buckets. Whilst for secure hashing algorithms it is computationally inefficient to invert the hash function, it is worth noting that as the length of a SHA-1 hash is a total of 40 hexadecimal characters long and 5 characters is utilized by the Hash Prefix, the total number of possible hashes associated to a Hash Prefix is $16^45 \approx 8.40 \times 10^{52}$.

The Hash Prefix $EDC9E$ had the fewest amount of hashes associated to it at 215 hashes, by contrast the Hash Prefix with the largest amount of hashes associated to it was 00000 which had 420 hashes associated to it. The median amount of hashes associated to a given Hash Prefix was 305, the mean amount was also 305 hashes.

The Hash Prefix 00000 contained 25 more hashes than the second largest Hash Prefix ($2456B$ which was associated to 395 hashes). It can be expected for the 00000 Hash Prefix to contain the most amount of hashes, as small hash values are left-padded with 0s to ensure the hexadecimal strings are 40-characters long. Despite this, the largest hash bucket was only 95% larger than the smallest, with the largest Hash Prefix still containing few enough hashes for relatively fast computational evaluation. Depending on the Hash Prefix, each bucket of hashes can range from $8.6KB$ to $16.8KB$ in size, with the median size being $12.2KB$.

Whilst not directly relevant to this evaluation; it may be of interest to note that if hashes are truncated to a single character (i.e. the Hash Prefix becomes only the first character of the hash), the Hash Prefix F becomes the most common (containing $20,104,332$ hashes), with 3 being the least common (containing $19,905,086$ hashes). The distribution of hashes by their first character is demonstrated in Fig. 4. In this configuration, the difference between the hashes associated to the largest and the smallest Hash Prefix was $\approx 1\%$.

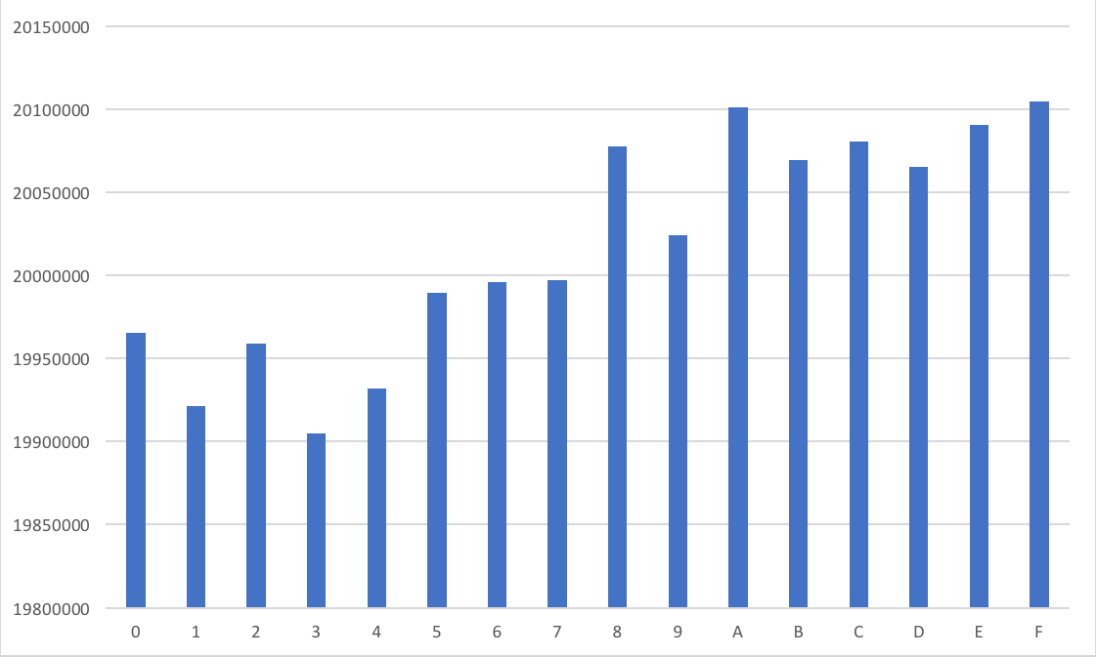

Figure 4. Number of Hashes by their First Character

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3322v1 | CC BY 4.0 Open Access | rec: 5 Oct 2017, publ: 5 Oct 2017

CONCLUSION

Password reuse poses a significant threat to the integrity of password authentication systems, however so far no alternative authentication systems have been developed which retain the complete set of benefits offered by password authentication. Additionally, restrictive password composition rules have been proven ineffective at mitigating password reuse. Even the most technologically-educated users are often unequipped to understand the risks of passwords reuse, however just-in-time fear appeal have had success at persuading users to replace breached passwords. Recent security standards have caught up with this knowledge, with the U.S. National Institute of Science and Technology adding requirements to dissuade users from reusing passwords in authentication standards.

Lists of passwords obtained from data breaches have been published on the Internet, however in order for users to check whether their password has been leaked they must either trust a third-party site with their password or download the entire database of leaked passwords and check whether it contains their leaked password. Hashing of passwords provides limited security benefits in this context, as the input hash needs to be compared to a database, password hashes cannot be protected using unique salts.

This paper presents a mechanism for providing a service to validate whether a password has been in a data breach by using an anonymized version of the password hash. An implementation is designed for password hashes to be generalized to hold property of k-anonymity. The implementation performs generalization by truncating all password hashes to a specific length until any individual password hash is indistinguishable from at least one other generalized hash. The original password hashes are then grouped together by their generalization. Remote consumers are able to request all hashes associated to a particular generalization.

For a client to evaluate whether a password has been breached, they will locally generalize a password hash by truncating it and will then query a remote service for all password hashes which start with the same prefix. From here the client can then locally validate whether their own password hash matches any of the hashes provided by the remote service.

This implementation is evaluated on a dataset of over 320 million breached passwords and we find the Maximum Prefix Length that all hashes can be truncated to, whilst maintaining the property k-anonymity, is 5 characters. When hashes are grouped together by a Hash Prefix of 5 characters, we find the median amount of hashes associated to a Hash Prefix is 305. With the range of response sizes for a query varying from $8.6KB$ to $16.8KB$, the dataset is usable in many practical scenarios.

Further work can consider how the uneven usage distribution of leaked passwords can impact k-anonymity when searching for a hash which is connected to a previously leaked password. Additionally, it can be of use to see how password security notifications affect completion rates of a website sign-up forms. There is potential to explore these topics in greater detail in further work.

REFERENCES

Bauman, E., Lu, Y., and Lin, Z. (2015). Half a century of practice: Who is still storing plaintext passwords? In *ISPEC*, pages 253–267.

Bonneau, J., Herley, C., Van Oorschot, P. C., and Stajano, F. (2012). The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 553–567. IEEE.

Campbell, J., Ma, W., and Kleeman, D. (2011). Impact of restrictive composition policy on user password choices. *Behaviour & Information Technology*, 30(3):379–388.

Comer, D. (1979). Ubiquitous b-tree. *ACM Computing Surveys (CSUR)*, 11(2):121–137.

Das, A., Bonneau, J., Caesar, M., Borisov, N., and Wang, X. (2014). The tangled web of password reuse. In *NDSS*, volume 14, pages 23–26.

Eckersley, P. (2010). How unique is your web browser? In *Privacy Enhancing Technologies*, volume 6205, pages 1–18. Springer.

Friedman, A., Schuster, A., and Wolff, R. (2006). k-anonymous decision tree induction. In *PKDD*, pages 151–162. Springer.

Gaw, S. and Felten, E. W. (2006). Password management strategies for online accounts. In *Proceedings of the second symposium on Usable privacy and security*, pages 44–55. ACM.

Ghinita, G., Karras, P., Kalnis, P., and Mamoulis, N. (2007). Fast data anonymization with low information

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3322v1 | CC BY 4.0 Open Access | rec: 5 Oct 2017, publ: 5 Oct 2017

loss. In *Proceedings of the 33rd international conference on Very large data bases*, pages 758–769. VLDB Endowment.

Grassi, P. A., Fenton, J. L., Newton, E. M., Perlner, R. A., Regenscheid, A. R., Burr, W. E., Richer, J. P., Lefkovitz, N. B., Danker, J. M., Choong, Y.-Y., Greene, K. K., and Theofanos, M. F. (2017). *NIST Special Publication 800-63B Digital Identity Guidelines*, chapter Authentication and Lifecycle Management. National Institute of Standards and Technology, U.S. Department of Commerce.

Herley, C. and Van Oorschot, P. (2012). A research agenda acknowledging the persistence of passwords. *IEEE Security & Privacy*, 10(1):28–36.

Hornby, T. (2016). Crackstation's password cracking dictionary. "`https://crackstation.net/buy-crackstation-wordlist-password-cracking-dictionary.htm`".

Hunt, T. (2017). Have i been pwned? - pwned passwords. "`https://haveibeenpwned.com/Passwords`".

Jenkins, J. L., Grimes, M., Proudfoot, J. G., and Lowry, P. B. (2014). Improving password cybersecurity through inexpensive and minimally invasive means: Detecting and deterring password reuse through keystroke-dynamics monitoring and just-in-time fear appeals. *Information Technology for Development*, 20(2):196–213.

Kumar, H., Kumar, S., Joseph, R., Kumar, D., Singh, S. K. S., and Kumar, P. (2013). Rainbow table to crack password using md5 hashing algorithm. In *Information & Communication Technologies (ICT), 2013 IEEE Conference on*, pages 433–439. IEEE.

Li, N., Li, T., and Venkatasubramanian, S. (2007). t-closeness: Privacy beyond k-anonymity and l-diversity. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 106–115. IEEE.

Lu, Y. and Tsudik, G. (2010). Towards plugging privacy leaks in the domain name system. In *Peer-to-Peer Computing (P2P), 2010 IEEE Tenth International Conference on*, pages 1–10. IEEE.

Machanavajjhala, A., Kifer, D., Gehrke, J., and Venkitasubramaniam, M. (2007). L-diversity: Privacy beyond k-anonymity. *ACM Trans. Knowl. Discov. Data*, 1(1).

Manweiler, J., Scudellari, R., and Cox, L. P. (2009). Smile: encounter-based trust for mobile social services. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 246–255. ACM.

Menezes, A. J., Van Oorschot, P. C., and Vanstone, S. A. (1996). *Handbook of applied cryptography*. CRC press.

Naor, M. and Yung, M. (1989). Universal one-way hash functions and their cryptographic applications. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 33–43. ACM.

Paar, C. and Pelzl, J. (2010). *Hash Functions*, pages 293–317. Springer Berlin Heidelberg, Berlin, Heidelberg.

Provos, N. and Mazieres, D. (1999). Bcrypt algorithm. USENIX.

Ruoti, S., Andersen, J., and Seamons, K. E. (2016). Strengthening password-based authentication. In *WAY@ SOUPS*.

Visconti, A., Bossi, S., Ragab, H., and Calò, A. (2015). On the weaknesses of pbkdf2. In *International Conference on Cryptology and Network Security*, pages 119–126. Springer.

Xiao, X. and Tao, Y. (2006). Anatomy: Simple and effective privacy preservation. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, pages 139–150. VLDB Endowment.

Yang, Y., chen, F., Zhang, X., Yu, J., and Zhang, P. (2017). Research on the hash function structures and its application. *Wireless Personal Communications*, 94(4):2969–2985.

Yang, Y., Yu, J., Zhang, Q., and Meng, F. (2015). Improved hash functions for cancelable fingerprint encryption schemes. *Wireless Personal Communications*, 84(1):643–669.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3322v1 | CC BY 4.0 Open Access | rec: 5 Oct 2017, publ: 5 Oct 2017