Cano *et al. Cybersecurity* (2023) 6:57
https://doi.org/10.1186/s42400-023-00193-6

Cybersecurity

RESEARCH

Open Access

Performance evaluation of Cuckoo filters as an enhancement tool for password cracking

Maria-Dolores Cano[1*], Antonio Villafranca[1] and Igor Tasic[1]

Abstract

Cyberthreats continue their expansion, becoming more and more complex and varied. However, credentials and passwords are still a critical point in security. Password cracking can be a powerful tool to fight against cyber criminals if used by cybersecurity professionals and red teams, for instance, to evaluate compliance with security policies or in forensic investigations. For particular systems, one crucial step in the password-cracking process is comparison or matchmaking between password-guess hashes and real hashes. We hypothesize that using newer data structures such as Cuckoo filters could optimize this process. Experimental results show that, with a proper configuration, this data structure is two orders of magnitude more efficient in terms of size/usage compared to other data structures while keeping a comparable performance in terms of time.

Keywords Authentication, Passwords, Password cracking, Cybersecurity, Cuckoo filters

Introduction

Cybercrime has significantly grown over the last decade, impacting individuals, enterprises, and governments worldwide. The variety and depth of breaches and incidents, such as social engineering or denial of service attacks, have increasingly created a permanent concern amongst security professionals to the point that it is considered a pandemic (The Economist 2021).

Examining the latest security events compromising confidentiality, integrity, and availability of information assets, and those that resulted in the exposure of data to unauthorized parties as published in (Verizon 2021), we observe that attacks with financial motivation are the number one and the organized crime is still the leading actor. Also, it is interesting also to realize that basic web application attacks represent the second most common pattern in incidents and breaches. In contrast, social engineering ranks first in breaches and third in incidents. Commonly, some of these incidents are due to stolen credentials or password cracking. Similarly, the compromise of password-based or single-factor user credentials is identified by (VMWare et al. 2022) as one of the top concerns for many security leaders.

Understanding novel and agile forms of password cracking is paramount to creating robust defense systems and increasing the culture of cybersecurity amongst professionals and organizations. This paper proposes and evaluates a novel method to optimize the password-cracking process. The idea relies on incorporating cuckoo filter data structures to accelerate the matchmaking between guesses and targets. To evaluate this proposal, we use the widely known password cracking tool John the Ripper, due to its wide use in the cybersecurity communities, to crack an anonymized, publicly available dataset of leaked password hashes. Then, we compare the performance of the cuckoo filter data structure for matchmaking with other alternatives, specifically, binary search trees, binary search, linear search, and hash tables. Results show that (i) among the tested methods, Cuckoo filters are the second most efficient solution in terms of time consumption and that (ii) when compared in terms

*Correspondence:
Maria-Dolores Cano
mdolores.cano@upct.es
[1] Department of Information and Communication Technologies, Universidad Politécnica de Cartagena, 30202 Cartagena, Spain

of usage, Cuckoo filters are two orders of magnitude more efficient than the second best option, hash tables.

The rest of the paper is organized as follows. Section "Related work" summarizes recent trends in password-cracking methodologies from the related literature. The operation of Cuckoo filters and how to incorporate them into the cracking process is described in Sect. "Cuckoo filters". The methodology is shown in Sect. "Methodology". The evaluation of the proposal and the corresponding results are shown and discussed in Sect. "Results". The paper concludes by highlighting the most important findings and our future work.

Related work

Textual passwords are still one of the most common authentication methods (Shi et al. 2021; Bonneau et al. 2012). They are composed of a set of alphanumeric characters, where the rule of thumb is that the longer and more complex the passwords are, the higher the security offered, which raises the known tradeoff between usability and security. Password authentication goes beyond using several characters to gain access to a system. As explained by (Ali et al. 2021), passwords can be divided into three categories: token-based, biometric-based, and knowledge-based. Within this last category, we have both graphical-based (involving a mouse password entry) and textual-based passwords. Then, a textual password could belong to a direct keying or a reformation-based scheme. In this work, we focus on direct keying passwords.

The reasons to investigate password cracking are varied, from law enforcement and forensic investigations (Kanta et al. 2020; Kanta et al. 2021; Maqbool et al. 2020) to cybersecurity training (Švábenský et al. 2021) and ethical hacking (Bishop and Klein 1995; Yang et al. 2022), through understanding the effect of people's culture and background on the use of passwords (Shin and Woob 2022; Brown et al. 2004) (Bonneau 2012; Wang et al. 2017; Wang et al. 2018). Much has been written about password cracking during the last decades. It is worth mentioning that whereas password cracking is an offline technique in which the attacker (usually) has access to the password hashes, password guessing is performed online while trying to gain unauthorized access to a system.

Some research works from the related literature have proposed new methods for enhancing the performance of password cracking. In order to reduce the time needed for this operation, the work done by (Weir et al. 2009) was the first to show that it is possible to use Probabilistic Context Free Grammar (PCFG) to create either password guesses or templates via training and then employ them to crack passwords more efficiently. Numerous works have followed this trend with very interesting results (Veras et al. 2014; Houshmand et al. 2015). In (Ali et al. 2021), the authors presented a brute force algorithm to test the security of its reformation-based password scheme and compare it to other solutions from the related literature. From a different perspective, a Markov-model-based scheme was introduced by (Narayanan and Shmatikov 2005) to improve dictionary-based attacks. The authors were able to generate guesses that accelerate the process compared to rainbow attacks, and some other works have followed this research line (Dürmuth et al. 2013). Lately, with the boost in the application of machine learning and deep learning methods, several authors have presented promising proposals, such as (Hitaj et al. 2019), where the authors use a Generative Adversarial Network (GAN) to learn the distribution of real passwords and to produce guesses, (Xia et al. 2020), combining a neural network with PCFG, (Kaleel and Nhien-An 2020) studying the performance of PassGAN, and other interesting works such as (Melicher et al. 2016) (Yang et al. 2022).

To the authors' knowledge, the two most complete works on password cracking up to date were presented by (Ji et al. 2017) and (Shi et al. 2021). In (Ji et al. 2017), the authors conducted a large-scale empirical study with almost 150 million real passwords. They tested the level of correlation, the effectiveness of commercial password meters demonstrating their inconsistency, and the best strategy in terms of what algorithms are more effective (e.g., with or without training, intra- or cross-site, etc.), concluding that a hybrid option could be more appropriate. Similarly, (Shi et al. 2021) studied over 220 million plaintext passwords. Although they also concluded that there does not exist a particular better cracking method due to the impact of multiple factors, the individual analysis of each dataset showed noteworthy results. For instance, they demonstrated that whereas English datasets were better cracked with PCFG algorithms, Chinese datasets responded better to Markov-based methods.

Regarding commercial tools for password cracking, Hashcat (Advanced Password Recovery 2022) and the latest versions of John the Ripper (Open Wall 2022), e.g., Jumbo, are some of the most common applications. JtR enables several modes of operation, specifically:

- Wordlist mode, the simplest operation mode where it is only necessary to specify a wordlist and a (or some) password file; word mangling rules can be enabled.
- Single crack mode, a faster mode than the wordlist one, but that only uses login names and directory names, being able to apply more mangling rules than in the previous mode.
- Incremental mode, considered the most powerful mode because all possible character combinations are verified, with the consequent increase in time.

- Markov mode, one of the latest improvements to JtR, tests guesses using statistical analysis of similarities among known passwords.

To the authors' knowledge, neither previous research works nor commercial tools for password cracking have studied the use of cuckoo filters to enhance their operation. The following section explains the operation of this data structure, whose main advantage is double: guarantying a zero false-negative rate and a low false-positive rate while maintaining a comparative low time consumption.

Cuckoo filters

Bloom filters (Bloom 1970) and Cuckoo filters (Fan et al. 2014) are probabilistic data structures commonly used to provide membership checking. Their two main advantages are speed and memory efficiency. Both methods offer a zero false-negative rate and a non-zero false-positive rate, i.e., the answer to a membership query will always be *definitely not* (with no error) or *maybe yes* (with a false positive rate).

Some general characteristics of Bloom filters are the following. The larger the bit array (space) is, the lower the false-positive is. The more hash functions you have, the slower your Bloom filter and the quicker it fills up, and if the hash functions are just too few, we will encounter many false positives. In addition, an essential feature of classic Bloom filters is that to delete existing items, it is required to rebuild the complete filter. This fact led to the introduction of new Bloom filter variants (Cao et al. 2000; Song et al. 2005; Lim et al. 2017; Wu et al. 2021). However, some of these modifications incur a significant performance or space overhead. For this reason, (Fan et al. 2014) proposed Cuckoo filters as an alternative that allows adding and deleting items dynamically, demonstrating better performance and higher space efficiency if the false positive rate is kept below 3%.

Standard Cuckoo filters employ a modification of cuckoo hashing. As a brief note, a cuckoo hash table is composed of an array of buckets, and each bucket can hold several items. The number of items that can be stored in bucket is denoted by b. The number of candidate buckets where a new element x can be inserted equals the number of hashing functions used, k, and the output of the k hash functions gives the bucket numbers. Consequently, to insert an item x, the k hashes of x should be calculated. Then, we check if either of x's buckets is empty. If both buckets are empty, the algorithm chooses one of the candidate buckets and inserts the item. If all candidate buckets are in use, the algorithm ejects one of the existing items in one of the x's candidate buckets, re-inserting it into its alternate bucket. These relocations can be executed recursively several times until a free bucket is found or until a maximum number is reached.

The next advance in the use of these data structures was the introduction of Cuckoo filters. In contrast to Bloom filters, which store 1 bit for each item, or cuckoo-hashing, which store the complete item, a Cuckoo filter stores a *fingerprint* for each inserted item x. The *fingerprint* is a bit string obtained from the item x using another hash function. Furthermore, unlike cuckoo-hashing, Cuckoo filters will apply partial-key cuckoo hashing. When a set membership query for item x is required, the algorithm outputs true just in case an identical fingerprint of x is found. It is important to observe that the fingerprint is not the hash of x, and the original key-value pairs are not stored, consequently being non-retrievable. This fact implies that we could not calculate an item's alternate bucket as we did with cuckoo hashing. Partial-key cuckoo hashing was introduced to enable this capacity by stating that only two hash functions will be used. In other words, only two bucket candidates given by $h_1(x)$ and $h_2(x)$ are employed, and these two hashes will follow the rule depicted in (1). This rule guarantees that x's alternate bucket can be obtained using the location of either bucket $h_1(x)$ or $h_2(x)$ and x's fingerprint due to the XOR operation. That is, there is no need to know x. Figure 1 depicts an example of inserting elements in a Cuckoo filter and checking membership.

$$\begin{aligned} h_1(x) &= hash(x) \\ h_2(x) &= h_1(x) \cdot hash\left(x'sfingerprint\right) \end{aligned} \tag{1}$$

Following the same philosophy of Bloom filters, the Cuckoo filter's false-negative rate is zero, and the false-positive rate ε is shown in (2), where n is the number of items expected to be inserted into the set, m is the size of the bit array, b is the number of items that a bucket can hold, and f is the fingerprint length in bits. The Bloom filter's false positive rate is shown in (3), where k is the number of hash functions (Reviriego et al. 2020). Table 1 summarizes some characteristics of Bloom filters and Cuckoo filters that should be considered in their implementation.

$$\varepsilon_{Cuckoo\ filter} \approx \left(\frac{8 \cdot \frac{n}{m \cdot b}}{2^f} \right) \tag{2}$$

$$\varepsilon_{Bloom\ filter} \approx \left(1 - \left(1 - \frac{1}{m} \right)^{k \cdot n} \right)^k \tag{3}$$

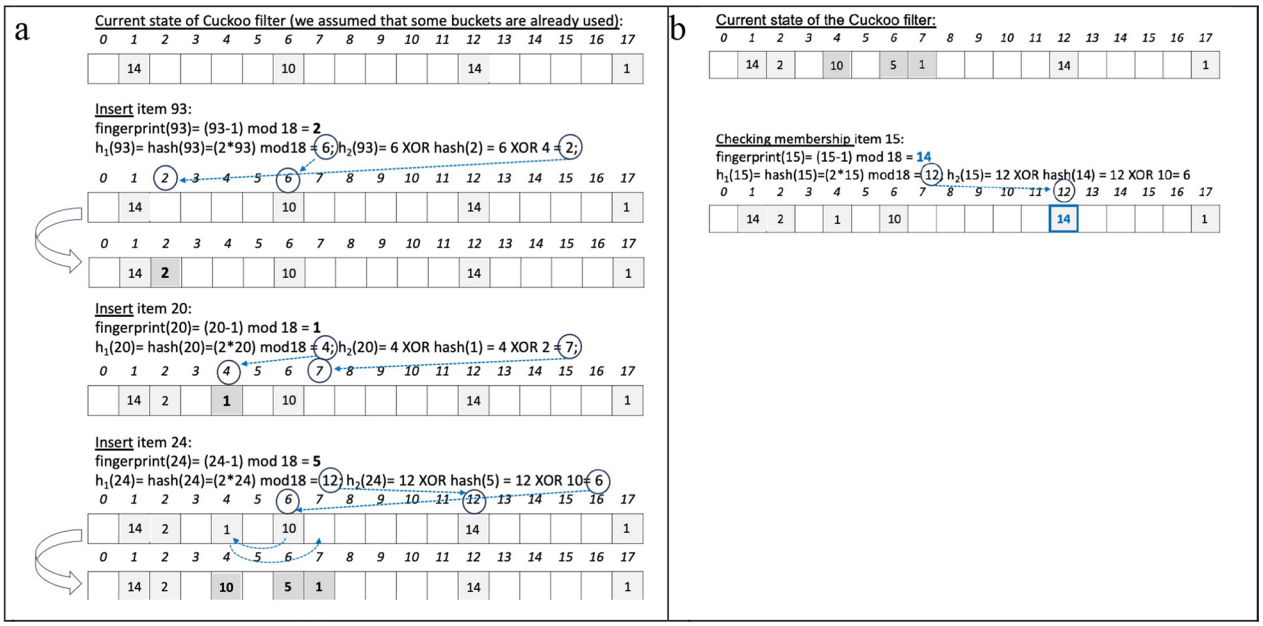

Fig. 1 An example of the Cuckoo Filter operation using a bit array of size 18 ($m=18$), each bucket can store only one fingerprint ($b=1$), and the fingerprint length in bits is 5 ($f=5$). The hash function is hash(x) = 2*x mod 18 and fingerprint(x) = ($x-1$) mod 18. (a) Inserting elements: To insert item 93, the value fingerprint(93) = 2 should be stored either in position $h_1(93)=6$ or in position $h_2(93)=2$, because position 6 is occupied the value 2 is inserted in position 2. To insert item 20, the value 1 should be stored either in position 4 or in position 7, both are available, so the value 1 is inserted in position 4. To insert item 24, the value 5 should be stored either in position 12 or in position 6, and both buckets are occupied. Consequently, we randomly choose bucket 6 and displace the current fingerprint 10 to its alternate bucket 4. To calculate the alternate bucket, we know that $h_1(x)=6$, then $h_2(x)=6 \oplus \text{hash}(10)=6 \oplus 2=4$, so the new location should be position 4. Because bucket 4 is also in use, we displace the current fingerprint 1 to its alternate bucket 7 (if $h_1(x)=4$ then $h_2(x)=4 \oplus \text{hash}(1)=4 \oplus 2=7$), which is free. (b) Verifying membership: To test the membership of element 15, we calculate its fingerprint, fingerprint(15) = 14, and the two possible buckets where the fingerprint could be located (positions $h_1(15)=12$ and $h_2(15)=6$), because position 12 stores the fingerprint of item 15, we can state that the item belongs to the set with a non-zero false-positive rate

Table 1 Comparative characteristics of Bloom filters and Cuckoo filters (Notation: $m \equiv$number of buckets for Cuckoo or size of the array for Bloom; $n \equiv$number of items; $b \equiv$bucket size for Cuckoo; $\alpha \equiv$load factor $0 \leq \alpha \leq 1$; $k \equiv$number of hash functions; $f \equiv$fingerprint length in bits for Cuckoo; n/a≡not applicable)

	Standard Bloom filter	**Cuckoo filter**
Lookup operation	$O(k)$	$O(1)$ maximum of two buckets to check
Insert operation	$O(k)$	$O(1)$ longer as load factor approaches capacity
Delete operation	N/a	$O(1)$ maximum of two buckets to inspect
Number of hash functions k_{opt}	$\left(\frac{m}{n}\right) \cdot \ln 2$ to minimize false positive rate	2 to achieve the close-to-best space efficiency for the most common acceptable false positive rate
Minimal fingerprint size (bits)	N/a	$\left\lceil \log_2\left(\frac{1}{\varepsilon_{Cuckoo\ filter}}\right) + \log_2(2b) \right\rceil$
Best number of entries per bucket b	N/a	4 to achieve the close-to-best space efficiency for the most common acceptable false positive rate
Bits per item (load factor = 95.5%)	$1.44 \log_2\left(\frac{1}{\varepsilon_{Bloom\ filter}}\right)$	$1.05\left(\log_2\left(\frac{1}{\varepsilon_{Cuckoo\ filter}}\right) + \log_2(2b)\right)$

Fig. 2 An example of NTLM hashes from the dataset in hexadecimal format

Methodology

In this section, we explain how to improve the performance of password cracking by incorporating Cuckoo filters. Our hypothesis is that the time required to perform password cracking can be reduced if (1) the corresponding target hashes are inserted into a Cuckoo filter and (2) the guessed passwords are checked against this data structure.

In order to evaluate this proposal, we selected one leaked password dataset with New Technology Lan Manager (NTLM) hashes that may have resulted from different types of attacks.[1] The NTLM authentication protocols authenticate users and computers based on a challenge/response mechanism that proves to a server or domain controller that a user knows the password associated with an account. The format is shown in Fig. 2, where we can see the password's hash and the number of times this password had been seen in the source data breaches separated by a colon. It is important to state that we only employ these password hashes for research purposes, and no personally identifiable information is being used, explored, or disclosed.

The general method of password cracking, for instance, using well-known tools such as JtR, is as follows. A password guess is created. Then, the corresponding password hash of the guess should be computed and tested (compared) against the target hash value. This target hash value is frequently included in a large file with hundreds or thousands of other "leaked" hashes. Therefore, we could wonder, once the guess is generated and the program looks for a match to see if it is a valid crack, what is the most common technique used for searching? There could be many options: linear search, a hash map, etc. Particularly with JtR, if salts are correctly used, there is only one target hash per salt value. Therefore, it would be a one-to-one comparison, and there would be no need for a search. In other cases, it is more efficient to compute multiple hashes at once from multiple password guesses using newly available computing power such as Graphics Processing Units (GPU). In this case, the comparison would be many-to-one (Open Wall 2017). However, for some systems that use NTLM hashes, the comparison step requires using a searching algorithm, such as bitmap structures, hash tables, or linear searches (Open Wall 2023). NTLM hashes are still used by current Windows. They are very fast and don't use salts, so when cracking those the comparison step is in fact a bottleneck. Consequently, we propose to include a Cuckoo filter in the search process to improve its efficiency by speeding up the comparison stage.

In order to evaluate the performance of our proposal, we designed several tests with the following methodology (Fig. 3). Given that our goal is to measure the effectiveness of incorporating a Cuckoo filter into the cracking process, we decided not to modify JtT or Hashcat, which will be considered for future work, but to compare the performance of a linear search, a hash table, a binary search tree, a binary search, and a Cuckoo filter under the same case study of password cracking.

First, we split the leaked password dataset with NTLM hashes into n sub-datasets of different sizes (see Table 2). Each sub-date is called $target_i$ ($i=1..n$). For each sub-dataset $target_i$, we generate the corresponding Cuckoo filter using the cuckoo filter library for Python (Guan 2019). That is, we fill the Cuckoo data structure by inserting the items of the sub-dataset $target_i$. Next, we crack the passwords of the sub-dataset $target_i$ using JtR and store the obtained cracked hashes in a temporary file tmp_i. New random, fake hashes that do not belong to any of the cracked passwords are then inserted into the temporary file tmp_i. The reason is to get as close as possible to reality so that there will be correct and incorrect guesses during the cracking process. Therefore, the temporary file tmp_i includes the correct hashes, i.e., the target hashes corresponding to the cracked passwords, and filler (fake) hashes (50%).

Then, the search process is implemented using as input the temporary file tmp_i, emulating the hashes created by the password cracking program, and checking if the hashes that it contains match the target hashes $target_i$ (and therefore, meaning that we were able to crack the passwords). The search time t_i^s is computed as a performance indicator. Figure 3 summarizes the evaluation process. The comparison has been carried out using a Virtual Machine (Virtual Box) with Debian (64-bit) (Kali Linux), 8192 MB RAM, six processors, 80 Gb storing capacity, VMSVGA 128 Mb graphics, and Python 3.11.2. In

[1] https://haveibeenpwned.com/Passwords

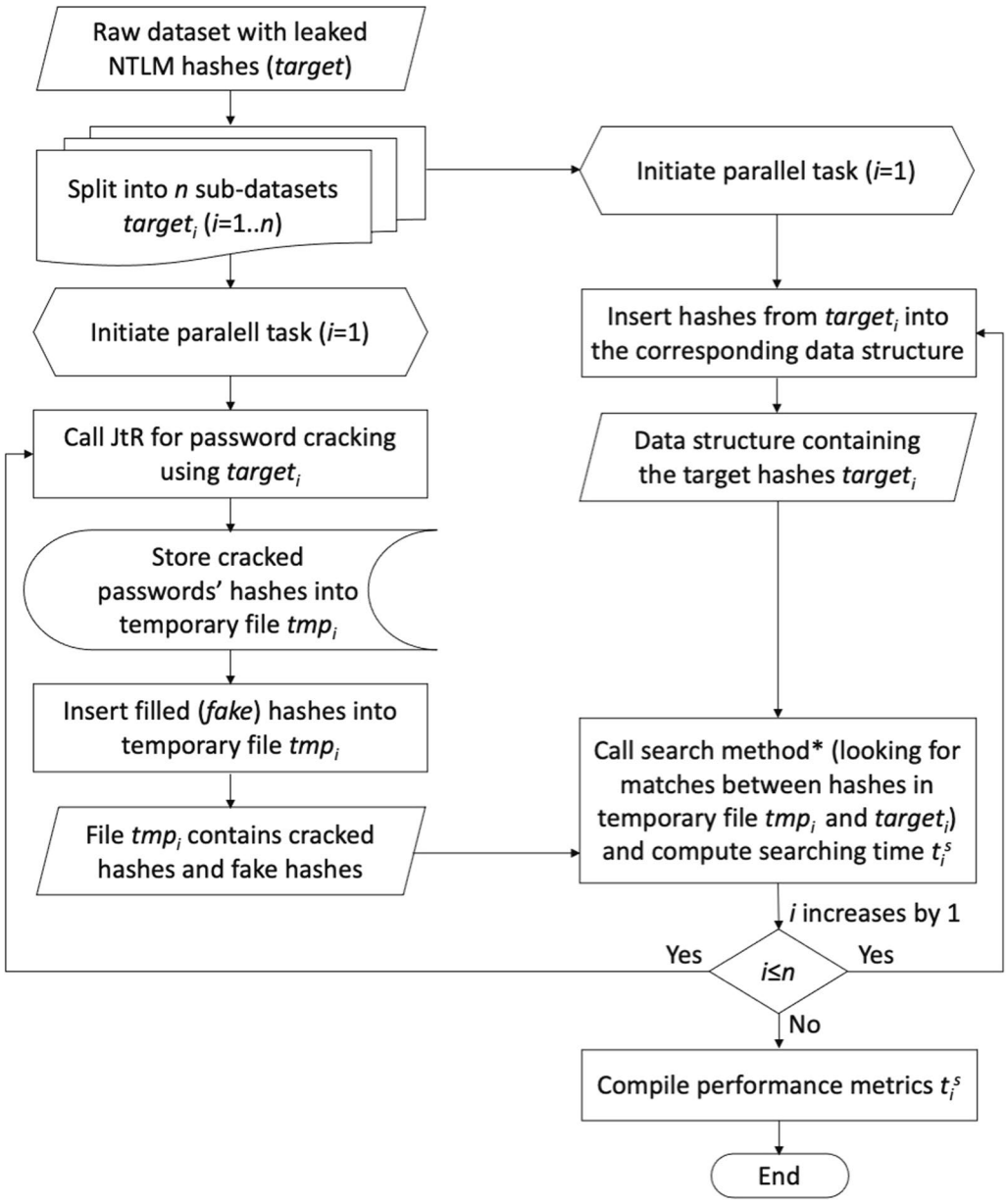

Fig. 3 Flow diagram for the evaluation process. This process is repeated for each data structure under study. (*) The search methods are linear, hash table, binary search tree, binary search, and Cuckoo filter

addition, specifically for the Cuckoo filter data structure, additional tests are carried out varying its configuration parameters to evaluate their impact on performance.

Results

Results are illustrated in Fig. 4. In Fig. 4a, we compare the different search methods with the Cuckoo Filter using its default parameters ($m = total\ number\ of\ hashes$, $b = 4$, $f = 8$). We can observe that the time required to find matches between the computed hashes (from the guess passwords) and the target hashes for each search algorithm t_i^s is lower for the hast table. The second and third faster methods are cuckoo filter and the binary search methods, respectively. Nevertheless, we can deduce from Fig. 4b that the sizes of the data structures are not negligible, making the Cuckoo filter the most optimal one in two orders of magnitude compared with the hash table.

Table 2 Details of the files and hashes employed for the performance evaluation

Subdata set	Size of the original *leaked* file (MB)	# Hashes of the original leaked file	#Cracked hashes using JtR	#Fake hashes introduced	#Total number of hashes	Size of the filled file *tmp* (MB)
1	134	3,897,669	3,897,669	3,897,668	7,795,337	340
2	67	1,948,834	1,948,834	1,948,833	3,897,667	168
3	34	974,417	974,417	974,416	1,948,833	84
4	17	487,208	487,208	487,207	974,415	42
5	8.3	243,604	243,604	243,603	487,207	20.7
6	4.2	121,802	121,802	121,801	243,603	10.3
7	2.1	60,901	60,901	60,900	121,801	5.1
8	1	30,450	30,450	30,449	60,899	2.6
9	0.5341	15,225	15,225	15,224	30,449	1.3
10	0.267	7612	7612	7611	15,223	0.6587
11	0.1335	3806	3806	3805	7611	0.3318
12	0.0667	1903	1903	1902	3805	0.1679

Linear search is simple but inefficient for large data sets. Although it is easy to implement, its search time increases linearly with data size, making it unsuitable for large data sets. However, it could be a viable option for very small data sets. Binary search significantly improves performance compared to linear search using an ordered data structure. It is an acceptable method for large data sets and presents a logarithmic search time. In addition, it offers high accuracy and has no false positives or negatives. However, it requires the data to be pre-sorted and can be more complex to implement. Binary search trees offer reasonable search time and high accuracy. Their performance improves compared to linear search but is inferior to binary search. The size of the data structure depends on the number of nodes and the depth of the tree, which implies significant memory consumption. Binary search trees are useful when a tidy data structure is required and accuracy is critical.

Hash tables are highly efficient regarding search time, as they allow direct access to items through a hash function. They provide high search speed, especially for large data sets. However, the size of the data structure grows with the number of elements, which can be a limitation in terms of memory consumption. In addition, it is critical to select a suitable hash function and consider possible collisions. Cuckoo filters are an attractive option for balancing speed, accuracy, and memory consumption. They offer fast search time and a compact data structure. The memory consumption of the Cuckoo filter is lower compared with the other methods, as depicted in Fig. 4 (b), which makes it an efficient alternative.

Correctly setting the Cuckoo filter parameters is important to minimize false positives. To study the Cuckoo filter's performance with more detail, we show in Figs. 5, 6, and 7 the effect of using different bucket sizes ($b=2$, $b=4$, $b=6$), different fingerprint sizes ($f=8$, $f=16$, $f=24$, $f=32$), and different filter sizes ($m=total$, $m=2{\cdot}total$, $m=3{\cdot}total$ $m=4{\cdot}total$, where *total* is the total number of hashes to be tested). For $b=4$, i.e., each bucket can store four fingerprints, we can see in Fig. 5 that varying the size has little impact on the searching time. However, using a fingerprint length of more than 8 bits decreases the false-positive rate to almost zero. This effect is also noticeable for $b=2$ and $b=6$, i.e., each bucket can store two or six fingerprints, respectively, as shown in Figs. 6a and 7a. The reason is that a larger fingerprint reduces the likelihood of a collision.

As expected from the definition of a Cuckoo filter, larger values of m increase the size of the data structure. If this parameter growths then there will be more "space" available to store items. Nevertheless, this does not mean that the performance will be better as shown in Figs. 5, 6, and 7. Observing these figures for $f=16$, $f=24$, and $f=32$ bits, we can see that the best configuration in terms of searching time is found for the combination $\{b=2, f=16\}$ (Fig. 6.b) and $\{b=6, f=24\}$ (Fig. 7c). In the former, there is a slightly higher size cost because the best result is obtained with $m=3$, opting for $\{b=2, f=16\}$ with $m=1$ as the best configuration considering the searching time and size.

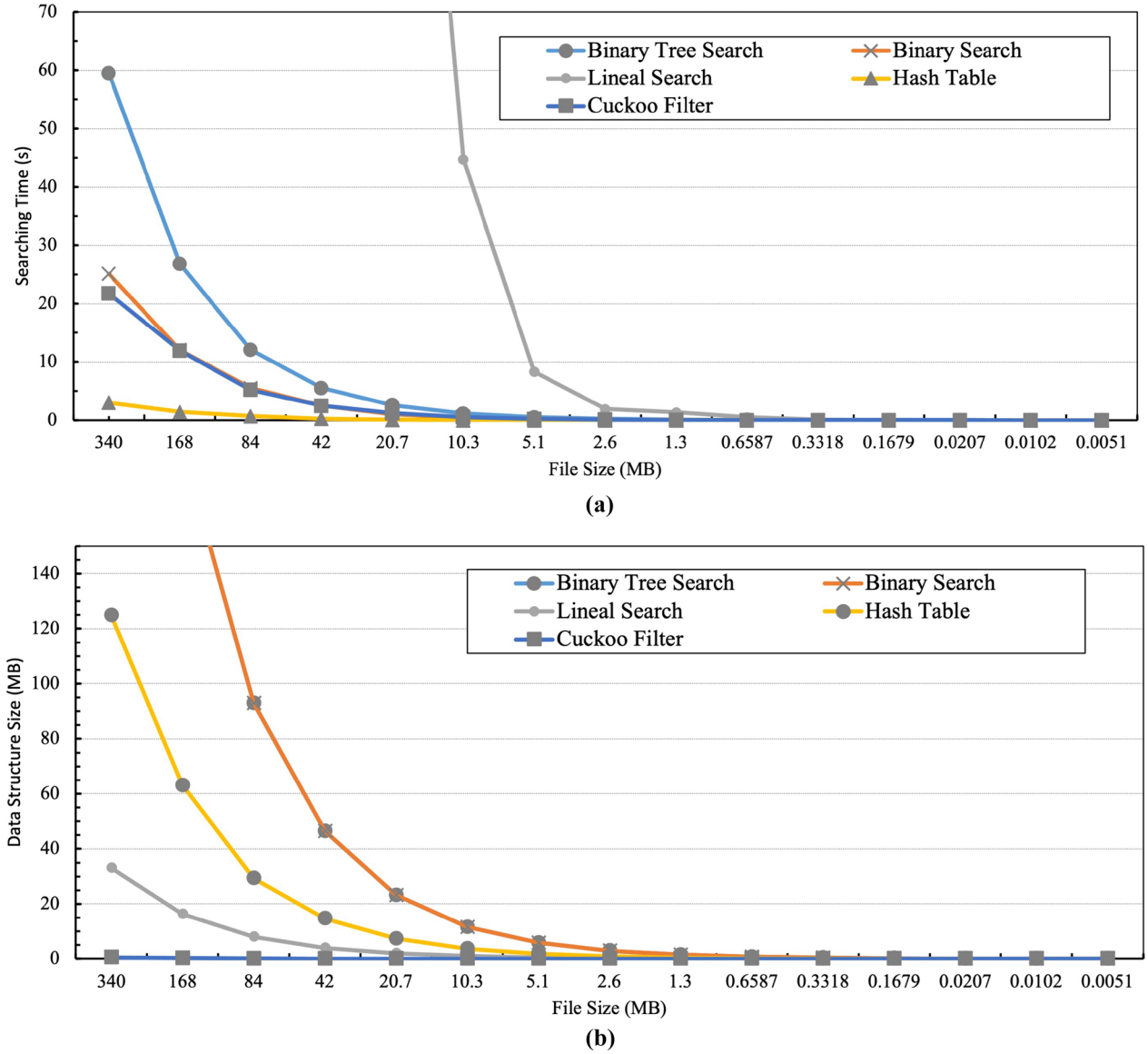

Fig. 4 Comparison of the use of different data structures for matchmaking. **a** Time in (s) required for matchmaking, i.e., finding matches between the computed hashes (from the guess passwords) and the target hashes for each search algorithm **b** Size in (MB) of the data structure used for matchmaking. The represented size is the output of the Python primitive sys.getsizeof() that returns the size of an object in bytes, considering that only the memory consumption directly attributed to the object is accounted for, not the memory consumption of objects it refers to

Conclusion

New data structures, such as the Cuckoo filters, have been proven efficient in several computer network applications. Nevertheless, its use in security has been limited mainly to authentication tasks. In this work, we have introduced a new use of Cuckoo filters as a valuable tool within the password-cracking process. The proposed method is particularly interesting for systems that use NTLM hashes because, in this scenario, the comparison step between generated hashes and target hashes requires a searching algorithm. Results show that whereas there is no a direct reduction in time, the gain in terms of memory usage is of two orders of magnitude compared to commonly employed data structures, which opens the door to further research in this direction.

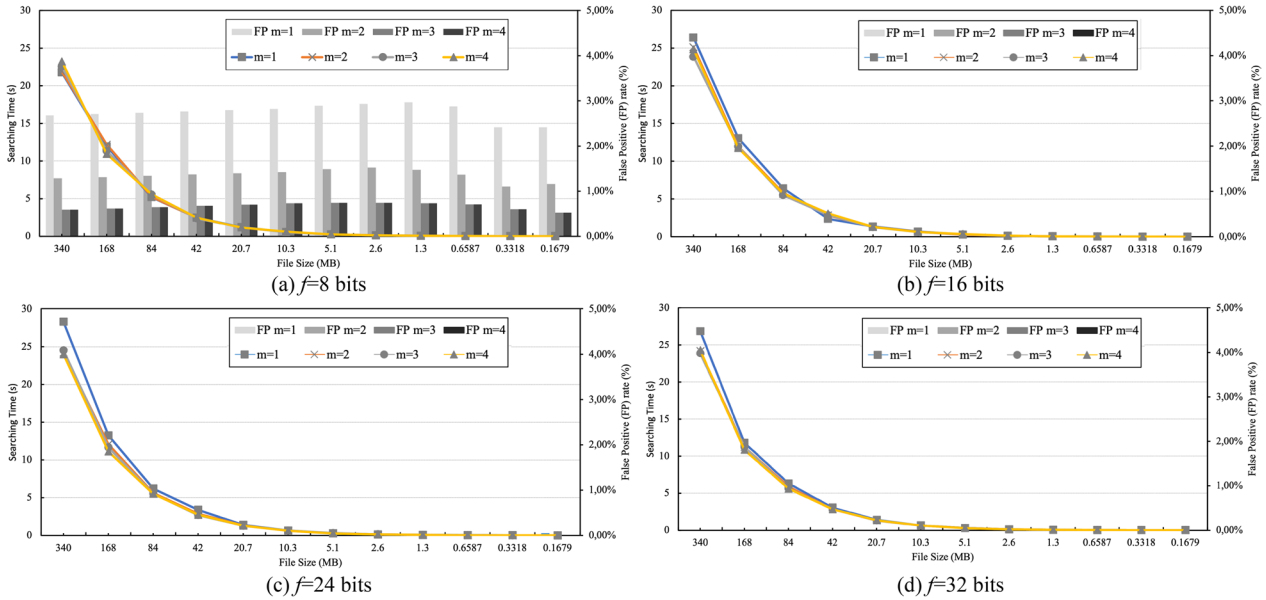

Fig. 5 Cuckoo filter performance in terms of searching time and False-Positive (FP) rate for $b=4$ varying the size m and the fingerprint length f. $m=x$ means that the size of the Cuckoo filter is x times the total number of hashes to be tested (e.g., $m=2$, the size of the Cuckoo filter is twice the total number of hashes to be tested)

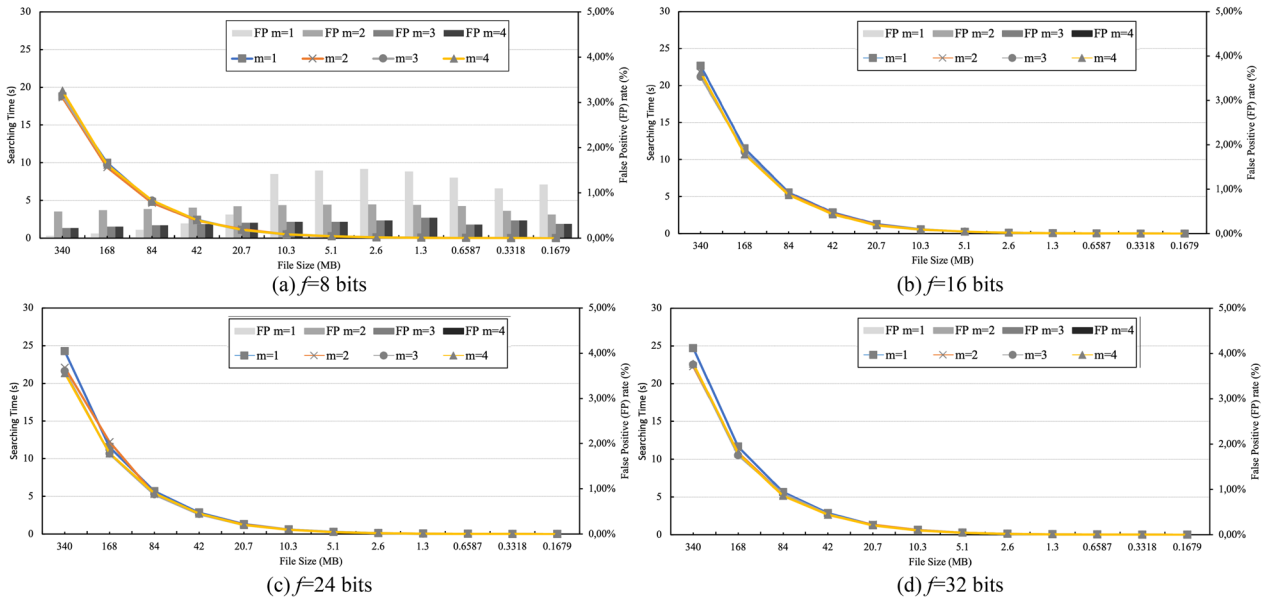

Fig. 6 Cuckoo filter performance in terms of searching time and False-Positive (FP) rate for $b=2$ varying the size m and the fingerprint length f. $m=x$ means that the size of the Cuckoo filter is x times the total number of hashes to be tested (e.g., $m=2$, the size of the Cuckoo filter is twice the total number of hashes to be tested)

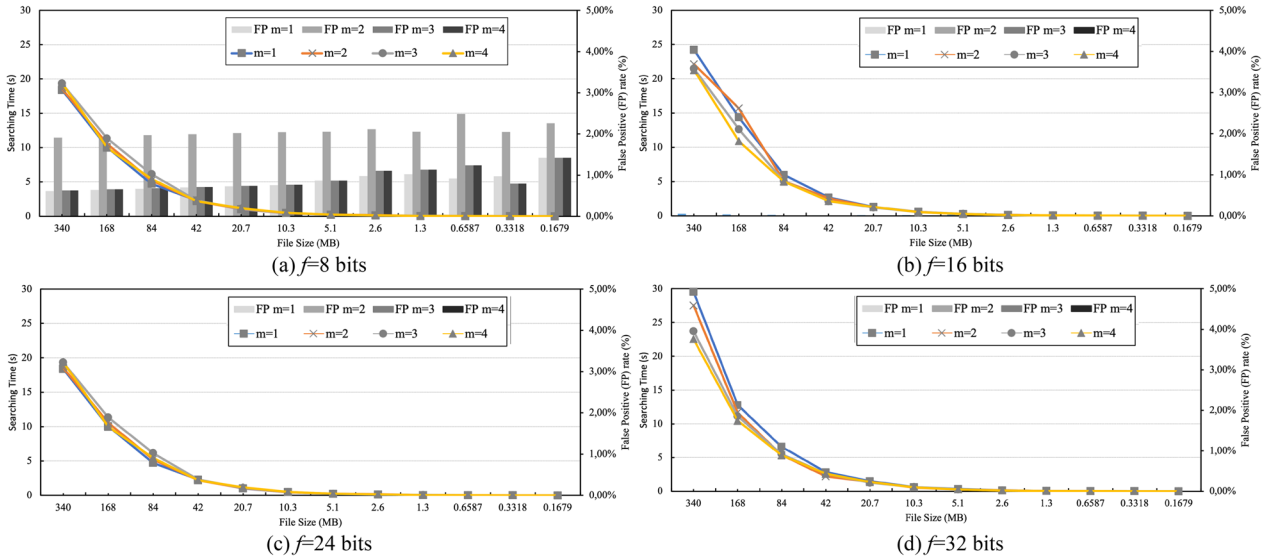

Fig. 7 Cuckoo filter performance in terms of searching time and False-Positive (FP) rate for $b=6$ varying the size m and the fingerprint length f. $m=x$ means that the size of the Cuckoo filter is x times the total number of hashes to be tested (e.g., $m=2$, the size of the Cuckoo filter is twice the total number of hashes to be tested)

Author contributions

M-DC: Conceptualization, data curation, formal analysis, funding acquisition, investigation, methodology, project administration, resources, validation, visualization, writing—original draft, writing—review and editing, supervision. AV: Data curation, formal analysis, investigation, software, visualization, writing—original draft. IT: Conceptualization, validation, investigation, methodology, visualization, writing—review and editing.

Funding

This research was supported by the research Grant PID2020-116329 GB-C22 funded by MCIN/AEI/10.13039/501100011033.

Availability of data and materials

The datasets used during the current study are available from the corresponding author on reasonable request.

Declarations

Competing interests

The authors declare that they have no competing interests.

Received: 24 July 2023 Accepted: 22 September 2023
Published online: 09 November 2023

References

Advanced Password Recovery (2022) Hashcat. https://hashcat.net/hashcat/. Accessed 9 Feb 2022

Ali M, Baloch A, Waheed A, Zareei M, Manzoor R, Sajid H, Alanazi F (2021) A simple and secure reformation-based password scheme. IEEE Access 9:11655–11674. https://doi.org/10.1109/ACCESS.2020.3049052

Bishop M, Klein DV (1995) Improving system security via proactive password checking. Comput Secur 14:233–249. https://doi.org/10.1016/0167-4048(95)00003-Q

Bloom BH (1970) Space/time trade-offs in hash coding with al lowable errors. Commun ACM 13:422–426. https://doi.org/10.1145/362686.362692

Bonneau J (2012) The science of guessing: Analyzing an anonymized corpus of 70 million passwords. In: In Proc. IEEE symposium on security and privacy. San Francisco, CA, pp 538–552

Bonneau J, Herley C, Oorschot PC van, Stajano F (2012) The quest to replace passwords: a framework for comparative evaluation of web authentication schemes. In: In Proc. 25th USENIX security symposium. San Francisco, CA, pp 175–191

Boudreau PE, Bergman WC, Irvin DR (1994) Performance of a cyclic redundancy check and its interaction with a data scrambler. IBM J Res Dev 38:651–658. https://doi.org/10.1147/rd.386.0651

Brown AS, Bracken E, Zoccoli S, Douglas K (2004) Generating and remembering passwords. Appl Cogn Psychol 18:641–651. https://doi.org/10.1016/j.fsidi.2021.301186

Cao FP, Almeida J, Broder AZ (2000) Summary cache: a scalable wide-area Web cache sharing protocol. IEEE/ACM Trans Netw 8:281–293. https://doi.org/10.1109/90.851975

Dürmuth M, Chaabane A, Perito D, Castelluccia C (2013) When privacy meets security: Leveraging personal information for password cracking. https://arxiv.org/abs/1304.6584

Fan B, Andersen DG, Kaminsky M, Mitzenmacher MD (2014) Cuckoo filter: practically better than bloom. In: In Proc. CoNEXT'14. Sydney, Australia

Fowler G, Noll LC, Vo K-P, et al (2019) The FNV Non-Cryptographic Hash Algorithm. In: IETF Draft

Guan J (2019) Cuckoofilter Python Library

Hitaj B, Gasti P, Ateniese G, Perez-Cruz F (2019) PassGAN: A Deep Learning Approach for Password Guessing. In: In Applied cryptography and network security: 17th international conference, ACNS 2019. Bogota, Colombia, pp 217–237

Houshmand S, Aggarwal S, Flood R (2015) Next Gen PCFG password cracking. IEEE Trans Inf Forensics Secur 10:1776–1791. https://doi.org/10.1109/TIFS.2015.2428671

Ji S, Yang S, Hu X et al (2017) Zero-sum password cracking game: a large-scale empirical study on the crackability, correlation, and security of passwords. IEEE Trans Dependable Secur Comput 14:550–564. https://doi.org/10.1109/TDSC.2015.2481884

Kaleel M, Nhien-An L-K (2020) Towards a new deep learning based approach for the password prediction. In: In Proc. IEEE 19th international conference on trust, security and privacy in computing and communications (TrustCom). pp 1146–1150

Kanta A, Coisel I, Scanlon M (2020) A survey exploring open source Intelligence for smarter password cracking. Forensic Sci Int Digit Investig 35:1–11. https://doi.org/10.1016/j.fsidi.2020.301075
Kanta A, Coray S, Coisel I, Scanlona M (2021) How viable is password cracking in digital forensic investigation? Analyzing the guessability of over 3.9 billion real-world accounts. Forensic Sci Int Digit Investig 37:1–11. https://doi.org/10.1016/j.fsidi.2021.301186
Lim H, Lee J, Yim C (2017) Complement bloom filter for identifying true positiveness of a bloom filter. IEEE Commun Lett 19:1905–1908. https://doi.org/10.1109/LCOMM.2015.2478462
Maqbool Z, Aggarwal P, Pammi VSC, Dutt V (2020) Cyber security: effects of penalizing defenders in cyber-security games via experimentation and computational modeling. Front Psychol 11:1–11. https://doi.org/10.3389/fpsyg.2020.00011
Melicher W, Ur B, Segreti SM, et al (2016) Fast, lean, and accurate: modeling password guessability using neural networks. In: In Proc. 25th USENIX Security Symposium (USENIX Security 16). Austin, TX, USA, pp 175–191
Narayanan A, Shmatikov V (2005) Fast dictionary attacks on passwords using time-space tradeoff. In: In Proc. 12th ACM SIGSAC conference on computer communications and security. New York, NY, pp 364–372
Open Wall (2022) John the Ripper. https://www.openwall.com/john/. Accessed 9 Feb 2022
Open Wall (2017) Large-scale password hashing
Open Wall (2023) Password security: past, present, future
Reviriego P, Martínez J, Larrabeiti D, Pontarelli S (2020) Cuckoo filters and bloom filters: comparison and application to packet classification. IEEE Trans Netw Serv Manag. https://doi.org/10.1109/TNSM.2020.3024680
Shi R, Zhou Y, Li Y, Han W (2021) Understanding offline password-cracking methods: a large-scale empirical study. Secur Commun Networks 2021:1–16. https://doi.org/10.1155/2021/5563884
Shin Y, Woob SS (2022) PasswordTensor: analyzing and explaining password strength using tensor decomposition. Comput Secur. https://doi.org/10.1016/j.cose.2022.102634
Song H, Dharmapurikar S, Turner J, Lockwood J (2005) Fast hash table lookup using extended bloom filter: An aid to network processing. In: In Proc. Applications, technologies, architectures, and protocols for computer communications. Philadelphia, PA, pp 181–192
Švábenský V, Celeda P, Vykopal J, Brišáková S (2021) Cybersecurity knowledge and skills taught in capture the flag challenges. Comput Secur 102:1–14. https://doi.org/10.1016/j.cose.2020.102154
The Economist (2021) To stop the ransomware pandemic, start with the basics. In: Econ. https://www.economist.com/leaders/2021/06/19/to-stop-the-ransomware-pandemic-start-with-the-basics. Accessed 17 Feb 2022
Veras R, Collins C, Thorpe J (2014) On the semantic patterns of passwords and their security impact. In: In Proc. network and distributed system security (NDSS) symposium. San Diego, CA, pp 1–16
Verizon (2021) Verizon 2021 Data Breach Investigations Report
VMWare, Kroll, RedCanary (2022) The state of incident response 2021: It's time for a confidence boost
Wang D, Cheng H, Wang P et al (2017) Zipf's law in passwords. IEEE Trans Inf Forensics Secur 12:2776–2791. https://doi.org/10.1016/j.fsidi.2021.301186
Wang D, Cheng H, Wang P, et al (2018) A security analysis of honeywords. In: In Proc. 25th Annual Network and Distributed System Security Symposium. San Diego, CA, pp 1–15
Weir M, Aggarwal S, Medeiros B de, Glodek B (2009) Password cracking using probabilistic context-free grammars. In: In Proc. 30th IEEE Symposium on Security and Privacy. Oakland, CA, pp 391–405
Wu Y, He J, Yan S et al (2021) Elastic bloom filter: deletable and expandable filter using elastic fingerprints. IEEE Trans Comput PP: https://doi.org/10.1109/TC.2021.3067713
Xia Z, Yi P, Liu Y et al (2020) GENPass: a multi-source deep learning model for password guessing. IEEE Trans Multimed 22:1323–1332. https://doi.org/10.1109/TMM.2019.2940877
Yang K, Hu X, Zhang Q et al (2022) VAEPass: a lightweight passwords guessing model based on variational auto-encoder. Comput Secur 114:1–12. https://doi.org/10.1016/j.cose.2021.102587

Publisher's Note