

# Using LSH to hash passwords: A balance of security and user convenience

Nicholas Tung

## **Abstract**

The use of passwords in authentication systems is one of the most common methods of securing a system. Easily remembered passwords are also usually easily cracked, while difficult-to-remember passwords end up not being stored securely. This project is an attempt to develop a password protected system that provides security while also being more forgiving of user mistakes. This is accomplished using a locality-sensitive hash function. TLSH is a locality-sensitive hash function that can be used to determine similarity between strings by calculating the similarity of their hash digests (output of a hash function). It has been shown to have more accurate results compared to other locality-sensitive hashes. Using this similarity value, an incorrect password attempt can be categorized as a user end mistake or as a malicious attempt to gain access.

Results indicate that similarity scores from variations on the correct password can be used to generate score ranges to determine if a password is close to the correct one, or if it is drastically different. However, these ranges can occasionally be extremely large or extremely small. The testing used to determine ranges can yield inexplicably high scores for single character changes, but calculations of average range sizes show that, in general, using the scores generated, a reasonably sized bucket can be generated. As long as buckets are not too large, we can determine if a password is similar to the correct password, and penalize against the number of attempts accordingly. Although this system offers too much variability to be used in a real system, with some optimizations to how the ranges are generated, this can grow into a viable system that offers a usable system that is both secure and convenient.

## Introduction

Computer security is one of the prevailing problems of the 21st century. These systems are the primary barrier that protects information and controls many aspects of life. Maintaining information security requires a functional authentication system that ensures that information is only accessed by users with permission to access it. One of the most common methods of authentication of a user is by requiring a password (Halderman, J.A., 2005). A password unique to the user can be used to check that the person attempting to access the information is in fact the person they claim to be.

Use of passwords in computer systems is widespread, but faces some inherent security holes by itself. A simple but effective attack on a password-authenticated system is the brute-force attack ("McAfee Labs Threat", 2017). This attack simply attempts many random passwords until it guesses the correct one. There are various ways of preventing this attack from completing, one of which is by locking an account after a certain number of tries.

Another problem with passwords is inherent in people and the passwords they generate. While people may be concerned with maintaining security, the passwords they end up using are inevitably insecure. Some users say that they keep simple passwords so that they are less likely to incorrectly enter it, which can disrupt his/her workflow. Often, guidelines to secure password generation make passwords more difficult to remember and may be ignored. Many people use one base password and make modifications for each account (Inglesant, P. G., Sasse, M. A., 2010). A questionnaire on user password perceptions revealed that when restrictions on passwords are implemented, people will circumvent them to choose a more memorable password, one that is usually less secure (Adams, A., 1999).





				
Password	p4s5w3rdz	p4s5w3rdz	p4s5w3rdz	p4s5w3rdz
Salt	-	-	et52ed	ye5sf8
Hash	f4c31aa	f4c31aa	1vn49sa	z32i6t0

Figure 1: Table illustrating the effect of salts on password hash outputs. Two users with the same password can have different associated hashes because the salt that is appended before hashing changes the output of the hash function, resulting in different outputs for each user.

Passwords must also be securely stored as well, which raises an interesting problem of storing passwords such that a correct one can be easily confirmed, but not derived. One common solution is the use of hash functions. One-way hash functions can take a password and generate a unique output that is almost impossible to reverse. Hashing an attempted password and comparing it to the stored hash accomplishes the same thing as comparing an attempted password to a correct password, but never stores the user's plaintext password (Preneel, B., 2010). These types of password hashing schemes are recommended to be used in combination with salts. Salts are randomly generated strings, each assigned to a user. When the user's password is checked, the salt is appended and then hashed, ensuring that even if two users share a password, their salts will be different, resulting in a different hash output (Figure 1). Using hash functions combined with salts can mitigate the majority of precomputation attacks used against passwords (Scarfone, K., 2009).

Current widely adopted methods of authenticating users with passwords are unable to determine if the attempted password is close to the actual password. Due to the nature of the

cryptographic hash functions used, namely bcrypt, PBKDF2, and others, a small difference in the attempted password can result in a large difference in hash value. However, there exist hash algorithms that can determine the similarity between hashed strings. Called locality-sensitive hash functions, they attempt to maximize collisions given 2 sets of data. This means that similar datasets will have a similar hash value, and can be compared to determine similarity (Paulevé, L., 2010). Although traditionally used on larger datasets such as documents (Kumar, A. 2007), or for other types of problems like nearest neighbor searches (Slaney, M., 2008) or image searching (Wang, J., 2010), they can be used on smaller strings as well. An example of a locality-sensitive hash function is the TLSH hash, which has been shown to be more consistent than other locality-sensitive hash functions with a lower false positive rate. TLSH, which will be used in this study, analyzes strings by scanning parts of the string and mapping them into quartile buckets. Then, the output is generated based on the bucket counts (Oliver, J., 2013).

By using these types of hash functions, an attempted password can be compared to the correct password by comparing the hash values. This information can be used to change the number of remaining attempts if the password is too far from the correct password, or to not punish the user if the password is close to the correct password. The result would be a more flexible authentication system that can determine if the account is being broken into or not, and take action accordingly.

Error checking passwords to allow tolerance in authentication has been explored in past papers. These solutions use different methods of calculating string distance, then using it to different effects. A previous study, titled “Spelling-Error Tolerant, Order Independent Pass-Phrases via the Damerau-Levenshtein String-Edit Distance Metric” uses the

Damerau-Levenshtein String-Edit Distance Metric to calculate the level of error in a passphrase. However, this system requires the use of predetermined dictionary words as a passphrase, barring users from creating their own passwords (although some may see this as being worth the inconvenience in safety, as noted previously). To compare the passphrases, edit distances are calculated and compared to the dictionary versions (Bard, G., 2007). This use of the Damerau-Levenshtein String-Edit distance, which quantifies the edit difference between two strings as the number of operations to change one word into another (Damerau, F. J., 1964), allows for mistyped strings that are reasonably close to the correct spelling succeed in granting a user access to a passphrase-protected system.

Another example of tolerant password authentication is described in the patent titled “Techniques for entry of less-than-perfect-passwords”. The first of three claims describes a technique of scoring differences between passwords by organizing the strings into patterns, and calculating differences in said patterns. If differences in the password are within an acceptable maximum, the password is accepted. The other two claims describe methods of calculating comparative scores that compare two passwords. Scoring is defined by missing letters, extra letters, or letters shifted in position on a keyboard (Harris, S. C., 2008).

The approaches defined above are clever implementations of scoring distance between passwords. However, both rely on comparisons of strings stored, transmitted, and processed in plaintext. This means that using the proposed systems leaves passwords exposed, and should the storage system be breached, the passwords are revealed with no effort. Unlike the previously described salt and hash system, these systems relying on difference scoring of plaintext passwords offer no protection against potential system breaches. While an attacker that breaches

a system storing salted password hashes has obtained essentially unreadable data, breaching a system storing plaintext passwords yields instant access to any user's password.

Hashing passwords using locality sensitive hashing, in this case TLSH, gives the benefits of both systems. Only the hash digests of salted passwords are stored, and transmission and comparisons are only of the hash outputs, not the plaintext passwords. Interception of a password or a breach of the system storing the password does not immediately yield user passwords. This system is able to reap the benefits of tolerant password systems while still keeping stored passwords safe.

## **Methods**

All code used for this study is available on my github [here](#).  
(<https://github.com/st0p47/TLSHPasswordHash/>) in case the link is broken.

### **Generation of passwords**

Passwords used in this study were randomly generated by constructing a string of random characters. These characters were chosen from a set of characters. The set was constructed using built-in constants in the Python "string" module. The large character set was the concatenation of the built in character sets, with some concatenated multiple times. This was done to construct a character set that would resemble a realistic password, which is comprised mostly of alphabetic characters, with fewer numbers and special characters. Data collected from CMU students shows that with a mean password length of 10.49, 5.94 of those characters are lower case letters, 1.54 uppercase letters, 2.70 numbers and 1.39 symbols (Shay, R., 2010). For each position in a 10

character password, a random character from the predetermined character set described above was used to determine the resulting complete password.

### Variation of Passwords

Original: Password1!	Insertion	Substitution	Deletion	Capitalization Change
One Error	Padssword1!	Pa3sword1!	Pssword1!	PasswoRd1!
Two Errors	Pasesgword1!	Pas!worN1!	assword!	pasSword1!
Three Errors	aPasesword1!z	*lsswErd1!	Paswrd1	PASSworD1!

Figure 2: Table describing possible outcomes of the variation functions on an arbitrary password.

To have incorrect variations of passwords to generate difference scores for testing, each password had 9 variations generated, as well as a corresponding random password that was not similar to the original. The 12 variations were comprised of 4 types of string variations: insertion(s), substitution(s), deletion(s), and capitalization toggle(s). Each variation was performed to generate a password with a given variation 1, 2, and 3 times in a string. Figure 2 illustrates the effect of the different variations. Positions and new characters when necessary are chosen by random number generator.

### Hashing and Calculating Difference Scores

To test a realistic implementation of a system using TLSH to hash passwords, each password is assigned a pair of salts, one before the salt and one after. The password in between the two salts is multiplied 5 times to ensure that a given change has a significant effect, since the salts, which are constantly correct, do not outweigh errors in the password. This approach has a side effect of exemplifying the effect of a deletion or an insertion on the difference score.



```
stringToHash = prefixSalt + (attemptedPassword * 5) + suffixSalt
```

Each password and its variations are hashed, and the digests are stored. The TLSH python package has pre-written methods for hashing strings and calculating the difference score between two hashes. The difference scores are also stored for later data analysis.

```
diffScore = tlsh.diff(correctPasswordHash, attemptedPasswordHash)
```

## Data Analysis

Passwords that were generated are stored in a pandas dataframe. **pandas** is a new Python library of data structures and statistical tools initially developed for quantitative finance applications that has grown into a full-fledged library of data analysis tools (McKinney, W., 2010). For the dataset that I used, there were 9872 passwords of the original 10000 generated after removing invalid entries. Various attributes of the data, specifically the difference scores of a given variation, were analyzed and plotted using matplotlib, a 2D graphics package used for Python for application development, interactive scripting, and publication-quality image generation across user interfaces and operating systems (Hunter, J. D.).

## Results and Discussion

In this study, I aimed to develop a proof of concept for a password authentication system that utilized TLSH to allow leniency when logging into a password-protected account, while still maintaining good security practice and secure storage of passwords. In order to test this, I calculated and analyzed the difference scores of passwords with different numbers of different types of errors.

The difference scores generated by the TLSH difference calculation start at 0, and increase with increased differences. The lower a difference score, the more similarities the two passwords compared share.

### Pilot Studies and Proof of Viability

Before implementing this idea into a pseudo-system, proof that there was potential in this project was necessary. To ensure this, I hashed the passwords and salts as described above and plotted the average difference scores of the 3 variations of passwords with each error type. This was done to validate the efficacy of the hashing algorithm scaling scores appropriately with the number of errors (Figure 3).

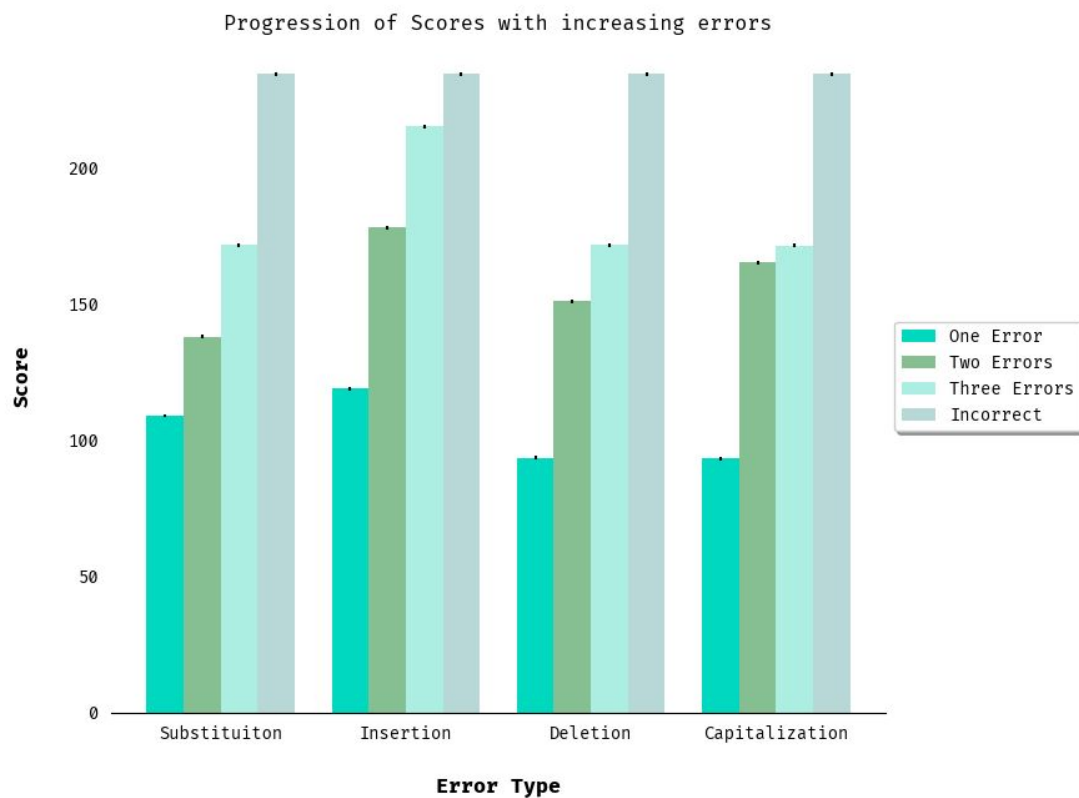


Figure 3: This data was obtained by taking 9872 randomly generated passwords, systematically generating variations on said passwords, and then taking the difference score between the correct password and the varied password. Each bar represents the average score in a given category of variations (substitution of characters, inserting characters, deleting characters, and changing capitalization of alphabetical characters) for a given number of errors. Each average score was taken from 9872 randomly generated variations on passwords in that category (ie. 9872 passwords with one, two and three substitution errors, 9872 passwords with one, two, and three insertion errors, etc.). Error bars represent standard error of the mean.

Figure 3 demonstrates that although the scoring suffers from inconsistencies, specifically in the score increase between passwords with two and three errors compared to the score increase between passwords with one and two errors, for the most part the score increases are steady with the change in errors. Although variation is present in the scores of each error type, there is no significantly different score among them. Because of variation in generated passwords and the scoring of the different error types, no conclusion can be drawn from the differences in scores of passwords with the same number of errors but different types of errors.

Another test to check viability is calculating the average change between passwords with different numbers of errors that share an error type. This analysis is represented in Figure 4.

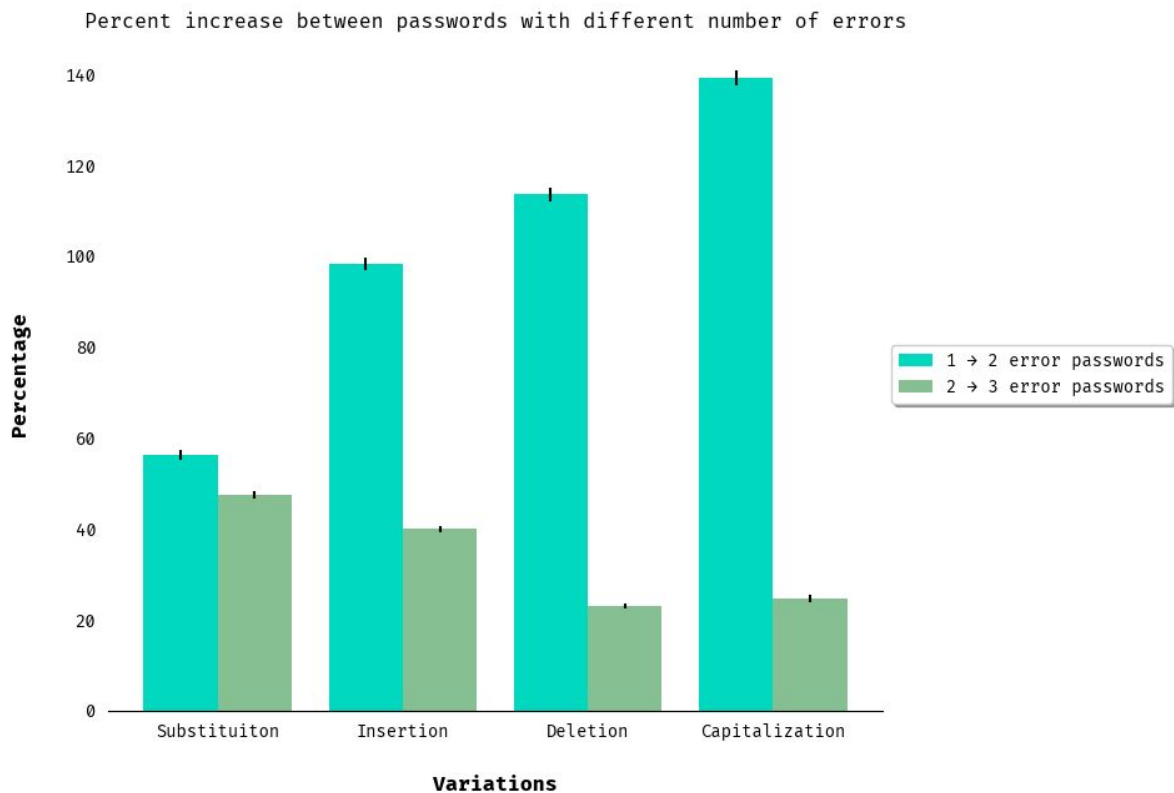


Figure 4: Average percent change between variations of the same password: single error to double error, and double error to triple error. Error bars represent standard error of the mean.

Figure 4 better illustrates that the average change between passwords sharing an error type but differing in the number of errors has wild variation among different error types. The changes in difference scores for passwords with substitution errors is the closest to ideal, where the change would be constant as the number of errors increases. On the other end of the spectrum, capitalization errors have a massive score increase when one error becomes 2 errors, but a significantly smaller change when two errors becomes three errors. An apparent issue of inconsistency is raised with this graph, since the scoring increases is not agnostic of error type.

### Generation of Password Score Bins for Categorization of Passwords

Since each password had 12 variations generated and scored against the original, categorization of an attempted password based on these pre-computed metrics is possible. This can account for the variation in scores between different passwords. By only categorizing passwords based on scores of variations of that password, there is no risk of a password with one error naturally yielding a high difference score being judged by score standards set for the majority of passwords.

When a user creates an account and the passwords associated with it, generating ranges of passwords to guess the level of error is possible by using data from known variations of a password. The score ranges to be used for password categorization is described in Figure 5.

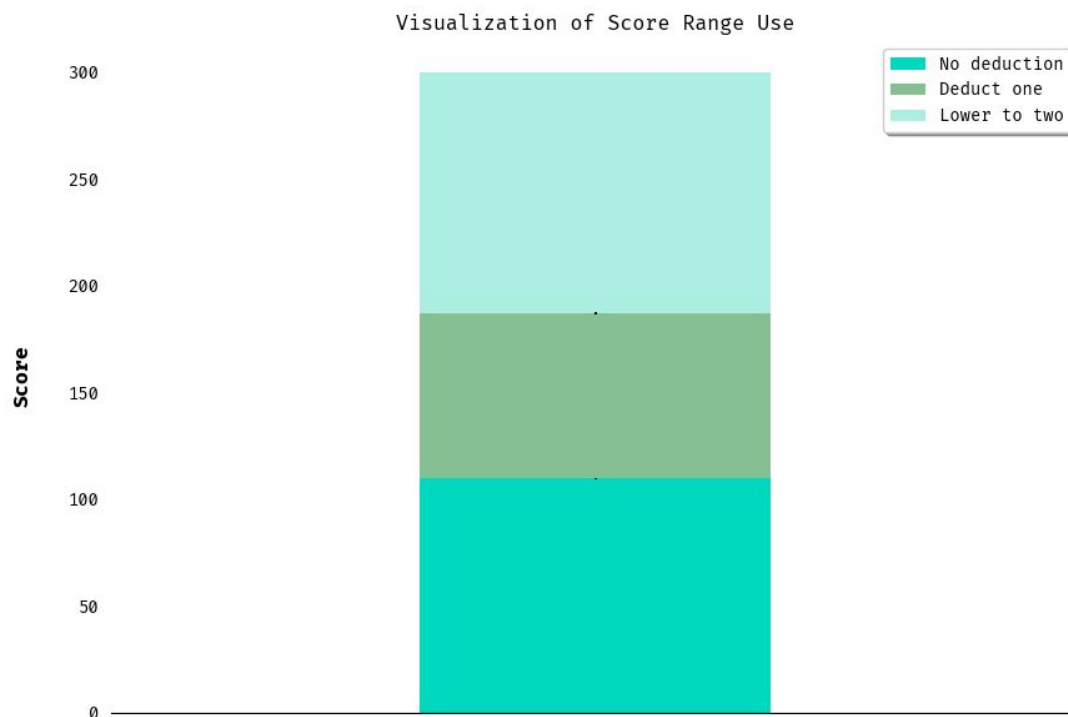


Figure 5: The size of each bar is the average range size for the generated passwords and their variations. The cutoffs were taken from the average thresholds of the 9872 passwords described above. Each bar represents the score range that triggers the labeled responses. The error bars represent standard error of the mean.

The ranges illustrated in Figure 5 will be used to categorize passwords to guess their origins. The thresholds are calculated by taking the average score of all passwords and adding the standard deviation of all passwords to get the upper boundary, and subtracting the standard deviation to get the lower boundary. Essentially, the middle bar is calculated, and the other 2 are filled in.

Ideally, this method of generating the “middle” password bin will successfully fit passwords with 2-3 errors. Passwords with 1 error will not be 0, but will fall into the “bottom” bar, signifying that it is very similar to the password. On the other hand, passwords with 3 or more errors will yield a difference score high enough to fall into the “highest” score range. Anything above the upper bound of the middle range, from 4 errors to completely wrong, will be considered to be too far from the correct password to simply be a mistake. This earns it a categorization as a malicious attempt at breaching the account, so the system will take immediate and severe action.

This method of score generation does have drawbacks. Generating score ranges with so few data points leaves a lot of room for variation in the lower bound, upper bound, and sizes. The average size of a calculated threshold (the difference between the upper and lower boundary of the threshold) is 77.3, but suffers from a very high standard deviation: 37.2. This means that there is a chance that the middle bin of a password will be extremely large or extremely small, both cases resulting in ineffective categorization of passwords.

The frequency of a given threshold size is illustrated with Figure 6.

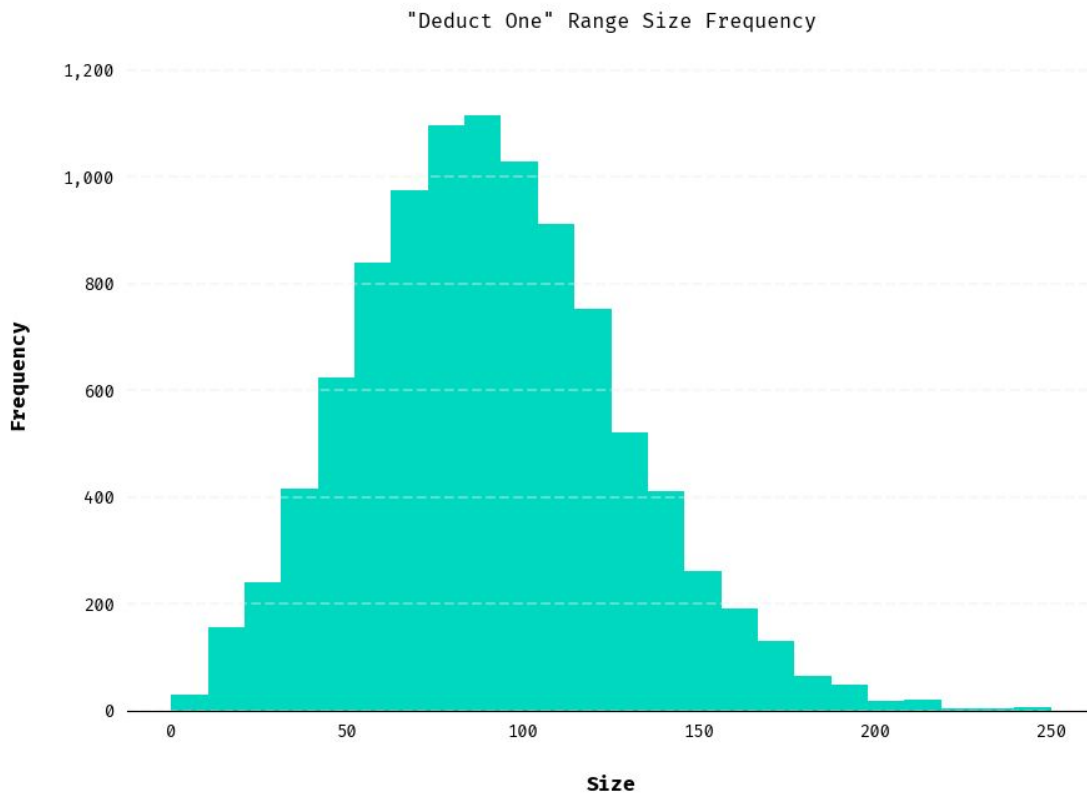


Figure 6: Histogram describing how many ranges fall in each bin size. Difference between upper and lower bounds was counted and tallied.

As shown by the histogram, the threshold sizes are normally distributed, with the majority of the sizes being between 70 and 100. The attributes of the set of threshold sizes shows promise for the proposed system, because of the fact that the sizes generated appear to be close to the average difference between one error and three error passwords. The average threshold size of 77.3 is fairly close to the average difference between one error and three error passwords of 90.7.

Overall, one error passwords have an average score of One error passwords have an average score of 103.8, two error passwords have an average score of 158.3 and three error passwords have an average score of 182.7. The average size of a generated threshold is 77.3,

close to the average difference score between one error and three error password of 90.7. As variation increases, TLSH successfully increases the difference score corresponding to the increase in variation. Through this preliminary testing, this system appears to be viable with further development.

## **Future Research**

This study has successfully shown that a system using TLSH to hash passwords is viable with further testing. An additional study would study a greater number of password variations, and a greater number of each type of variations. Due to constraints of time and computational resources, this study was limited to 12 variations, with 1 of each type. Ideally, more passwords of the same error type would be studied to increase the sample size when generating thresholds, and further eliminate outliers arising from anomalies in hashing.

Another method of testing would include field testing the proposed threshold system of categorizing passwords. Although the data shows that the threshold system could potentially work, it is still hypothetical. By repeatedly trying different variations on the correct password, the efficacy of the threshold system at categorizing passwords can be analyzed and tweaked to be more accurate.

## **Conclusion**

A TLSH-based password hashing system may be a viable solution to the problems that face users of password-authenticated systems today. It can be used to introduce tolerance in a system, allowing well-meaning users to make mistakes without seriously compromising security.



It maintains good security practice of storing salted hashes of passwords, not relying on dangerous plaintext storage. The users of such a system would have fewer worries of getting locked out of their account, and can create complex passwords knowing that if they mess up a few times, a lockout will not occur.

## **Acknowledgements**

Special thanks to Dr. Hersh for her invaluable guidance this year.

Thank you to Dr. Truglio and Ms. Spinelli for mentoring me throughout my time in research.

Thank you to the developers and maintainers of the SciPy scientific computing suite, for without their tools, this project would not be impossible.

## **Bibliography**

“McAfee Labs Threat Report,” <https://www.mcafee.com/ca/resources/rpquarterly-threats-sept-2017.pdf>, 2017

Adams, A., & Sasse, M. A. (1999). Users are not the enemy. *Communications of the ACM*, 42(12), 40-46.

Bard, G. (2007). Spelling-Error Tolerant, Order Independent Pass- Phrases via the Damerau-Levenshtein String-Edit Distance Metric, Fifth Australasian Symposium on ACSW Frontiers - Volume 68 (Ballarat, Australia, January 30 - February 02, 2007), 117-124.

Damerau, F. J. (1964). A technique for computer detection and correction of spelling errors. *Communications of the ACM*, 7(3), 171-176.

Halderman, J.A., B. Waters, and E.W. Felten. (2005). A Convenient Method for Securely Managing Passwords. In *Proceedings of 14th International World Wide Web Conference*, 2005.

Harris, S. C. (2008). U.S. Patent No. US7467403B2. Washington, DC: U.S. Patent and Trademark Office.

Hunter, J. D. (2007). Matplotlib: A 2D Graphics Environment, *Computing in Science & Engineering*, 9, 90-95.

- Kumar, A., Jawahar, C. V., & Manmatha, R. (2007). Efficient Search in Document Image Collections. *Computer Vision – ACCV 2007 Lecture Notes in Computer Science*, 586-595.
- McKinney, W. (2010). Data Structures for Statistical Computing in Python, *Proceedings of the 9th Python in Science Conference*, 51-56.
- Oliver, J., Cheng, C., & Chen, Y. (2013). TLSH -- A Locality Sensitive Hash. *2013 Fourth Cybercrime and Trustworthy Computing Workshop*.
- Paulevé, L., Jégou, H., & Amsaleg, L. (2010). Locality sensitive hashing: A comparison of hash function types and querying mechanisms. *Pattern Recognition Letters*, 31(11), 1348–1358.
- Scarfone, K., & Souppaya, M. (2009). Guide to Enterprise Password Management (Draft). NIST Special Publication 800-118 (Draft).
- Slaney, M., & Casey, M. (2008). Locality-Sensitive Hashing for Finding Nearest Neighbors [Lecture Notes]. *IEEE Signal Processing Magazine*, 25(2), 128–131.
- Shay, R., Komanduri, S., Kelley, P.G., Leon, P.G., Mazurek, M.L., Bauer, L., Christin, N., Cranor, L.F. (2010). Encountering stronger password requirements. *Proceedings of the Sixth Symposium on Usable Privacy and Security - SOUPS '10*.
- Wang, J., Kumar, S., & Chang, S. (2010). Semi-supervised hashing for scalable image retrieval. *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*.
- Inglesant, P. G., & Sasse, M. A. (2010). The true cost of unusable password policies. *Proceedings of the 28th International Conference on Human Factors in Computing Systems - CHI*