Dynamic Calculation of Password Salts for Improved Resilience Towards Password Cracking Algorithms

Arun K Mani
Applied Computer Science
University of Winnipeg
Winnipeg, Canada
mani-a@webmail.uwinnipeg.ca

Oluwasola Mary Adedayo
Applied Computer Science
University of Winnipeg
Winnipeg, Canada
m.adedayo@uwinnipeg.ca

***Abstract* — Passwords have been an integral part of our lives from the dawn of the internet and keeping them secure has been of paramount importance. Each attempt to secure our digital lives has been met with increased complexity and scope in attacks to compromise security measures. This paper explores a novel methodology to calculate password salts by using the password itself and multiple texts to generate lookup values into a text corpus that is then used to calculate salt values dynamically and on the fly. The proposed method allows an authentication system to use salts for password storage without storing the salts in a database where they might be compromised.**

Keywords—Password security, Password salting, Network security, Encryption, Salts.

I. Introduction

The word *password* has been made a part of our common parlance over the past 20 years due to the proliferation of the internet. The concept of a password is very simple at its core; a word or phrase that only an individual knows and which grants the individual access to an online resource. If used properly, passwords can be a very powerful tool for restricting access, making sure that only authorized people can access specific information. Passwords have been around for a very long time, millennia in fact. One of the first recorded uses of passwords or watchwords as they were referred to back in those periods was by the Roman military in 322 BC. Since then, they have been used throughout history from the prohibition era where people used passwords to get into speakeasys to World War II where information would be encrypted by a password to keep state and military secrets from falling into the wrong hands. It wasn't until the 1960s that passwords were used in the way that we think of them today. That form of usage started in 1960 when passwords were used to provide access to a computer called the Compatible Time Sharing System (CTSS) [1].

By the 1970s, the method in which passwords were being stored was changing. In 1974, Robert Morris invented what is known as hashing [2] or the one-way conversion of a string into a unique string of fixed length. Hashing is unique in that; each string produces a unique hash for all practical intents and purposes. Thus, each password has one hash, and the password cannot be reverse-engineered from that hash. In primitive password authentication systems passwords were hashed and then stored in the database. When a password was entered the password was hashed and then compared with the stored value. If the two hashes matched it meant that the password that was provided was correct and the user was authenticated. This ensured that if the database was compromised only the hashes would be leaked. Since the hashes were one way an attacker would not be able to get the password from the hashes.

Since the 1970s, the use of passwords and the way they are stored have changed to mitigate different types of password attacks. One such approach involves the use of password salts where both the passwords and the salt are stored separately from each other. This approach is susceptible to latency in password access and ensues latent security risk which may emerge from a breach of both the password database and the salt database. In this paper, we present an improved approach for maintaining passwords in authentication systems. Our approach embraces the benefits of using the salting method for password management but presents an alternative method of addressing these limitations.

The paper is structured as follows. In section II, we discuss some of the related works and attack methods that are often used to expose passwords, highlighting how each of these approaches has led to the current state of password management methods. Section III discusses the main contribution of this paper – an alternative method for generating password salts. Section IV describes our experimentation and the results of the proposed approach. In section V, we give the conclusion and highlight some future related to this research.

II. Background and Related Works

This section provides an overview of the different methods that are often used in attempts to compromise a password. We discuss the impact of each of these methods based on the amount of time and computational resources it takes on average to break or compromise a password by using such a method and describe how password management techniques have improved to address these attack methods.

A. *Brute Force Attacks*

The brute force attack method is the simplest method of password cracking. In practice, it involves the specification of conditions that a password should satisfy and an automated generation of all possible passwords that satisfy the condition [3]. The brute force attack method is very time-consuming and resource-intensive due to the need to try all the possible sets of generated passwords. The method is often defeated by timeout counters built into most websites and authentication systems. Fig. 1 represents the limitation of the brute force approach; as the character count of a password that needs to be cracked increases the time needed to break the password increases exponentially.

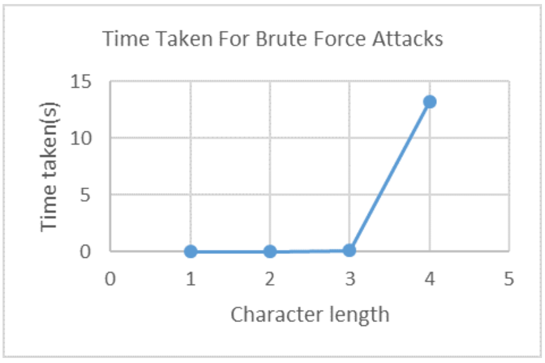

Fig. 1: Time taken by brute force method

B. Dictionary Attacks

Dictionary attacks exploit the fact that humans are creatures of habit. Assuming every individual always chooses passwords that are randomized and long, then passwords would be impervious to brute-force attacks. Unfortunately, most individuals do not operate this way. People tend to choose words and phrases that are in our common parlance and vocabulary as their passwords [4]. This set of words can be considered as a dictionary. A malicious attacker can go through such a dictionary and try each password to see if it is a fit for each person. This attack vector is somewhat mitigated by the fact that most, if not all modern authentication systems have timeouts and lockouts built into them, which means that an attacker will be denied access to the account if they exceed a certain number of attempts. This is where an attacker can employ a variation of the dictionary attack. This method is called password spraying. Password spraying works by trying the same password on multiple different accounts [5] over a certain amount of time, thereby circumventing the restriction on the number of attempts that one individual can make for a given account.

Another interesting approach to compromising passwords is to combine brute force attacks with dictionary attacks. It is not uncommon to see passwords such as '*password1*' and '*qwerty123*', in practice. These patterns can be generated by the hybrid attack. This hybrid attack can be remarkably effective in coming up with variations of passwords that are typically hard to guess or too hard to generate using the brute force method. This method can be extended once more to generate rule-based attacks [6]. The rule-based attack is a heuristic password attack algorithm that generates a password by applying various transformation rules (e.g., insertion, deletion and reordering) to transform strings in a password dictionary. The main drawback of the rule-based attack is the fact that the substitution and transformations take up significant computational resources, and this slows down the process of cracking the password. This approach is also known as a mangled dictionary attack.

As password selection rules became more complex and storing large words became easier, the phonetic dictionary attack has gained popularity. This involves words in the English language that sound similar to each other, for example, words such as '*Night*' and '*Knight*'. Dictionaries can be built using these substitutions. Another dictionary-based attack that we consider and evaluate in our experiments is the Markov chain attack [7], which is based on some of the

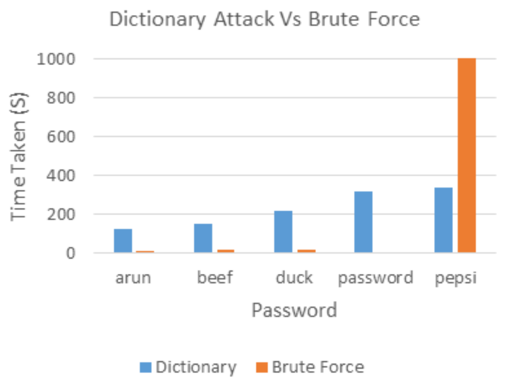

Fig. 2: Time taken by brute force compared to dictionary attacks

statistical intricacies of the English language. For instance, one could say that the letter *i* often follows the letter *e*, and use this as a way to make it easier to guess passwords. These varying optimizations make dictionary-based attacks far more effective than brute-force attacks. Fig. 2 compares the time that it takes for each of the words that we tested to be cracked using the respective algorithms. We see that for short words the brute force algorithm is faster, as there are very few letter combinations but as shown in Fig. 1, the time taken grows exponentially, the five-letter word '*pepsi*' took 8200 seconds to crack. Fig. 2 shows only up to 1000 seconds to preserve scale.

C. Rainbow Table Attacks

Rainbow table attacks involve the concept of a hash. A hash is a mathematical function that takes in a string or a message and converts it into a unique fixed-length string. No two strings have the same hash for all practical intents and purposes. Collisions are possible in certain instances, but these are very rare. This property of a hashing function is leveraged in password management systems. Most secure systems do not store end-user passwords on their databases to prevent the possibility of the databases getting compromised and revealing passwords to a potential attacker. Most authentication systems hash a password at the time of creation and then store the hash, thus the system does not have to store the password itself, thereby increasing its security since hashes cannot be reverse-engineered even in the event of a compromised database access. This opens up the attack vector

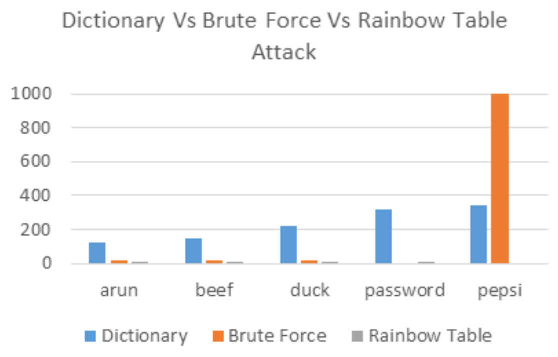

Fig. 3: Time taken for brute force attacks compared to dictionary attacks and Rainbow table attack

for the rainbow table attack. Due to the declining price of computation power, it is now financially viable to create hashes of multiple words, thus the process of finding the password from a leaked hash has been reduced to performing a join and then checking the results [8]. Multiple rainbow tables are available on the internet that are gigabytes in size which can be used for this purpose. For instance, the *loweralpha#1-10* is 179 GB [9]. If the hash of a password that is a common word is leaked then it would be relatively simple to find the password with that information.

To thwart such an attack, the concept of salt was introduced. The salt is a string that is appended to a password before it is hashed so that the hash is not found in a rainbow table. The salt is a random string that is pre-computed and stored in a database along with the password hash.

III. Main Contributions

In most authentication systems, we have a situation where the passwords and the salts are stored separately from each other. However, given multiple password leaks that have occurred in the past, we know that databases can be breached. Thus, it is not out of the realm of possibility to surmise that a breach could affect both the password database and the salt database. This represents, in our opinion, a latent security risk that has so far gone unnoticed. A determined attacker could put together the password and the hash and gain access to a system. In addition to this, since the hashes are stored in databases, they are susceptible to latency. In industries such as e-commerce or online shopping, a delay may cause a buyer to abandon the transaction. The database access delays for most cloud providers are in hundreds of milliseconds. For instance, Snap Inc. uses Google Cloud Platform to process its databases and their instance of KeyDB had an access time of 49-133ms [10]. The same trend holds for Amazon and its customer elasticache. In the case of Amazon's AWS, their access times were in the range of 100ms. These time frames were significantly shortened by implementing the databases in special servers built by the cloud provider. Thus a solution that removes the need for cloud databases would need to employ resources that are cost-competitive with cloud storage solutions.

Our proposed approach for handling salts relies on the use of computations rather than database accesses. Computation in the cloud is roughly as cheap as storage for small workloads, such as single-row retrieval. Small computations can be even more cheaply executed by using spare elastic capacity, such as the service offered by Amazon called EC2. Thus, a computational way to calculate salts in a fast manner would be more secure and not a burden financially. To accomplish this, the proposed algorithm needs to accomplish two main objectives: 1) the algorithm must be fast and 2) it must be dependable for the salt to work. An algorithm that produces different salts each time the algorithm is run won't be useful in personal or enterprise scenarios.

One way that this can be achieved is by utilizing the power of randomness. Most random number generators are in fact pseudorandom generators, meaning that the numbers aren't truly random but just seem so. Thus, any algorithm that requires randomness must find a source of truly random numbers. In the history of computing, this has led to some very creative solutions. For example, Cloudflare [11], the company that provides DNS and DDoS protection, uses lava lamps and harnesses the random motion of the lava particles to generate truly random numbers. To accomplish this without using lava lamps or any other hardware, one can turn to precomputed random number tables. For our experiments in this paper, the tables from random.org were used but for an enterprise implementation, random numbers collected or generated by the developers themselves would be a more secure option.

Another method of achieving randomness is to use a text block or file from which one can pull strings and numbers to generate the salt. These three items, i) the hash ii) the random number file, and iii) the text file, provide us with a secure way to generate salt phrases. The security comes from the fact that three separate items need to be compromised for the salts to be calculated. This is not considering the algorithms that are required to calculate the salt. In section IV we show that the salt that is derived depends on the specific algorithm and mappings that are used. Thus, if the hashing algorithm is kept secret, as most companies and entities do, then this can be considered as an additional level of security. This multilevel approach to security allows for greater overall security. Since the files involved are all relatively small in size, they can be held in memory and thus be accessed by a parallelized program providing password-salting services to a large number of people with very low latencies that are theoretically orders of magnitude faster than finding values from database accesses. The experiments in this paper were conducted in Python and we expect that the results would be even better if the experiment were done in a statically typed language such as C++.

An improvement that could be done to speed up the processing time for salts is to employ vectorization. Due to the emergence of artificial intelligence (AI), consumer hardware such as GPUs have been optimized for AI calculations and these calculations often take the form of matrixes. A conversion of the input documents into matrixes or tokenizing them and then running the algorithm on a suitable GPU or AI accelerator would yield significant speed increases [12]. But with the high demand for GPUs and AI accelerators, running this algorithm on an AI-focused card could result in expenses that are not required and may negate the cost advantage that our dynamic calculation of password salts has over the traditional method.

Fig. 4 provides an overview of our proposed algorithm. The algorithm uses a *random.txt* file and a corpus of text as sources of randomness. The text file filled with random numbers was obtained from random.org, a website that publishes random numbers to be used in services like cryptography. Hypothetically, if one was unconvinced of the randomness of the information provided by a random number provider, it would be relatively easy to use random atmospheric noise [13] to generate create random number as does Random.org. The same goes for the corpus of random text that the block diagram refers to. For our experiments, we used the full works of Shakespeare in text form as our corpus of text. While not truly random, as the letters are in semi-predictable phrases due to word structure, this concern is obviated by the large size of the text used, as the probability of the random subset of the text that is selected being accurately guessed is close to zero.

At the start of the Algorithm 1, we hash the password using the CRC64 algorithm. The CRC64 algorithm is chosen because the hash produced is completely numeric. Alternatively, we could use other hashing algorithms and then convert the resulting alphanumeric hash into a number, but this conversion process would add an overhead to the process. Once the hash is calculated we designate the first and second digits as *start* and *end* values the third digit is used as the multiplier which will be used later in the algorithm.

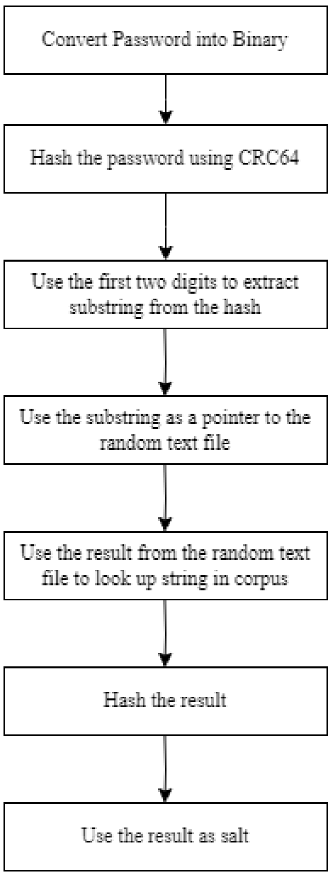
Convert Password into Binary
Hash the password using CRC64
Use the first two digits to extract substring from the hash
Use the substring as a pointer to the random text file
Use the result from the random text file to look up string in corpus
Hash the result
Use the result as salt

Fig. 4: Block Diagram of the algorithm

Using the *start* and *end* values, we splice the hash value; we refer to this as the *kernel*. The *kernel* is now multiplied by the *multiplier* that we previously obtained. Using the original hash value, we calculate a stopping value. The *kernel* and *stopping* values are then used as splicing values to extract some text from the text corpus. The selected text is then hashed to get a value that can be used as a salt.

Multiple aspects of the above algorithm fall within the control of the password management system owner or designer. This includes the random numbers used, the text corpus used, and the order and manner in which the calculations are done. The calculations used in Algorithm 1 are arbitrary and used to illustrate the feasibility of the method. Modifying the bits that are used to calculate the starting and stopping values or using a range of numbers to calculate these values would make the method more secure and point to a larger section of the text corpus for hashing. Also, using a wider range of values for *start* and *end* would make the algorithm more resilient as this would cover more of the text corpus, thus making the final selection of the hash even more randomized. The above algorithm explains the bare concepts of our method.

IV. Experiments & Results

Our experiments were conducted on a computer with 24 Gigabytes of RAM, with an Intel i5-9300H running Windows 19045.3693 and Python 3.9.14 in a stock Anaconda container. The experiments focus on examining how the proposed algorithm would work in the real world and evaluating its performance. We considered some of the known passwords

Algorithm 1 Dynamic Calculation of Password Salt

```
Input: Password in string form (password)
Output: Alphanumeric hash
1: $content$ = Data from random.txt file
2: $seed = CRC64(password)$
3: $start = seed[0]$
4: $end = seed[1]$
5: $multiplier = seed[2]$
6: if $multiplier == 0$ then
7:     $i = 3$
8:     while $i <$ length of seed do
9:         if $seed[i] == 0$ then
10:            $multiplier = seed[i]$
11:            break
12:        end if
13:        $i = i + 1$
14:    end while
15: end if
16: if $start > end$ then
17:     $kernel = seed[end : start]$
18: else
19:     $kernel = seed[start : end]$
20: end if
21: $kernel = kernel * multiplier$
22: $stopping = seed\_str[4 : 6]$
23: $extract = content[kernel : kernel + stopping]$
24: $text = open(TextCorpus.txt)$
25: $random1$ = numeric value based on $extract$
26: $random2$ = Choice of random number
27: $selection = text[random1 : random1 + random2]$
28: $salt = hash(selection)$
29: return $salt$
```

that people were using to protect their information. Information security companies such as Nord publish passwords used by people each year for each region. Table 1 shows the 10 most popular passwords.

The experiments were conducted on a limited dictionary, obtained from the internet that measures about 14 gigabytes in size. While that may seem like a large dataset of passwords, truly comprehensive password datasets are usually in the hundreds of gigabytes. Despite this, the limited datasets had

TABLE I. Popular Passwords

S/No	Password
1	123456
2	admin
3	12345678
4	123456789
5	1234
6	12345
7	password
8	123
9	Aa123456
10	1234567890

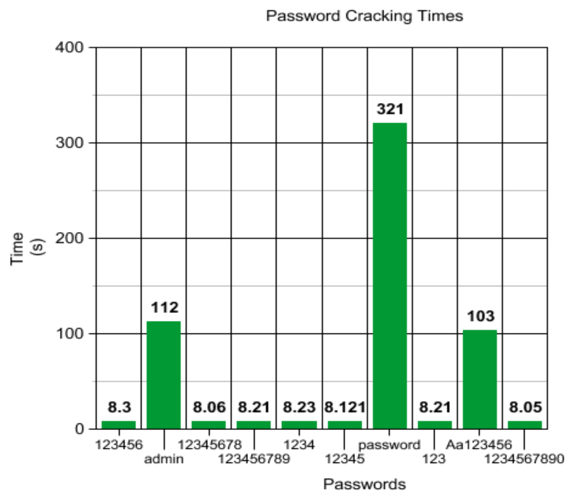

Fig. 5: Time taken to crack passwords using the dictionary attack

access to all the passwords in Table 1 and found them within a reasonable timeframe. Fig. 5 plots the password against the time that it took to crack them. This experiment illustrates the requirement of having strong passwords that are long and unpredictable. Malicious attackers have more complicated dictionaries and hardware that could crack said passwords even faster.

Having established that cracking popular passwords is a trivial affair with a low barrier to entry we examined how fast our newly proposed new salting algorithm is in comparison to the traditional method of randomly generating a salt.

Fig. 6 compares the time that it takes to calculate the salt in the traditional way which is when you randomly generate 64 random characters. From Fig. 6 we can see that the traditional salt generation method comes in at 0.1 seconds while the new salt generation method is at 0.12 seconds. In other words, the old method is faster by a factor of 10. However, it is important to note that there is one operation in the traditional method creates a bottleneck that takes up a significant amount of time. In the traditional method, the salt is only calculated once and then stored in a database. Thus, the actual time consumption in this regard is in the database access time or the database latency, which our approach avoids.

A challenge with database latency in online password authentication systems lies in the fact that many of the systems are hosted in one of the three major cloud providers – Amazon Web Services, Microsoft Azure, or Google Cloud Platform. The cloud providers and many of the companies that host their services in the cloud do not publish their statistics and there is very little research into database access times in the cloud. Even if this data was available, the different tiers of service often provided by cloud providers would have to be taken into consideration since higher tiers of hardware perform better and with lower latencies than other tiers. Database access times are also highly dependent on the hardware that is used in the data centers and how they are connected.

We chose to gauge the performance of the cloud databases through an alternative approach. Cloud providers usually put out press releases when they acquire a major new partner to their platform. In many cases, they provide data on how well their client is performing in their new data center. Although

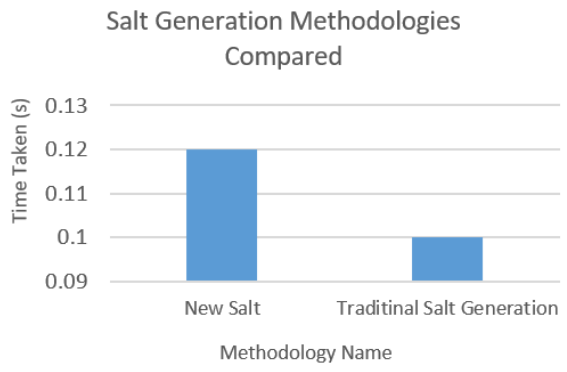

Fig. 6: Time taken to calculate salt with the respective methods

these press releases are for new cutting-edge and expensive databases that may probably not be used for storing password salts, they do often provide performance numbers for how the same workload performs on older hardware that might be running databases that store password salts. The purpose of this is to gain a high-level ballpark figure for database access times so that we can contextualize the performance of the new salt algorithm. A quote from Amazon Web Services' press release advertising their client, Near, stated that "After migrating to ElastiCache, Near saw close to four times faster read and write performance on its user profile and ID management services – reducing latency from 15 milliseconds to 4 milliseconds post migration." [14]. From this, we see that 15 milliseconds is what the database was performing before and now on the new and more expensive system, that is elasticache, which is used for mission-critical applications, it is down to 4 ms. We see a similar trend for the Google Cloud Platform. When Google Cloud Platform moved their client Snap onto their KeyDB system, the press release showed the database latency between the Google Cloud US Central Region 1 and the AWS US Region East data centers. According to their press release, "KeyDB, hosted in Google Cloud, caches frequently requested data to avoid repetitive cross-cloud calls and minimize latency. Before implementing KeyDB, the average P99 latency between Google Cloud us-central1 region and AWS us-east-1 region was between 49-133ms." [10].

This shows that there is a wide range between the access times of the databases and that it can fluctuate and vary a lot. If the variation in access times is significantly large, it can degrade the user experience to the point that the user might opt for a competing service. Our dynamic salt calculation method eliminates the need for database access. In addition, it is capable of using preexisting infrastructure to perform the calculations.

In Fig. 7, we factor in the database latency to evaluate how our dynamic salt calculation approach performs relative to using the traditional method on a cloud database.

V. Conclusion and Future Work

From Section IV and Fig. 7, we can conclude that the dynamic salt calculation methodology is faster than traditional approaches with database latency. The dynamic salt calculation approach allows us to have the low latency available on a fast database without the cost associated with it. In addition, it provides an added layer of security. This is due to the fact that we no longer have to store the salt values in a database and track the associations between the passwords and the salts. For the traditional method, even if the passwords and salts are encrypted, there is a non-zero chance it can get leaked as evidenced by the multiple password breaches that happen every year.

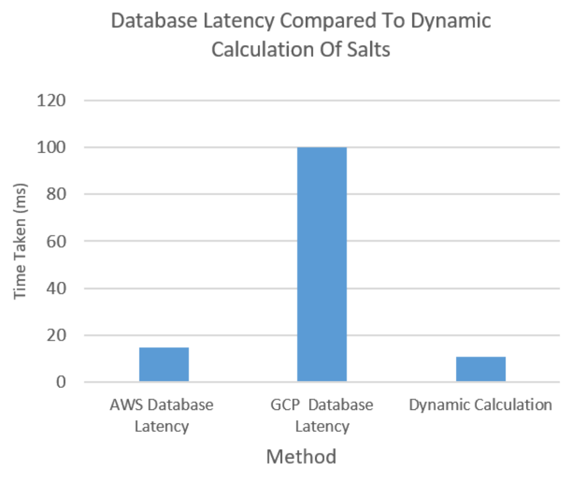

Fig. 7: Time taken to calculate salt with the respective methods

In terms of future work, an improvement that can be made to this algorithm is in the way in which the pointers to the random.txt file and the text corpus are calculated. As proof of concept, we chose single digits to calculate variables *start* and *end*. Choosing a range of values to generate the start and end values would make it even more difficult for any sort of attack to succeed. A range of values between 0 and 10000 would be enough to provide enough obfuscation for security purposes. We also intend to test the proposed algorithm on cloud databases in order to evaluate its performance relative to the traditional approach more efficiently in a cloud environment.

However, based on our experiments, we can conclude that the proposed method for dynamic calculation of password salts for improved resiliency against password cracking algorithms is a fast and efficient way of calculating password salts to prevent password attacks.

REFERENCES

[1] F. J. Corbato, M. Merwin-Daggett and R. C. Daley, "CTSS-the compatible time-sharing system," in IEEE Annals of the History of Computing, vol. 14, no. 1, pp. 31-54, 1992, doi: 10.1109/85.145324. J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892

[2] R. Morris and K. Thompson. 1979. Password security: a case history. Commun. ACM 22, 11 (Nov. 1979), 594–597. https://doi.org/10.1145/359168.359172

[3] W. Jansen and R. Ayers. (2004), Guidelines on PDA Forensics, Recommendations of the National Institute of Standards and Technology, Special Publication (NIST SP), National Institute of Standards and Technology, Gaithersburg, MD, [online], https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=150217

[4] P. Grassi, R. Perlner, E. Newton, A. Regenscheid, W. Burr, J. Richer, N. Lefkovitz, J. Danker, and M. Theofanos (2017), Digital Identity Guidelines: Authentication and Lifecycle Management [including updates as of 12- 01-2017], Special Publication (NIST SP), National Institute of Standards and Technology, Gaithersburg, MD, [online], https://doi.org/10.6028/NIST.SP.800-63b

[5] Canadian Centre for Cyber Security. Strategies for protecting web application systems against credential stuffing attacks (ITSP.30.035)

[6] K. Yang, X. Hu, Q. Zhang, J. Wei, and W. Liu, "A honeywords generation method based on deep learning and rule-based password attack", in Shi, W., Chen, X., Choo, KK.R. (eds) Security and Privacy in New Computing Environments. SPNCE 2021. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, vol 423. Springer, Cham. https://doi.org/10.1007/978-3-030-96791-8_22

[7] N. Ye, Y. Zhang, and C. M. Borror, "Robustness of the Markov-chain model for cyber-attack detection," in IEEE Transactions on Reliability, vol. 53, no. 1, pp. 116-123, March 2004, doi: 10.1109/TR.2004.823851.

[8] L. Zhang, T.Cheng, Y. Fei, "An Improved Rainbow Table Attack for Long Passwords, Procedia Computer Science", Vol. 107, 2017, Pages 47-52, ISSN 1877-0509, https://doi.org/10.1016/j.procs.2017.03.054.

[9] Free Rainbow Tables, Distributed Rainbow Table Project. Available at: https://freerainbowtables.com. Accessed: Feb. 2024.

[10] I. Zelker and I. McKee, " How Snap reduced latency by 96 percent with KeyDB database on Google Cloud", 2023. Available at: https://cloud.google.com/blog/products/application-modernization/snap-deploys-keydb-on-google-cloud-to-reduce-cross-cloud-latency

[11] Cloudfare. https://www.cloudflare.com/ . Accessed Feb. 2024.

[12] F. Franchetti, Y. Voronenko, M. Püschel, "A Rewriting System for the Vectorization of Signal Transforms", in High Performance Computing for Computational Science, M. Daydé, J. M. Palma, Á. L. Coutinho, E. Pacitti, J. C. Lopes (eds). VECPAR 2006. Lecture Notes in Computer Science, vol 4395. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-71351-7_28

[13] Grauer, Jared A.. "A Comparison of Three Random Number Generators for Aircraft Dynamic Modeling Applications." (2017).

[14] A. Nair and K. Kandasamy, "How Near reduced latency by four times and achieved 99.9% uptime by migrating to Amazon ElastiCache", 2021. Available at: https://aws.amazon.com/blogs/database/how-near-reduced-latency-by-four-times-and-achieved-99-9-uptime-by-migrating-to-amazon-elasticache/