Journal of Cybersecurity and Privacy

Article

Password Similarity Using Probabilistic Data Structures

Davide Berardi , Franco Callegati , Andrea Melis * and Marco Prandini

Dipartimento di Informatica Scienza e Ingegneria (DISI), Università di Bologna, 40126 Bologna, Italy; davide.berardi6@unibo.it (D.B.); franco.callegati@unibo.it (F.C.); marco.prandini@unibo.it (M.P.)
* Correspondence: a.melis@unibo.it

Received: 3 November 2020; Accepted: 27 December 2020; Published: 31 December 2020

Abstract: Passwords should be easy to remember, yet expiration policies mandate their frequent change. Caught in the crossfire between these conflicting requirements, users often adopt creative methods to perform slight variations over time. While easily fooling the most basic checks for similarity, these schemes lead to a substantial decrease in actual security, because leaked passwords, albeit expired, can be effectively exploited as seeds for crackers. This work describes an approach based on Bloom Filters to detect password similarity, which can be used to discourage password reuse habits. The proposed scheme intrinsically obfuscates the stored passwords to protect them in case of database leaks, and can be tuned to be resistant to common cryptanalytic techniques, making it suitable for usage on exposed systems.

Keywords: password; Bloom Filters; hash functions; text analysis

1. Introduction

Text-based passwords are still the most common way to authenticate users against services [1]. According to a typical authorization mechanisms classification, they fall in the “what you know” category. The other categories, “what you are” and “what you have”, are most commonly used only as a second factor in the so-called multiple factor authentication (MFA) schemes [2]. An example of MFA is the, today very popular, one time password available on a personal device (e.g., smartphone) at each login attempt into a critical service. Ideally, passwords should be easy to remember, but hard to guess. In a long term game of cops and robbers, users try to base them on dictionary words [3] in order to make them easier to remember, and system administrators write policies to block these attempts, to limit the probability of password guessing. As a common reaction, users make simple variations to hide the base, easy-to-remember common word. This makes the password harder to remember, so users try to stick to the same one forever, but system administrators add expiration times to their policies. Users then adapt by making the smallest possible change at each password update. Unfortunately this still results in an insecure behavior, making life much easier to malicious actors trying to guess current passwords, because old ones are often available through leaked databases. The countermeasure to discourage this behaviour is to prevent choosing a password too similar to the previous one(s).

The problem with the simplest approaches to similarity estimation—for example based on the Levenshtein distance [4] between two clear-text strings—is that passwords are sensitive and personal data. Storing them to enable similarity checks exposes users to an additional high risk of breaches into their accounts. This demands for tools to implement similarity check among passwords of different age that can work without storing the password history.

In this paper we propose a Bloom Filters based system to solve this issue, which checks passwords similarity over obfuscated data. The effectiveness of the schemes depends on the tuning of the filter, which is thoroughly analyzed to determine the values of the parameters that ensure password secrecy, yet allowing an effective detection of similarity. The scheme natively allows the integration of a cryptographic access control method, which will be investigated in a future work.

This paper is structured as follows. Section 2 discusses a list of previous works and state-of-the-art tools that employ similar techniques to evaluate similarity between data. Some additional details of the password similarity issue, the design of our system and a possible attack on this kind of data analyzers are introduced in Section 3. Our scheme was implemented in C language for the Linux operating system, and a performance analysis is presented in Section 4. Finally we illustrate some application scenarios and propose future improvements of this work in Section 5.

2. State of the Art

The foundations for this work can be traced back to the work of Schnell et al. [5], which describes a method for querying private data over a Bloom Filter structure, with focus on medical records. The goal is to extract data which cannot be linked to the owner, which is different from the password similarity scenario considered here. In particular an attack such as the one presented in Section 3.4, could lead to a complete compromise of the scheme. Our work propose then a solution which can address the Anagram attack. Another work presented in [6] describes the application of privacy methods like differential privacy to probabilistic data structures such as Bloom Filters. This approach is vulnerable to an attack called the "profile reconstruction attack", a weakness that is due to differential privacy methods (which are not considered in our work) and not to the filter itself. Anonymized datasets, using techniques such as differential privacy, purposely introduces errors and noise in the data in order to hide the presence of specific information or to ensure that links between users and their correspondent data cannot be established. These controlled errors are compensated in large dataset: the errors do not effect the quality of the evaluations. In our use cases, users—even if forced to change password regularly—cannot generate a password dataset big enough to compensate the introduced noise. That is, anonymization techniques can affect the quality of password similarity queries, with several false positives compared to the proposed Bloom Filter approach. RAPPOR [7] is a system used by Google to get data from the Chrome browser. The data is hashed in a Bloom Filter, anonymized by introducing a perturbation on the values, and then retrieved and reconstructed at server side.

RAPPOR is used to collect binary statistic, therefore cannot be used to detect similarities between responses. It is based on binary queries and not on difference between sets. These exact matches (without false negatives and with a probability of false positives), are considered in other works. That is due to the construction and the use of the Bloom Filters. One of the closest work to this one, referenced as SSDD, is presented in [8]. It employs Homomorphic Encryption to compute the Jaccard coefficient between two Bloom Filters. One of the downsides of Homomorphic Encryption is the exponential-order complexity of the algorithms, which makes it unfeasible for many systems such as embedded ones. As the authors say, it can leverage on pre-computed values, that can be saved in a sort of cache that can speed up the encryptions and decryptions. Unfortunately if the output of the pre-computation is not saved in a secure way, this can leak information to an attacker. In SSDD as well as in the method proposed in this manuscript, the filters are used to return a real number which represent the similarity, not a boolean value, placing at the opposite side of the spectrum, with respect to RAPPOR.

Moreover in general the aforementioned work do not explicitly consider the use of secure hash functions (such as cryptographic secure hashes). The added security of these methods is therefore not in the scope of the cited papers. Other related works, which are not directly based on Bloom Filters, are the ones which employs fuzzy hash functions. The main work in this area is called Context Triggered Piecewise Hash (CTPH) [9]. CTPH fuzzy hash function is extensively used in its reference implementation, ssdeep (https://ssdeep-project.github.io/ssdeep/index.html), by platforms such as VirusTotal (https:/virustotal.com). This implementation is particularly useful to check the similarity of uploaded samples with known malware files and to perform forensic analysis [10]. To the best of our knowledge, the security of this hash function was not extensively compared to "classic" families of hash functions such as SHA and MD.

To provide a more straightforward comparison of the characteristics of existing methods and our proposal, in Table 1 we lists the main peculiarities of the various proposed approaches, and compare the different goals.

Table 1. Comparison between similar approaches. The ● symbol defines a full compliance to the row, the ◖ symbol denotes a partial compliance, and the ○ symbol denotes an absence of compliance to the row. The acronyms legend is the following one. BF: Bloom Filter, HE: Homomorphic Encryption, PHFs: Piecewise Hash Functions.

Peculiarity	**RAPPOR [7]**	**SSDD [8]**	**Schnell et al. [5]**	**CTPH [9]**	**Our Method**
Detect exact matches	●	○	○	●	○
Detect similarities	○	●	●	●	●
Can be (natively) encrypted locally	◖	◖	◖	○	●
Uses or can use secure hash function	◖	◖	●	◖	●
Main focus	Crowdsourcing	Documents	Medical Records	Malware analysis	Passwords
Main technology	BF (binary)	BF + HE	BF	PHF	BF

While these works are mainly related to privacy scenarios, in the password strength evaluation implementation we often see clear-text mechanisms to check the similarity of the last passwords. Another approach is to require the last password, but, if lost, it can result in a total lockout for the user or a subvert of the password change mechanism. Current guidelines of the National Institute of Standards and Technology (NIST) [11] do not provide any rule against password reuse, only for temporary secrets.

To the best of our knowledge, in the literature there is not a use case of application of Bloom Filters to a password management scenario. Nowadays, companies and research groups try to deal with such attacks by mixing different strategies. Companies such as Facebook claimed to have bought black-market passwords in order to analyze the similarity among passwords in order to defeat

The No. 1 cause of harm on the internet [12].

Other research groups otherwise proposed the idea of changing the structure of the password file in such a way that each user would have multiple possible passwords, sweetwords and only one of them is real. The false passwords are called honeywords. As soon as one of the honeywords is submitted in the login process, the adversary will be detected.

Despite these attempts, we claim that research has not yet provided an extensive analysis of this field. For this reason the aim of this paper is to study it, illustrate the advantages it brings, and discuss the security-related issues that it introduces.

3. Password Similarity

It is widely known that password reuse is a common behaviour which can turn into a threat if passwords get leaked online [13]. Password leaks are a common form of information leak that happens regularly (https://us.norton.com/internetsecurity-emerging-threats-2019-data-breaches.html). A similar threat appears when users are allowed to choose a new password with little variations from the previous one (e.g., password2020 changing password from password2019). This behavior is tempting especially in corporate settings which enforce a policy of frequent password expiration. This can make brute force attacks very effective, since the new password is easily computed by a limited number of mutations starting from a dictionary of leaked ones. We can describe the password mutation as a perturbation of the password with slight variants. Tools such as password crackers or word-lists generators like cupp (A password generator based on personal and open source data: https://github.com/Mebus/cupp) or johntheripper (johntheripper is a common password cracker and generator https://www.openwall.com/john/) implement various combination methods for password generation; passwords can also be generated by neural-based techniques such as adversarial generation [14]. Against these approaches, choosing a similar password is almost as insecure as choosing a dictionary password [15]. It is worth noticing that common methods to guide users to choose "robust" passwords are focused on avoiding the direct use of dictionary words. These methods have two shortcomings: users resort to variations that are easily discovered by mutation by the aforementioned tools, and there is no detection of password similarity when a password change is mandated.

For the purpose of this work, password similarity can be informally defined as the structural similarity of the text composing two password being compared, i.e., it has nothing to do with the possible underlying meaning of the string. This definition of password similarity can be used to guarantee that a user is not recycling what, in terms of actual entropy, can be considered the same password every time. A straightforward method to detect password similarity over a meaningful time span would require saving old passwords in clear-text into a database, which is obviously a despicable approach. We propose a solution based on Bloom Filters that overcomes this shortcoming.

3.1. Bloom Filters for Text Similarity

A Bloom Filter, denoted in this paper as β, is a probabilistic data structure which can be used to cache data and speed up operations such as lookup in databases [16–18]. It is composed by:

- a bucket which can be an array of bits initially set to the false value (0), we reference to its size in the number of bits as κ;
- Γ, a set of hash functions which will be used to insert and check values.

The relevant operations on a filter to measure similarity are:

- $Create(\Gamma, \kappa) \rightarrow \beta$ which generates a Bloom Filter β using the hash functions present in the set Γ with a bucket of size κ.
- $Insert(\beta, s)$ which inserts the bit string s in the Bloom Filter.
- $Check(\beta, s) \rightarrow Boolean$ which checks if the value s is not present in the filter or if it collides with a value which is already there.
- $QInsert(\beta, s, \nu)$ that inserts the string s splitting it in ν-grams.
- $Distance(\beta_1, \beta_2) \rightarrow Real$ that returns the distance between two Bloom Filters. To be comparable, two Bloom Filters must have the same bucket size κ, and need to use the same set of hash functions Γ.

An insertion operation ($Insert(\beta, s)$) of a string s in a Bloom Filter β is performed according to the following steps:

- the value that must be inserted into the bucket is hashed using the set of hash functions; The hash functions output must be re-mapped to provide indexes in the co-domain of cardinality κ.
- every bucket slot indexed by the keys got using the hash functions is set to the true value (1).

This operation therefore inserts the hashed value into the filter, setting the corresponding hash values to the true value. The process is described in Figure 1 which pictures an insertion of the strings $password1234$ and $password123!$.

The verification process of the string s presence in the filter β ($Check(\beta, s) \rightarrow Boolean$) is analogous to the insertion case:

- The element s is hashed against all the functions to get a list of indexes;
- If any index points to a false value, then the element is not present in the filter for sure. The Bloom Filter never exhibits false negatives.
- Otherwise the value could be present in the filter, but due to the collision possibility of the hash functions, the result can be a false positive.

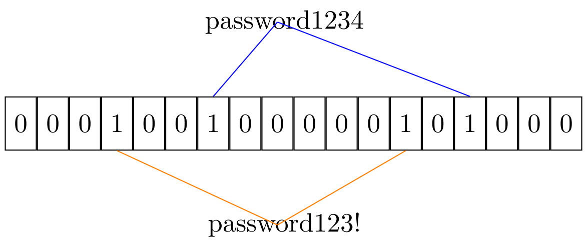


Figure 1. Insertion procedure with two strings. The strings *password*1234 and *password*123! are hashed independently. This insertion procedure processes the passwords as single items, leading to different hashed values.

This procedure is illustrated in Figure 2.

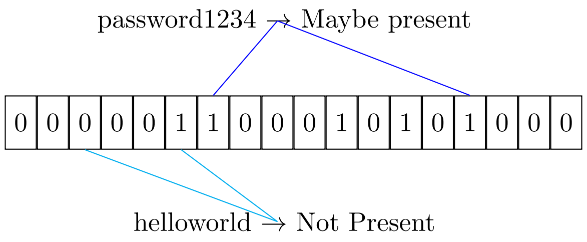


Figure 2. Check procedure with two strings. The strings *password*1234 and *helloworld* are hashed independently and the resultant indexes from the hash functions are checked in the bucket. If the lookup lead to a 0 value, the string is not present in the filter. Otherwise, it can be a value that is present in the filter or can be a collision (a false positive).

Bloom filters can be used for text similarity using distance-sensitive hash functions [19] such the ones introduced by Kirsch et al. in [20], or using an n-gram approach. This latter technique divides the string in n-grams and hashes every resultant n-gram with the hash functions present in the set, constructing a solution similar to locality-sensitive hashing [21,22]:

The hashing procedure to enable the measure of distance is presented in Figure 3. The similarity of the two sets can be calculated by using various distance definitions, by extending the check procedure to yield the number of n-grams which are the same in the two strings, independently of the order. The similarity distance of two Bloom Filters can be expressed using distance and similarities coefficients [23]. The main ones are The Jaccard coefficient [24] (which we will denote with J), the Dice coefficient [25] (which we will denote with δ) and the Cosine similarity [26] (denoted by ϕ). These coefficients can be calculated as functions between two Bloom Filters and are defined as:

$$J(\beta_1,\beta_2) = \frac{\gamma_{\beta_1,\beta_2}}{k\beta_1 + k_{\beta_2} - \gamma_{\beta_1,\beta_2}} \qquad \delta(\beta_1,\beta_2) = \frac{2\gamma_{\beta_1,\beta_2}}{k_{\beta_1} + k_{\beta_2}} \qquad \phi(\beta_1,\beta_2) = \frac{\gamma_{\beta_1,\beta_2}}{\sqrt{k_{\beta_1}} * \sqrt{k_{\beta_2}}}$$

where γ_{β_1,β_2} is the common number of true values in the sets of the two Bloom filters β_1 and β_2, and k_{β_1} and k_{β_2} are the number of true values of, respectively, the β_1 filter and β_2 filter.

The Jaccard coefficient between two filters has already been exploited, and found to be an effective method, for computing Bloom Filter similarity. This coefficient, introduced in [24] by Paul Jaccard, is shown to be suited to calculate the difference between two bit-sets, such as the one at the basis of the structure of the filter. Therefore, this coefficient is a common root of the systems based on the similarity between sets and Bloom Filters [27–29]. The main related previous works for its applications are [8,30]: these exploit the Jaccard coefficient for private documents, but are not tied to the password similarity field. The other functions were extensively analyzed in the works which we reference in this paper [5,17,31]. The choice of the best distance function is related to the application case. We propose a comparison between their performance in the field of password similarity using Bloom filters in Section 4.

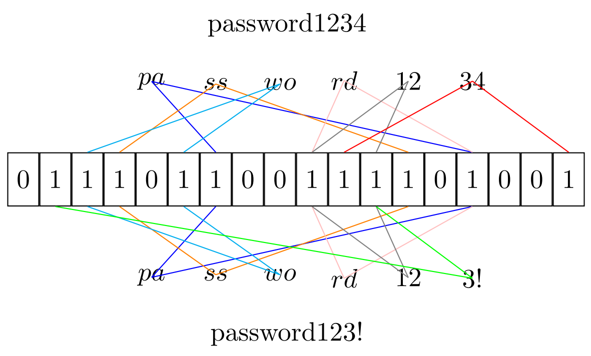


Figure 3. n-gram insertion procedure with two strings. The strings are divided into n-grams (in this case bi-grams) and hashed using an Insert operation for every n-gram.

As stated before, the structure of a Bloom Filter makes possible to have no false negatives (values that are returned as not inserted in the filter while they actually were there) and a number of false positives (values that are returned as inserted in the filter but no insertion procedure was applied on them). The percentage of false positives can be limited using the formulae presented in this section which are based on the choice of the hash function set Γ and the size of the bucket κ. The tuning of these parameters is essential to achieve a satisfactory trade-off between the utility of the query and the number of false positives. A wrong sizing of buckets or a choice of low-randomness hash functions can easily lead to a vulnerable filter (as detailed in Section 3.4) or to an unstable filter that exhibits too many false positives. This latter case is formally defined by the formulae presented in Section 3.2 and experimentally analyzed in Section 4.

3.2. Privacy Guarantees

To size the filter, some criteria can be derived from the following formula, as stated in the "classical" work by Burton H. Bloom [16],

$$fpp = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{k} \tag{1}$$

where fpp is the false positive probability of the filter. This can be calculated a-priori from the variables m, k and n, defined as:

- m, the cardinality of the set on which the filter is built;
- k, the number of different hash functions that are used to hash values into the filter;
- n, the number of elements which will be inserted into the filter.

The formula is derived from the evaluation of the probability of an item to be represented by only 0 in the Bloom Filter set. In cases of uniform distribution, as it happens with hash functions used by a Bloom Filter, this probability is defined as $1 - \frac{1}{m}$. This value needs to be exponentiated to the number of hash functions k multiplied by the number of elements. That is a consequence of the construction of the *Insert* operation: an operation of this kind is required for every inserted value. A single *Insert* operation will hash a single input string for every hash function in the hash-set Γ. The resultant false positive probability is therefore the probability of having only value set to 1 for every result of our hash function set.

From this formula we can derive the optimal values for the filter size and the number of hash functions to use. For a specific number of elements, that will be inserted in the filter, we can choose a fixed fpp to get a desired percentage of false positives. To get the formulae to calculate these optimal values, we first need to approximate the value of fpp to $fpp \approx (1 - e^{-kn/m})^{k}$. As explained in [32] the optimal number of hash functions can be found from the derivative of the fpp. Let us declare $g = k\,ln(1 - e^{\frac{-kn}{m}})$, minimizing this function g will lead us to the optimal (minimum) number of hash function. Therefore we can find that the global minimum is defined as: $k' = \frac{m}{n} ln(2)$. From this formula we can derive the optimal value for the size of the array. Replacing k′ to k in the Equation (1) we find

$$fpp \approx (1 - e^{\frac{-kn}{m}})^k = (1/2)^{\frac{m}{n}\ln(2)}$$

Which can be resolved for m, obtaining its optimal value, which is:

$$m_{opt} = -\frac{n\ \ln(fpp)}{(\ln 2)^2}$$

Obviously, these values must be integers, therefore we can apply a ceil to the result of the formulae, obtaining:

$$m' = \lceil m_{opt} \rceil = \left\lceil -\frac{n \ln fpp}{(\ln(2))^2} \right\rceil \qquad\qquad k' = \left\lceil \frac{m'}{n} \ln(2) \right\rceil$$

The rationale behind these sizing formulae comes from the observation that as the size of buckets increases, the probability of collisions decreases. Using this approach, a Bloom Filter with a controllable number of false positives can be tuned to fit any specific scenario. Conversely, as it is useful for the proposed application in order to enhance password confidentiality, a Bloom Filter can be designed in such a way to have a big number of false positives, thus obfuscating data by forcing collisions. Password confidentiality could be protected also by adopting privacy-preserving approaches, such as δ-presence [33] or differential privacy [34]. These approaches take in consideration the amount of data stored into a database, or the filter in this case, and try to anonymize the data among many false positives. In this case the approach is directly applicable to the filter which can gain a lot of advantages in terms of privacy from this approach. Specific metrics for these scenarios were studied by Bianchi et al. in [35], which proposed γ-deniability and after Xue et al. in [36].

3.3. *Analysis of the Hash Function Family*

Another key component of the filter, which impacts on confidentiality, is the distribution of the values returned by the chosen hash function family. In this kind of probabilistic data structures, it may be necessary to deploy a huge number of hash functions. Using a different algorithm for every index can be unfeasible, and in any case a large variability within the hash function set hinders a precise analysis of the randomness of the generator. To overcome these problems, the hash function set can be generated by using the well-known salt approach, as described in Figure 4. When a value has to be inserted, many random numbers (salts) are generated. Prepending these numbers to the value, and using a fixed hash function, the effect is analogous to having adopted different, random hash functions. In our implementation we used $MD5$ as the base hash function. The salts can be generated using different hash functions, such as Murmur hash [37] or secure cryptographic hash functions, such as SHA512. These are easily interchangeable, as libraries such as OpenSSL (https://openssl.org) implement the various methods using interfaces that facilitate high-level use of an hash function. In Section 4 we propose a compared analysis of the speed of two hash cryptographic functions: $MD5$ and $SHA512$. The security of the hash function was studied in numerous works [38–40], in this case, the security of the system also relies on the division in n-grams. The salts are saved in the same location as the filter set, to allow reloading the state in subsequent invocations of the algorithm. We acknowledge that this trivial solution is vulnerable to attacks if the file is saved in clear-text form, like the one described in Section 3.4.

Saving salts in a secure way seems not too challenging, and is the subject of current and future investigation aimed at making the filter immune to this kind of attacks. For example, a salt can be generated using a cipher like AES using an user provided key and a fixed payload similarly to AES-CTR mode [41]. AES, in this mode, generates an encryption stream starting from a counter c. The encryption stream is derived by encrypting this counter, initialized to a random IV. After this generation step, the encryption stream can be combined with the payload, using a xor operation. Similarly, we can extend the hash functions set of the filter with a function $GenerateHashes(key) \rightarrow \Gamma$ which generates the set of hash functions Γ starting from the key key. The salt of the functions is generated and encrypted with a key using a symmetric cypher such as AES. The security in this case relies on the key, which must be chosen by the user so as to withstand known attacks such as brute force and dictionary attacks. Using this technique, the filter set can be saved without specific protections, since it can be verified only using the chosen secret key. An in-depth analysis of the security of the encryption scheme, particularly concerning the peculiarity of having very short payloads due to the division in n-grams, will be the focus of ongoing research work. These approaches to the generation of hash functions are described in Figure 4. In the first hash set we can see that $h1$ and $h2$ are two functions generated with a random padding applied to the $MD5$ hash function. In this scenario we suppose to have a $Random(n)$ function that can generate a random string of length n. This generation, when re-applied will lead to a totally different set of hash functions, making the distance function inapplicable. That is the concept of the construction of $h1'$ and $h2'$. These functions can return different results from the functions $h1$ and $h2$ described before. Reusing the same value for the salt applied to the functions will lead to the same set of results, making the distance calculation possible. This is the concept of the third figure, which describes how, applying a fixed salt to a symmetric cipher using a secret key k the filter will lead to the same set of results.

$$h1(s) = MD5(Random(10) + s)$$
$$h2(s) = MD5(Random(10) + s)$$
$$Insert(\beta,' password1234') \rightarrow \{6, 15\}$$

0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0

$$h1'(s) = MD5(Random(10) + s)$$
$$h2'(s) = MD5(Random(10) + s)$$
$$Insert(\beta', password1234) \rightarrow \{9, 14\}$$

0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0

$$h1''(s) = MD5(fixedsalt_1 + s)$$
$$h2''(s) = MD5(fixedsalt_2 + s)$$
$$Insert(\beta'', password1234) \rightarrow \{3, 11\}$$

0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0

$$\widehat{h1}(s) = MD5(AES(k, fixedsalt_1), s)$$
$$\widehat{h2}(s) = MD5(AES(k, fixedsalt_2), s)$$
$$Insert(\widehat{\beta}, password1234) \rightarrow \{5, 9\}$$

0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0

Figure 4. Different hash functions generation. These functions will lead to different use-cases. In the first two figures we have a fully random generation which leads to different cases every time as we cannot predict the output got from *Random* function. The fourth case employs a cryptographic function and a set of fixed strings ($fixedsalt_n$) to generate the same set of hash function based on a secret key k.

3.4. Anagram Attack

The system as described in the previous sections is vulnerable to an attack which exploits the order of the discovered n-grams. This attack aims to reconstruct the password as the anagram of the various n-grams. The attack is composed of four steps:

1. Generate all the hashes for a specific n-gram;
2. Hash the n-grams into a Bloom Filter;
3. Analyze the Bloom Filter and get the position of bits set to the true value;
4. Compose the various n-grams to create a password.

This scenario can be disruptive and can lead to the full disclosure of hashed password in no time. Additionally, this kind of attack can be enhanced with the help of a word dictionary similar to the one used in classical password attacks: the search tree can be pruned by excluding the words which do not contain the n-gram. For instance, let us imagine that a user inserts the password "password!!" in the filter and splitting the input in a list of bi-grams. In this work, we use a general definition of ν-grams functions, a ν-gram function will split the input in its ν-grams, where ν is the number of characters in each substring. In the case of bigrams, obviously $\nu = 2$. Accordingly, the attacker will generate all the possible bi-grams. This, using an alphabet Δ will result in a generation of $|\Delta|^2$ bi-grams which, for the ASCII case, is $(127 - 32)^2 = 9025$ bi-grams, an operation which requires at most a couple of milliseconds on any modern CPU. After this step, the attacker will hash the bi-grams inserting them into the filter, which requires $\Theta(n)$ insertions with n as the number of bi-grams. Subsequently, the attacker can create all the possible combinations in the search space generated by the pruned alphabet of bi-grams Δ_{II}. The research can be conducted by using an incremental number of repetitions. Therefore, the number of combinations which can be generated using the corresponding Bloom Filter are:

$$\binom{\Delta_\nu}{\frac{n}{\nu}} \tag{2}$$

with n as the length of the searched string. In the case of a common password of 8 ASCII characters and a Bloom Filter constructed with bi-grams the formula will result to:

$$\binom{9025}{\frac{8}{2}} \approx 2.76 \times 10^{14} \tag{3}$$

combinations. If we consider the worst case with repetitions the formula becomes

$$\binom{\Delta_\nu + \frac{n}{\nu} - 1}{\frac{n}{\nu}} \tag{4}$$

which, in this example, will result to $\binom{9025+4-1}{4}$ which is almost the same value as seen for the non-repetition case. Passwords can be longer than eight characters to provide enough security against brute-force or dictionary attacks. For a password varying from n characters to N the number of combinations are:

$$\sum_{i=n}^{N} \binom{\Delta_\nu + \frac{i}{\nu} - 1}{\frac{i}{\nu}} \tag{5}$$

In our data-set the average password size was 11.56, therefore, limits between 8 and 14 can be evaluated resulting in:

$$\sum_{i=8}^{14} \binom{9025 + \frac{i}{2} - 1}{\frac{i}{2}} \approx 9.96 \times 10^{23} \tag{6}$$

The crypto-analysis of the attack should include the details of the filter like the size of the bucket or the number of hash functions used.

4. Experimental Analysis

The specified method to check password similarity was been implemented in C language. The hash functions used was the standard OpenSSL (https://www.openssl.org/) implementations of hash functions, in this case MD5. The system was checked on a Ubuntu 18.04 system running in a VirtualBox virtual machine with 2 virtual CPUs and 1 GB of memory. The hypervisor ran over an Intel core i7-8700 cpu which clock frequency runs at 3.2 GHz, and the host was used exclusively to run the test VM. The random data were provided by /dev/urandom to avoid blocking behaviour [42] and read to generate random hash functions. We used the following queries used to check the filter:

1. $\beta \leftarrow Create(\Gamma, \kappa)$
2. $Insert(\beta, AAAA)$
3. $Insert(\beta, BBBB)$
4. $Check(\beta, AAAA)$
5. $Check(\beta, CCCC)$
6. $Check(\beta, BBBB)$

In this case the hash dimension seemed not to influence the performances of the filter. The computational load was dominated by the generation of the salt string. The performance of the salt string generation presented in Figure 5 is the average of the results of five runs of experiments, in which the size of the salt changed from 1 (really easy to brute-force) to 1000 (really hard to brute-force). As shown in the graph, the performance decreased linearly when the salt size increased. This happened because a single random character of the salt must be multiplied by the number of hash functions present in the filter. The *QInsert* and *Distance* performances were evaluated using the following querying pattern:

1. $\beta_1 \leftarrow Create(\Gamma, \kappa)$
2. $QInsert(\beta_1, thisismypassword, 2)$
3. $\beta_2 \leftarrow Create(\Gamma, \kappa)$
4. $QInsert(\beta_2, thisismyp4ssword, 2)$
5. $\beta_3 \leftarrow Create(\Gamma, \kappa)$
6. $QInsert(\beta_3, thisismypassw0rd, 2)$
7. $Distance(\beta_1, \beta_2)$
8. $Distance(\beta_1, \beta_3)$

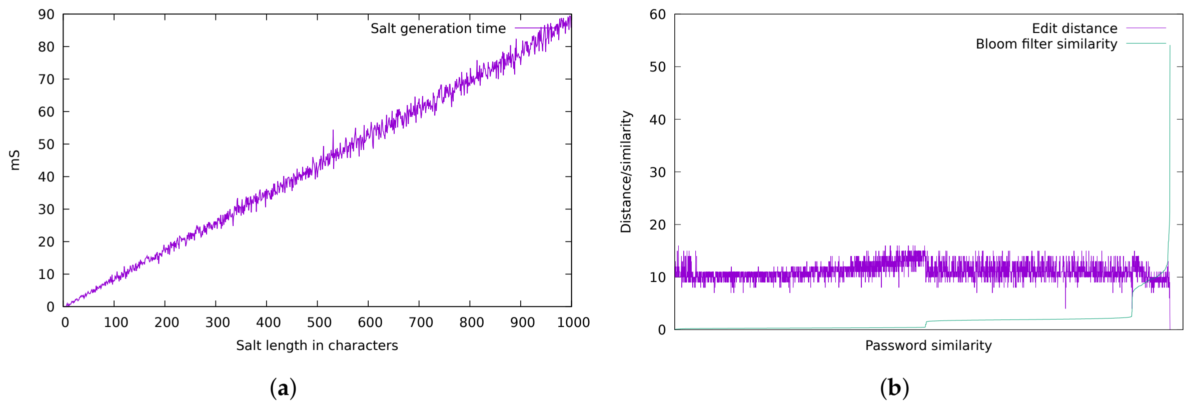


Figure 5. Speed of initialization of the filter and performances compared to edit distance. (**a**) Creation time of the filter changing the size of the salt strings. (**b**) Performances of the filter compared to the edit distance applied to the password data set.

In this test run, as in the *Insert* evaluation, the computational load was dominated by the filter generation. This system was also tested using ca. 5000 credentials from real-world leaked institutional logins. Data were clustered and analyzed using the filter, to observe which values were correctly identified as similar. The graph in Figure 5 represents the analysis of the filter's precision. The graph also shows the edit distance between the various strings.

The plots in Figure 6a–d are the experimental confirmations of the sizing criteria presented in Section 3.2. These graphs were generated using different credentials from the ones in Figure 5a,b, to confirm the general applicability of the filter. The data set was generated taking the common password database rockyou.txt (The one available in Kali Linux Distribution, https://www.kali.org/, under /usr/share/wordlists/rockyou.txt.gz), inserting in the filter 5000 random-chosen credentials, and comparing them with 5000 other credentials randomly generated using a password generator tool such as APG [43]. As clearly highlighted from the graphs, there was not a big performance gap between the two algorithms used, the secure cryptographic hash function SHA512 and the insecure MD5. Furthermore, the behaviour of the system respects the previously described formulae: a bigger filter size reduced the probability of a false positive. The hash functions number had a different impact on the scheme: it could degrade the performances if we chose a non-optimal number of functions. Choosing a low number of functions resulted in collisions by the same function, choosing too many functions resulted in filling the buckets easily, accelerating the occurrence of false positives.

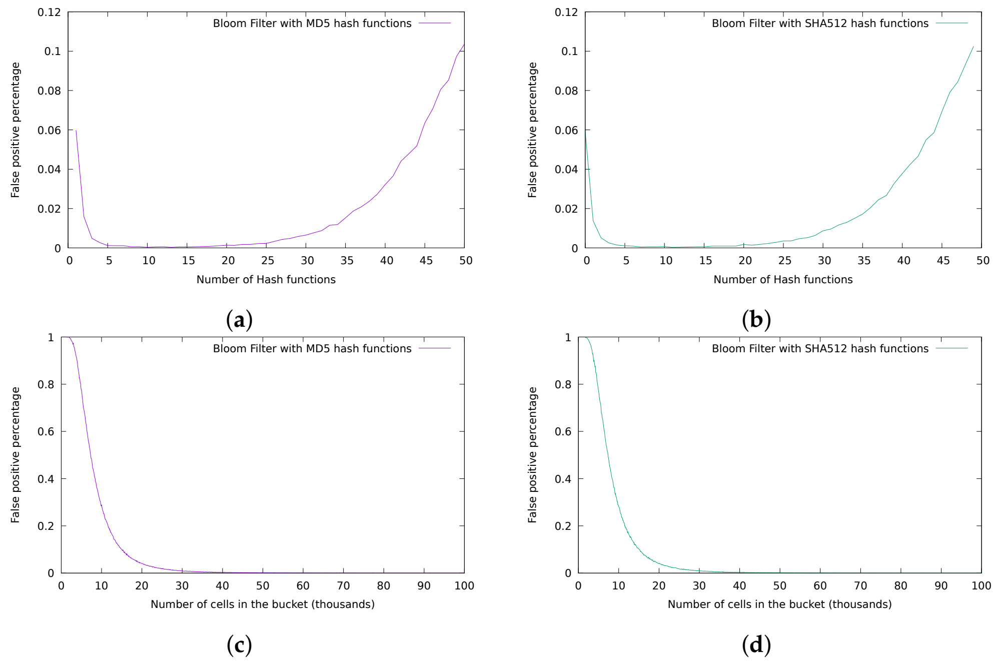
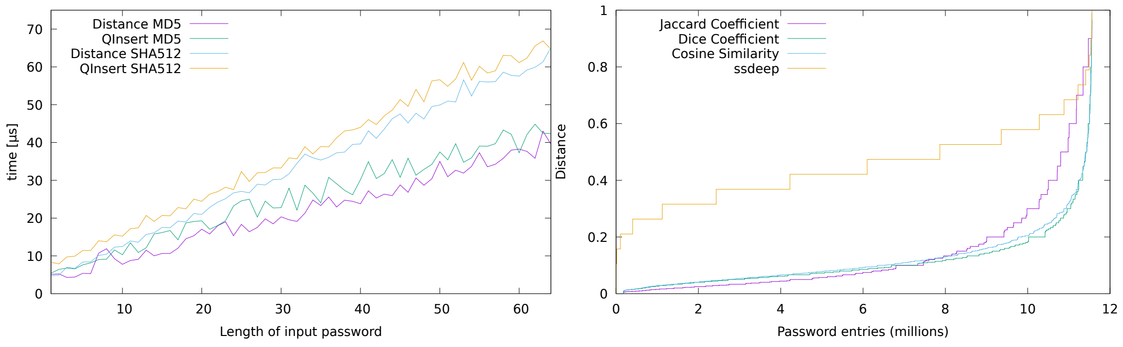


Figure 6. Performance of the filter in terms of False Positive Percentage with different sizing of Γ (number of hash functions employed) and κ (size of the internal bucket). (**a**) Performance of a Bloom Filter based on MD5 hash functions in respect to number of hash functions used. (**b**) Performance of a Bloom Filter based on SHA512 hash functions in respect to number of hash functions used. (**c**) Performance of a Bloom Filter based on MD5 hash functions in respect to size of the internal bucket. (**d**) Performance of a Bloom Filter based on SHA512 hash functions in respect to size of internal bucket.

Figure 7a represents the performances of the system, in terms of time required to execute a *QInsert* of a random string divided in bi-grams or a *Distance* operation. The graph shows that the system had a linear response to the length of the input. We analyzed lengths from 1 to 64 chars, which included the most common lengths of password strings (7, 20) [44]. Furthermore, the plot shows that the SHA512 hashing method was clearly slower than MD5. This speed gap was due to the intrinsic complexity of the algorithm. That introduced a trade off between the security of the scheme in terms of resistance to collisions and the speed of the overall procedure.

The last graphical representations in Figure 7b describe the difference between the three implemented distance measures presented in Section 3.1. The behaviour of the three functions was comparable across the spectrum of passwords lengths employed to generate the plots in Figure 7a. From the plot it is clear that Dice's method and the Cosine-based one gave more accurate results than the the Jaccard-based solution. The plot also shows the performance difference of our C implementation with a reference implementation of the CTPH scheme: ssdeep. Comparisons with the system presented in Table 1 would be infeasible due to the difference of application field (RAPPOR), or the unavailability of reference implementations and original data set used in the paper (SSDD and Schnell's solution). The data visualized in the plot confirm the fact that ssdeep was not well suited to process small inputs such as passwords.



(**a**) (**b**)

Figure 7. Comparison of the two hash functions implemented (MD5 and SHA512) and performances of the different distance metrics (Jaccard coefficient, Dice coefficient and Cosine similarity) compared to a reference implementation of CTPH [9] (ssdeep). (**a**) Performance of the system by inserting and checking random password of different lengths. (**b**) Comparison of different functions to calculate distance. Namely: Jaccard coefficient, Dice coefficient and Cosine similarity.

5. Conclusions

In this work, we described a technique to analyze password similarity maintaining a good trade-off between utility and privacy. The system we proposed uses Bloom Filters, a family of probabilistic data structures, as the core elements of its design. This choice is driven by two properties of these artifacts: the predictability of their behavior, and the deterministic crypto-analysis that can be executed over them. We claim that the proposed method can be integrated as a modular component in any kind of authentication system, to discourage the use of similar passwords over time and to prevent their use over different domains as pictured in Figure 8.

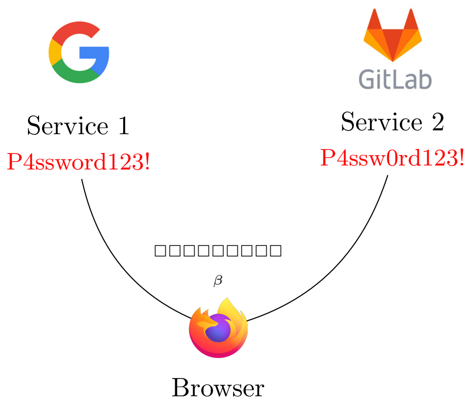


Figure 8. Application scenario: The browser can insert the passwords *P*4*ssword*123! and *P*4*ssw*0*rd*123! into the Bloom filter, checking if they are similar enough to trow a warning of password similarity.

The typical application could be a browser's plug-in that issues a warning when the distance between a new password chosen by the user and the strings saved in the plug-in is below a pre-declared threshold.

The application can be instrumented to report the results of equations in Section 3 regarding not only the similarity check result, but also the various parameters characterizing the filter, to evaluate the quality of the classification process.

Future Works

We are actually working on several improvements of the proposed password similarity algorithm:

- As stated in the Section 3, the filter can be generated using encrypted salts in conjunction with a strong cryptographic hash function (for example SHA3). This approach can be employed to ensure that data inserted in the structure are analyzable only by the owner of the secret key. The crypto-analysis of the resultant system should be explored to create secure Bloom Filters like the one presented in this paper and similar works [29].
- A comparison with deep-learning based techniques can be introduced. This comparison should therefore include an analysis of the resistance against data-set reverse engineering. We argue that a neural network-based system needs a bigger data-set than our Bloom Filter-based one, and that the former approach can be difficult to analyze using black-box classifiers.
- The analysis of the crypto-system can be improved with a more in-depth comparison with privacy preserving techniques, such as δ-presence, k-anonymity or t-closeness as described in the surveys on the topic [45,46]. As stated in Section 5, these approaches can suffer from the same issues affecting the deep-learning based one, i.e., the user cannot provide a data-set big enough to make the analysis valuable.
- Analysis of homomorphic encryption could lead to devise an encryption scheme to compute distances between encrypted strings using algorithms present in literature [47].

We claim that these analyses can lead to the creation of an useful password checker, which, while respecting user experience guidelines and security best-practices, can signal to them dangerous similarities between their passwords.

Author Contributions: Conceptualization, All; methodology, All; formal analysis, D.B. and A.M.; investigation, D.B. and A.M.; data curation, D.B. and A.M.; writing—review and editing, All; supervision, F.C. and M.P. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Schneier, B. Two-factor authentication: Too little, too late. *Commun. ACM* **2005**, *48*, 136. [CrossRef]
2. Scheidt, E.M.; Domanque, E.; Butler, R.; Tsang, W. Access System Utilizing Multiple Factor Identification and Authentication. U.S. Patent 7,178,025, 13 February 2007.
3. Stobert, E.; Biddle, R. The password life cycle: User behaviour in managing passwords. In Proceedings of the 10th Symposium on Usable Privacy and Security, Menlo Park, CA, USA, 9–11 July 2014; pp. 243–255.
4. Levenshtein, V.I. Binary codes capable of correcting deletions, insertions, and reversals. *Sov. Phys. Dokl.* **1966**, *10*, 707–710.
5. Schnell, R.; Bachteler, T.; Reiher, J. Privacy-preserving record linkage using Bloom filters. *BMC Med. Inform. Decis. Mak.* **2009**, *9*, 41. [CrossRef] [PubMed]
6. Alaggan, M.; Gambs, S.; Kermarrec, A.M. BLIP: Non-interactive differentially-private similarity computation on bloom filters. *Stabilization, Safety, and Security of Distributed Systems*; Springer: Berlin/Heidelberg, Germany, 2012; pp. 202–216.
7. Erlingsson, Ú.; Pihur, V.; Korolova, A. Rappor: Randomized aggregatable privacy-preserving ordinal response. In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, 3–7 November 2014; pp. 1054–1067.
8. Forman, S.; Samanthula, B.K. Secure Similar Document Detection: Optimized Computation Using the Jaccard Coefficient. In Proceedings of the 2018 IEEE 4th International Conference on Big Data Security on Cloud (BigDataSecurity), IEEE International Conference on High Performance and Smart Computing, (HPSC) and IEEE International Conference on Intelligent Data and Security (IDS), Omaha, NE, USA, 3–5 May 2018; pp. 1–4.
9. Kornblum, J. Identifying almost identical files using context triggered piecewise hashing. *Digit. Investig.* **2006**, *3*, 91–97. [CrossRef]

10. Li, Y.; Sundaramurthy, S.C.; Bardas, A.G.; Ou, X.; Caragea, D.; Hu, X.; Jang, J. Experimental study of fuzzy hashing in malware clustering analysis. In Proceedings of the 8th Workshop on Cyber Security Experimentation and Test (CSET'15), Washington, DC, USA, 10 August 2015.
11. Grassi, P.A.; Garcia, M.E.; Fenton, J.L. *DRAFT NIST Special Publication 800-63-3 Digital Identity Guidelines*; National Institute of Standards and Technology: Los Altos, CA, USA, 2017.
12. Facebook Buys Black Market Passwords to Keep Your Account Safe. Available online: https://www.cnet.com/news/facebook-chief-security-officer-alex-stamos-web-summit-lisbon-hackers/ (accessed on 15 December 2020).
13. Ives, B.; Walsh, K.R.; Schneider, H. The domino effect of password reuse. *Commun. ACM* **2004**, *47*, 75–78. [CrossRef]
14. Liu, Y.; Xia, Z.; Yi, P.; Yao, Y.; Xie, T.; Wang, W.; Zhu, T. GENPass: A general deep learning model for password guessing with PCFG rules and adversarial generation. In Proceedings of the 2018 IEEE International Conference on Communications (ICC), Kansas City, MO, USA, 20–24 May 2018; pp. 1–6.
15. Wood, C.C. Constructing difficult-to-guess passwords. *Inf. Manag. Comput. Secur.* **1996**, *4*, 43–44. [CrossRef]
16. Bloom, B.H. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* **1970**, *13*, 422–426. [CrossRef]
17. Mitzenmacher, M. Compressed Bloom filters. *IEEE/ACM Trans. Netw.* **2002**, *10*, 604–612. [CrossRef]
18. Gremillion, L.L. Designing a Bloom filter for differential file access. *Commun. ACM* **1982**, *25*, 600–604. [CrossRef]
19. Aumüller, M.; Christiani, T.; Pagh, R.; Silvestri, F. Distance-sensitive hashing. In Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Houston, TX, USA, 10–15 June 2018; pp. 89–104.
20. Kirsch, A.; Mitzenmacher, M. Distance-sensitive bloom filters. In Proceedings of the 2006 Eighth Workshop on Algorithm Engineering and Experiments (ALENEX), Miami, FL, USA, 21 January 2006; pp. 41–50.
21. Indyk, P.; Motwani, R. Approximate nearest neighbors: Towards removing the curse of dimensionality. In Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing, Dallas, TX, USA, 23–26 May 1998; pp. 604–613.
22. Gionis, A.; Indyk, P.; Motwani, R. Similarity search in high dimensions via hashing. *Vldb* **1999**, *99*, 518–529.
23. Brown, A.; Randall, S.; Boyd, J.; Ferrante, A. Evaluation of approximate comparison methods on Bloom filters for probabilistic linkage. *Int. J. Popul. Data Sci.* **2019**, *4*, 1–16. [CrossRef] [PubMed]
24. Jaccard, P. Le Coefficient Generique et le Coefficient de Communaute Dans la Flore Marocaine. *Mémoires de la Société Vaudoise des Sciences Naturelles* **1926**, *14*, 385–403.
25. Dice, L.R. Measures of the amount of ecologic association between species. *Ecology* **1945**, *26*, 297–302. [CrossRef]
26. Barkman, J.J. *Phytosociology and Ecology of Cryptogamic Epiphytes (Including a Taxonomic Survey and Description of Their Vegetation Units in Europe)*; Barkman Van Gorcum & Company. N. V.: Assen, Netherlands, 1958.
27. Niwattanakul, S.; Singthongchai, J.; Naenudorn, E.; Wanapu, S. Using of Jaccard coefficient for keywords similarity. In Proceedings of the International Multiconference of Engineers and Computer Scientists, Hong Kong, China, 13–15 March, 2013; Volume 1, pp. 380–384.
28. Vatsalan, D.; Sehili, Z.; Christen, P.; Rahm, E. Privacy-preserving record linkage for big data: Current approaches and research challenges. In *Handbook of Big Data Technologies*; Springer: Berlin/Heidelberg, Germany, 2017; pp. 851–895.
29. Niedermeyer, F.; Steinmetzer, S.; Kroll, M.; Schnell, R. *Cryptanalysis of Basic Bloom Filters Used for Privacy Preserving Record Linkage*; Working Paper Series, No. WP-GRLC-2014-04; German Record Linkage Center: Nuremberg, Germany, 2014.
30. Manalu, D.R.; Rajagukguk, E.; Siringoringo, R.; Siahaan, D.K.; Sihombing, P. The Development of Document Similarity Detector by Jaccard Formulation. In Proceedings of the 2019 International Conference of Computer Science and Information Technology (ICoSNIKOM), Jember, Indonesia, 16–17 October 2019; pp. 1–4.
31. Ji, S.; Yang, S.; Das, A.; Hu, X.; Beyah, R. Password correlation: Quantification, evaluation and application. In Proceedings of the IEEE INFOCOM 2017-IEEE Conference on Computer Communications, Atlanta, GA, USA, 1–4 May 2017; pp. 1–9.
32. Broder, A.; Mitzenmacher, M. Network applications of bloom filters: A survey. *Internet Math.* **2004**, *1*, 485–509. [CrossRef]

33. Nergiz, M.E.; Atzori, M.; Clifton, C. Hiding the presence of individuals from shared databases. In Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, Beijing, China, 12–14 June 2007; pp. 665–676.
34. Dwork, C. Differential privacy: A survey of results. *International Conference on Theory and Applications of Models of Computation*; Springer: Berlin/Heidelberg, Germany, 2008; pp. 1–19.
35. Bianchi, G.; Bracciale, L.; Loreti, P. "Better Than Nothing" Privacy with Bloom Filters: To What Extent? *International Conference on Privacy in Statistical Databases*; Springer: Berlin/Heidelberg, Germany, 2012; pp. 348–363.
36. Xue, W.; Vatsalan, D.; Hu, W.; Seneviratne, A. Sequence Data Matching and Beyond: New Privacy-Preserving Primitives Based on Bloom Filters. *IEEE Trans. Inf. Forensics Secur.* **2020**, *15*, 2973–2987. [CrossRef]
37. Appleby, A. *Murmurhash 2.0*; Open Source Software. 2008. Available online: https://sites.google.com/site/murmurhash/ (accessed on 30 December 2020).
38. Gueron, S.; Johnson, S.; Walker, J. SHA-512/256. In Proceedings of the 2011 Eighth International Conference on Information Technology: New Generations, Las Vegas, Nevada, USA, 11–13 April 2011; pp. 354–358.
39. Gilbert, H.; Handschuh, H. Security analysis of SHA-256 and sisters. *International Workshop on Selected Areas in Cryptography*; Springer: Berlin/Heidelberg, Germany, 2003; pp. 175–193.
40. Kim, J.; Biryukov, A.; Preneel, B.; Hong, S. On the security of HMAC and NMAC based on HAVAL, MD4, MD5, SHA-0 and SHA-1. *International Conference on Security and Cryptography for Networks*; Springer: Berlin/Heidelberg, Germany, 2006; pp. 242–256.
41. Álvarez-Sánchez, R.; Andrade-Bazurto, A.; Santos-González, I.; Zamora-Gómez, A. AES-CTR as a password-hashing function. In Proceedings of the International Joint Conference SOCO'17-CISIS'17-ICEUTE'17, León, Spain, 6–8 September 2017; Springer: Berlin/Heidelberg, Germany, 2017; pp. 610–617.
42. Gutterman, Z.; Pinkas, B.; Reinman, T. Analysis of the linux random number generator. In Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P'06): Oakland, CA, USA, 21–24 May 2006; p. 15.
43. Gasser, M. *A Random Word Generator for Pronounceable Passwords*; Technical Report; Mitre Corp.: Bedford, MA, USA, 1975.
44. Florencio, D.; Herley, C. A large-scale study of web password habits. In Proceedings of the 16th International Conference on World Wide Web, Banff, AB, Canada, 8–12 May 2007; pp. 657–666.
45. Zigomitros, A.; Casino, F.; Solanas, A.; Patsakis, C. A Survey on Privacy Properties for Data Publishing of Relational Data. *IEEE Access* **2020**, *8*, 51071–51099. [CrossRef]
46. Pannu, G.; Verma, S.; Arora, U.; Singh, A. Comparison of various Anonymization Technique. *Int. J. Sci. Res. Netw. Secur. Commun.* **2017**, *5*, 16–20. [CrossRef]
47. Cheon, J.H.; Kim, M.; Lauter, K. Homomorphic computation of edit distance. In *International Conference on Financial Cryptography and Data Security*; Springer: Berlin/Heidelberg, Germany, 2015; pp. 194–212.