Analysis and Implementation of the Encrypted Negative Password Authentication Scheme

Terryn Brunelle, Winston Fee, and Peter Rowley

May 10, 2022

Abstract

In this paper we describe and implement the Encrypted Negative Password (ENP) password storage scheme outlined in a paper of Luo et. al., which proposes to securely hash passwords without the need of any externally stored randomness. We assess the security, practicality, performance, and scalability of ENP in comparison to other schemes, and discuss the practical results we obtain in a proof-of-concept implementation. We conclude that existing schemes already appear to provide an adequate level of security, but that ENP could be useful for application administrators worried about improper salting or other database maintenance issues.

Contents

1 Introduction

Secure password storage and authentication schemes are the backbone of the Internet. Any website which has user accounts should store its users' data as securely as possible, since any passwords which are broken can be used to log into users' accounts on other websites due to the commonality of password reuse. In addition, due to users choosing insecure passwords, simply hashing the given password is often not enough to prevent passwords from being discovered if an internal data table is stolen.

Due to the importance of this problem, there have been several schemes which have been both proposed and used over the years. Some examples of such schemes beyond just hashing the password are salting, peppering, and key stretching, which we cover in Section 2. However, these schemes all have their drawbacks, such as vulnerability to certain attacks in the case of salting and difficulty of use in the case of key stretching.

In this paper, we describe and implement the Encrypted Negative Password (ENP) password storage and authentication scheme of [1]. We analyze the security, performance, and scalability of the scheme in comparison to currently used methods such as salting, key stretching, and peppering. We conclude that it provides a competitive level of security without requiring outside randomness such as a salt or pepper. We describe the scheme in Section 3, perform our analysis in Section 4, and we discuss our implementation of it and our testing of the efficiency of our implementation in Section 5.

2 Background

In this section, we give some examples of common schemes employed today as well as some of their strengths and weaknesses.

2.1 Hashed Passwords

One of the most basic forms of security for storing passwords is to simply hide them by hashing them. The hashed value that is obtained from hashing the registered password is directly stored in the database, and every time a user logs in, their password is hashed to determine if it matches the stored value. While extremely simple, this scheme is highly vulnerable against lookup table attacks, in which an attacker has precomputed the hashes of several common passwords and searches for the corresponding hashes in an exposed or insecure database.

2.2 Salted Passwords

Salting is a method used to give more security to basic hashing. Before a password is hashed and stored in the database, salt is added by appending random bits to the beginning or end of the password. The larger the salt that is added, the more security is gained. While salting protects against precomputation attacks, it still has less than ideal security against dictionary attacks. In a dictionary attack, the attacker will try items in a "dictionary" (list of possible passwords) in a brute force fashion until the password is found. Salting helps against dictionary attacks, but more security is needed, as noted in [1].

2.3 Key Stretching

Key stretching is a method in which stored hidden passwords are made much longer and more random. An example would be applying different hashes multiple times to expand and randomize the key. The number of times that we perform the hash would be denoted as the cost factor for the key stretching algorithm, codifying how computationally expensive it is to calculate an overall hash for each password. While key stretching (when combined with salted passwords) is highly effective in defeating lookup table attacks as well as dictionary attacks, it introduces more parameterization for the implementer and it increases the computational load on the security system, as noted in [1].

2.4 Peppered Passwords

In a peppered password scheme, after hashing the password we encrypt the hash using a symmetric key, which we call the pepper, described in [3][2]. The pepper similar to a salt, but is shared across all stored passwords rather than being unique per password like a salt. The pepper is typically a long number (e.g., > 128 bits) such that brute-force guessing it would take a long time. Importantly, the pepper is not stored in the database, but is instead stored inside of a secret vault or hardware security module (similarly to how we might store a secret key used for encryption), as described further in [3].

2.5 Encrypted Negative Password (ENP)

Encrypted Negative Password is the password storage scheme that we are focusing on in this paper. Like key stretching, it gives high security against lookup table attacks and dictionary attacks. It also attempts to eliminate some of the implementation/usability complexity of key stretching by removing the need for salting and parameterization, as noted in [1].

3 ENP

The Encrypted Negative Password authentication scheme depends on three main concepts: hashing, symmetric-key encryption, and negative databases. The specific cryptographic hash function and encryption algorithm have to match in the sense that the output to the hash function should be the same size as a valid key for the encryption algorithm. This requirement comes from the fact that the hash of the password is used as the key for encryption, so that it is not necessary to store additional data to be used as a key.

The central concept of the ENP scheme is that of a negative database. As such, we will begin by describing negative databases, and then we will describe the steps of the ENP registration and verification algorithms.

3.1 Negative Databases

Let DB be any database consisting of some set of length-n bit strings. The central idea of a negative database is to hold the same information by instead storing the complement of the set of entries in DB. More precisely, if U is the set of all length-n bit strings, a negative database will store $U - DB$, the set of all length-n bit strings not in DB.

The immediate problem with this idea is that for a constant-size database DB, the corresponding negative database as stated above will have exponentially many entries, quickly making it infeasible to store. In order to make this concept feasible, a significant amount of compression is needed. In particular, entries in the negative database NDB are stored as length-n strings using the symbols 0, 1, and $*$, where a $*$ indicates that either a 0 or a 1 can be put in its place. For example, $1*0*$ represents all length-4 bit strings with a 1 as the first bit and a 0 as the third bit, of which there are 4.

For the ENP algorithm, we will focus on the specific case that DB contains exactly one entry. In this case, there are multiple efficient algorithms for finding a valid negative database. We will use two algorithms in this paper, the prefix algorithm and a slightly more complicated variant of it. The details of both algorithms will be described in Section 5. See Table 1 for an example of a negative database given by the prefix algorithm.

DB	NDB
0000	***1
	*1*0
	*010
	1000

Table 1: Negative database using prefix algorithm

3.2 Registration Algorithm

As with any authentication scheme, ENP has two algorithms: the registration algorithm and the verification algorithm. The registration algorithm takes as input a new user's desired username and password, storing the username and some data based on the password (for example, its hash value, in the simplest case) in the internal authentication data table. The verification algorithm takes as input a username and password, checks whether the username is in the table, and checks whether the password matches the corresponding entry.

The registration algorithm of ENP has three main steps, as follows.

1. Hash the password
2. Convert to a negative password
3. Encrypt the negative password

Figure 1: Steps of the Registration algorithm

The first step is simple: the password is hashed using the chosen cryptographic hash function, so that the output has a constant number of bits, say m.

In the second step, we construct a negative database for the one-entry database consisting of the hashed password. We call this a *negative password*. This is the crucial step, largely because the algorithms for generating a negative password from a hashed password introduce randomness. However, importantly, any possible negative password can be solved to yield the original hashed password by simply checking which bit string is not in the database. In particular, we do not need to know the randomness that was used for the reverse direction, which removes the need to store additional data in the table.

However, a negative password on its own is not more secure than just a hashed password, since as noted, any negative password can be solved to obtain the original hashed password. Therefore, an encryption step is necessary. In this step, we calculate $\text{Enc}(H(pass), NP)$, where $H(pass)$ is the hashed password used as the key, and NP is the negative password calculated in the second step. Note that it is important that the chosen encryption algorithm takes m-bit keys, since that is the length of $H(pass)$. The resulting output is the encrypted negative password, which gives the scheme its name, and is stored in the authentication data table.

3.3 Verification Algorithm

Given a username and password, the process of verifying the password consists of the following four steps.

1. Retrieve the ENP corresponding to the given username
2. Hash the password
3. Decrypt the ENP to get a negative password
4. Check that the hashed password is the solution to the negative password

Figure 2: Steps of the Verification algorithm

Steps 1 and 2 are independent of each other and can be done in either order. In step 1, the username is found in the authentication data table, and the corresponding ENP is retrieved. If there is no entry corresponding to the given username, then the process ends with an error message of the form "username not found." In step 2, the given password is hashed using the chosen cryptographic hash function.

In step 3, we calculate $\text{Dec}(H(pass), ENP)$. If the password is correct, then $H(pass)$ is the key used to encrypt the negative password, so this will output the negative password NP generated in the registration algorithm.

In step 4, we check that $H(pass)$ is the unique entry not contained in NP. The details of how this is done are dependent on the specifics of how the negative password is generated, which will be covered in Section 5. If it matches, we return that the password is correct; if it doesn't, we return that the password is incorrect.

4 Analysis

In this section we provide an analysis of the ENP scheme compared to several other common password storage schemes introduced in section 2. We consider both theoretical and practical security, performance, and scalability.

4.1 Theoretical Security

The authors of [1] evaluate the theoretical security of the scheme under lookup and dictionary attacks, and compares the security to what is achieved by hashed password, salted password, and key stretching schemes. Here, we provide a brief summary of the conclusions reached in the paper and add in a comparison to the peppered password scheme, as well as some of our own takeaways.

4.1.1 Lookup Table Attacks

In a lookup table attack, an adversary has access to the password authentication database and attempts to precompute a table mapping plaintext passwords to their

corresponding hash values. The adversary performs a search and compare between their precomputed table and the authentication database to see if they can determine any username, password pairs.

To perform a lookup table attack on an ENP scheme using m-bit password hashes, an adversary would need to compute all possible ENPs for every password it wishes to check for a given username (usually a list of most common passwords). For ENPI, the size of the lookup table would be be $O(N_d \cdot m!)$, where the $m!$ term comes from the fact that ENPI uses a random permutation to generate negative passwords, and N_d is the number of passwords the adversary wishes to precompute. This size increases quickly with the size m of the password hash, and is usually too big to be precomputed given the limits of modern storage resources. ENPII provides even more security, since the additional randomness is added to the negative password generation process.

A hashed password scheme is more easily attacked, given that we only need to store one hash for each password we wish to check. The salted password and key stretching analyses instead replace the $m!$ term with 2^l, where l is the size of the salt. The 2^l term comes from the adversary not knowing which salt goes with the passwords they wish to check, and thus needing to pre-compute hashes for every possible salt. This space bound is sufficient to be too large and expensive time-wise for large enough l. The paper notes, however, that since $m!$ grows faster than 2^l, ENPI and ENPII provide strictly more security against lookup table attacks than salted password and key stretching schemes.

For all of these schemes, an attacker could feasibly download lookup tables from external sources rather than generating one themselves. Peppering instead ensures that the lookup table needed to attack the authentication database is unique to that database, because the attacker will need to compute all possible peppers for every password that they wish to precompute. The space complexity would then be $O(N_d \cdot 2^p)$, where p is the bit-length of the pepper, as in [3]. Thus, the space complexity still grows slower than that of ENP. Even if we were to use both salting and peppering, the space complexity would be $O(N_d \cdot 2^{l+p})$, which still grows slower than $m!$.

In sum, we agree with the paper's findings that ENP provides more security against lookup table attacks than existing schemes. We note, however, that since an exponential space growth factor is sufficient to render precomputing a lookup table on modern machines, the factorial growth does not seem to add much practical security, at least in a modern context.

4.1.2 Dictionary Attacks

In a dictionary attack, an adversary tries to brute-force guess passwords corresponding to hashes in the authentication database. These attacks are impossible to prevent altogether, but can be made so time consuming that they become impractical to undertake.

To perform a dictionary attack against the ENP scheme, an attacker would hack into the authentication database and, for each ENP in the database and for each of the N_p passwords that the attacker wishes to check:

- Obtain a hash of the plaintext password (in time T_h)

- Decrypt the ENP to a negative password, using the password hash as a key (in time T_d)
- Check if the password hash is the solution of the decyprted negative password (in time T_s)

The time complexity of this process depends entirely on how long each of these steps take. Since the adversary repeats these steps for each of the N_p passwords in their password list to test against and for each of the N_d ENPs in the authentication table, the total time complexity of conducting a dictionary attack is $O(N_d \cdot N_p \cdot (T_h + T_d + T_s))$ [1].

Similarly, the time complexities to attack salted password and key stretching schemes depends on how long it takes to compute hashes and check for hash equality. Since we can control how long hashing takes in key stretching through the cost factor, dictionary attacks could in theory be made to take longer on key stretching schemes than on ENP schemes [1]. We note, however, that we can also add a cost factor and multiple rounds of encryption to ENP, meaning that the scheme can match the theoretical security of key stretching against dictionary attacks.

Peppered password schemes instead seem to introduce a significant advantage, rendering dictionary attacks practically impossible if the pepper is unknown. Without knowing the pepper, the attacker doesn't know what inputs to use to the hash function for the passwords in our testing list (unlike when the attacker has access to a salt). The attacker would need to brute-force the pepper when testing against each plaintext, hashed password pair, increasing the time complexity of the attack by a factor of 2^p (where p is the bit-length of the pepper) [3].

Thus, even though ENP is certainly competitive against schemes such as key stretching, there are other more widely studied schemes that already give superior theoretical security against dictionary attacks. In particular, a scheme combining both peppering and key stretching could serve as a powerful defense against lookup table and dictionary attacks. We also note that the security achieved by ENP over key stretching against dictionary attacks does not seem to be much, as both remain dependant on increasing load on computational resources via multi-iteration encryption.

4.2 Practical Security

Next, we assess what the security of ENP might look like in practice. Since there is no need for salt, there is no possibility for administrative error such as salt re-use or making salts too short. There is also no risk of publicly exposing a global secret that could compromise the entire system, as is a risk in peppered password schemes, noted in [3].

As discussed in section 4.1.2, peppered password schemes have the important benefit of forcing the attacker to brute-force the entire database for every possible pepper even after a database breach. If the attacker knows access to the salt and plaintext password of some user (including themselves), however, it is possible that they could brute-force the pepper. Once the pepper is obtained, the attacker would have significantly reduced the overall security of the system. It is important for this scenario that the passwords

are salted, as otherwise the security is reduced to that of a hashed password scheme, which is not much. Still, after a breach, it is non-trivial to change the pepper. Given the difficulty of key rotation and breach recovery, it is important to store a separate pepper for each application managed by an entity, as stated in [3].

In an ENP scheme, the attacker does not learn anything about the rest of the passwords in the database by obtaining a single username-password pair. This is due to the fact that negative passwords are generated from an (independent) random permutation for each password. Even if there is a database breach, one could simply recompute the negative passwords with fresh randomness for each compromised password. This seems to make the scheme much more practical to maintain than peppering.

The paper also highlights the benefits of not needing to store a salt for each password. The absence of a salt, however, does not seem to matter much for practical security, as the main benefit of storing a unique salt for each password is that compromising a username-password pair does not leak any information about other passwords in the system. This feature is already achieved by ENP, as noted above.

4.3 Performance

Like key stretching, as we noted earlier, the security of ENP against dictionary attacks depends on the computational price of authentication. Thus, a downside of ENP is that there exists a fundamental tradeoff between performance and security. Instead, peppered password schemes do not rely on the time to authenticate for security against dictionary attacks, introducing a significant benefit over ENP.

4.4 Scalability

A notable advantage of ENP is that not needing to store a salt with each password improves the scalability when compared to salting and key stretching. Thus, the scheme has the same space complexity as a simply hashed password table, but with significantly more security. Peppered schemes have similar space complexity, only requiring a single secret value to be stored. Since peppering is often combined with salting, however, in practicality ENP still exhibits more space benefits and thus scalability.

In practicality, the added space needed for salts likely is not a problem for modern machines. Therefore, even though ENP is theoretically more scalable than the other considered schemes, this is not necessarily a significant consideration.

4.5 Takeaways

ENP remains on par with existing widely used password storage schemes. Though it achieves less theoretical security against dictionary attacks when compared to peppered schemes, ENP has the benefit of leading to easier-to-maintain authentication databases. Compared to key stretching, however, ENP does not seem to offer significant theoretical advantage. Furthermore, we believe that existing schemes such as salting and peppering can be just as if not more secure than ENP, and that combinations of these schemes can achieve similar properties to ENP.

Still, ENP is an interesting password scheme proposal that is both secure and maintainable. The scheme could be a good choice for administrators worried about insecurely salting passwords or the maintainability of a peppering scheme, given that it requires neither salts nor a pepper to be stored and maintained.

5 Implementation

The ENP Paper suggests two alternative implementations for the scheme, which mainly vary in terms of how Negative Passwords are generated. We decided to implement both of them in Python to get a feel for the difficulty of implementing each scheme, as well as to compare the two schemes. These schemes are called ENPI and ENPII, respectively.

Our implementations can be found at `https://github.com/nyrret/857-project`.

5.0.1 General Implementation

Our implementations include two primary functions: Register and Login.

In Register, the password is first hashed by SHA128, and then a negative password is generated for the hash using either ENPI or ENPII. The negative password is then encrypted using AES and stored in a dictionary with the key being the user. For our proof-of-concept implementation, we don't use a real database, but just store everything locally.

In Login, the hash of the password is found and the encrypted negative password of the user is found in the database dictionary. The AES-encrypted negative password is decrypted and the hash of the input password is solved by checking if the hash actually matches the negative password.

5.0.2 ENPI

As noted above, the primary distinction between ENPI and ENPII lies in the specific algorithm that is used to generate a negative password given a hashed password. The algorithm that is used in ENPI is called the prefix algorithm; its steps are shown in Figure 3. Note that we zero-index into strings, unlike in [1].

The central process of the prefix algorithm is deterministic; the randomness comes from the fact that we choose a random permutation of the input bit string to perform it on, and then perform the inverse permutation before adding any entry to the database. In the central process, we organize our entries into the database by the first index at which a given bit string disagrees with the permuted hashed password. As such, the ith entry (zero-indexing, before inverse permutation) will agree with $\pi(H(pass))$ up to the $(i-1)$st position, and then it will have the opposite bit in the ith position, and have $*$s after that. It is clear that all bit strings which do not match $\pi(H(pass))$ fall into one of these categories.

Now we turn our attention the ENPII verification algorithm, and in particular, checking that a given hashed password matches a given negative password. While it would be possible to just check that the hashed password does not correspond to any of the entries in the database, the verification algorithm presented in [1] provides

Input: a length-m bit string S
Output: a negative password NP

- Randomly choose a permutation π of m elements
- For each i from 0 to $m-1$:
  - toAdd $= \pi(S)[0:i] + \neg\pi(S)[i] + \text{“}*\text{”} \times (m-i-1)$
  - add π^{-1}(toAdd) to NP
- Return NP

Figure 3: The prefix algorithm

additional security by checking that the given negative password is of the correct form for a negative password, as shown in Figure 4. Note that a "specified position" in an entry of NP is any position that is not a $*$.

Input: a length-m bit string S and a negative password NP
Output: true or false

- Check that the ith entry (zero-indexed) of NP has $i+1$ specified positions
- Initialize an empty length-m array x
- For each i from 0 to $m-1$:
  - Check the the ith entry np_i of NP has one remaining specified position, say at index j
  - Let $x[j]$ be the opposite of $np_i[j]$
  - Check that $np_k[j] = x[j]$ for each $k > i$, and then set each of these entries to $*$
- Check that x matches S bit-by-bit

Figure 4: Verifying a negative password from the prefix algorithm

Each of these steps is done because any output from the prefix algorithm should pass the corresponding test. For example, the ith entry in a negative password from the prefix algorithm has (before inverse permutation) the first i entries agreeing with $\pi(S)$, then the next entry disagreeing, and the rest are $*$; this totals $i+1$ specified entries.

The second part of the verification algorithm checks the specific format: the ith entry should add exactly one new specified position, and that position should be specified in all future entries. In the ith entry it should disagree with S, while in all future entries it should agree. This follows, again, directly from the way the prefix algorithm

is defined.

5.0.3 ENPII

The ENPII negative password generation algorithm also uses a random permutation, but it also adds additional randomness to the main part of the algorithm. The steps of this algorithm are displayed in Figure 5.

Input: a length-m bit string S
Output: a negative password NP

- Randomly choose a permutation π of m elements
- For each i from $m-1$ to 2:
  - Initiate toAdd= " $*$ " $\times m$ and let toAdd$[i] = \neg\pi(S)[i]$
  - Randomly choose distinct j, k from $[0, i-1]$
  - Let toAdd$[j] = \pi(S)[j]$ and toAdd$[k] = \pi(S)[k]$
  - Add π^{-1}(toAdd) to NP
- If index 1 differs from $\pi(S)$ but 0 doesn't:
  - Initiate toAdd= " $*$ " $\times m$ and let toAdd$[0] = \pi(S)[0]$ and toAdd$[1] = \neg\pi(S)[1]$
  - Randomly choose j from $[2, m-1]$
  - Let toAdd$[j] = 0$ and add π^{-1}(toAdd) to NP
  - Let toAdd$[j] = 1$ and add π^{-1}(toAdd) to NP
- If index 0 differs from $\pi(S)$:
  - Initiate toAdd= " $*$ " $\times m$ and let toAdd$[0] = \neg\pi(S)[0]$
  - Randomly choose distinct j, k from $[1, m-1]$
  - Let toAdd$[j] = 0$, toAdd$[k] = 0$ and add π^{-1}(toAdd) to NP
  - Let toAdd$[j] = 0$, toAdd$[k] = 1$ and add π^{-1}(toAdd) to NP
  - Randomly choose $k \neq j$ from $[1, m-1]$ again (j stays the same)
  - Let toAdd$[j] = 1$, toAdd$[k] = 0$ and add π^{-1}(toAdd) to NP
  - Let toAdd$[j] = 1$, toAdd$[k] = 1$ and add π^{-1}(toAdd) to NP
- Return NP

Figure 5: Variant of the prefix algorithm

This algorithm works because if both indices 0 and 1 of any given bit string agree with $\pi(S)$, then the first loop will produce at least one entry that matches that bit

string. If at least one of those indices disagrees, one of the second two bullets will produce an entry which matches it. Note that the negative password returned from this algorithm will have $m + 4$ entries.

Since the negative password generation algorithm in ENPII is more complicated, the corresponding verification algorithm is commensurately more involved. Its steps can be seen in Figure 6. The function Merge(np_j, np_k) puts a $*$ wherever np_j and np_k disagree, and puts the same symbol as both of them where they agree.

Input: a length-m bit string S and a negative password NP
Output: true or false

- Check that the size of NP is $m + 4$ and each entry has exactly three specified positions
- For each pair of indices $(j, k) = (m + 2, m + 3)$, $(m, m + 1)$, and $(m - 2, m - 1)$:
  - Check that np_j and np_k differ in exactly one place
  - Let $np_j = \text{Merge}(np_j, np_k)$
- Check that np_m and np_{m+2} differ in exactly one place
- Let $np_m = \text{Merge}(np_m, np_{m+2})$
- Initialize an empty length-m array x
- For each i from $k - 1$ to 0:
  - Check that there is only one remaining specified entry of np_i, say at index j
  - Let $x[j] = \neg np_i[j]$
  - Check that $np_k[j] = x[j]$ or $np_k[j] = *$ for each $k < i$, and then set each $np_k[j] = *$
- Check that x matches bit-by-bit with S

Figure 6: Verifying a negative password from the ENPII algorithm

This algorithm, as with the one in Figure 4, matches the form of the output of the corresponding negative password generation algorithm. It is clear from Figure 5 that the size of the negative password is $m + 4$ and that each entry has exactly three specified positions. In addition, it is not hard to deduce from the second to bullet points of Figure 5 that indices $(m+2, m+3)$, $(m, m+1)$, and $(m-2, m-1)$ each have one place where they differ, as well as $(m, m + 2)$ after merging. The justification of the loop (second to last bullet) above is similar to that of the analogous step in Figure 4; the only difference is that it is possible for the other elements to be $*$ instead of specified bits. We leave the details of verifying that this algorithm works to the reader.

5.0.4 Implementation Analysis and Testing

Each of ENPI and ENPII were relatively straightforward to implement. Storage is quite simple for ENP, as the only item that needs to be stored is the encrypted negative password. There are also no configurations needed for ENP (like key stretching might need) and all that is used are general hashing and encryption algorithms. The primary complexity in implementing ENP came from generating negative passwords, which requires non-trivial permutation algorithms. While non-trivial, we still found these both to be reasonable to implement; ENPI was, naturally, a bit simpler than ENPII.

We also tested speed of performance between ENPI and ENPII. Each of these generation and verification algorithms have a theoretical runtime of $O(m^2)$, where m is the size of the output of the cryptographic hash function in bits. Despite the theoretical result, we wanted to see if there were any empirical differences between the two. To test this, we timed how long it took to either register or login p different users in a row. The results are shown here in Figure 7, where p-values of 10 and 100 were used to run three tests for each process of each algorithm. Figure 7 shows the average runtimes of each set of three tests.

Test Results

	ENP1	ENP2
Register (x10)	1.2166 (sec)	1.2789 (sec)
Register (x100)	12.3562 (sec)	12.9246 (sec)
Login (x10)	0.0795 (sec)	0.0901 (sec)
Login (x100)	0.7951 (sec)	0.8977 (sec)

Figure 7: Code testing results

As can be derived from the table, ENPI was marginally fasater than ENPII, with a maximum improvement margin of $\approx 5\%$ for Registration and $\approx 12\%$ for Login.

In choosing which of ENPI or ENPII is better, there is little practical difference. While ENPII is slightly more difficult to implement and slightly slower in runtime than ENPI, both are simple enough to implement and the speed difference would make essentially no difference in practice. Theoretical differences between the two should be the main consideration in choosing which one to use as the practical differences are negligible, except perhaps for applications with a very high number of users that could be authenticating at the same time.

6 Conclusion

In this paper we provided an analysis and implementation of the ENP scheme, contributing an open-source proof of concept. In our analysis of the scheme, we conclude that it provides theoretical and practical security on par with existing widely used schemes. It does not seem to offer a significant edge over such schemes, but could be useful in scenarios where a database administrator is concerned about insecurely generating and/or storing external randomness. In future work, it could be good to evaluate how well ENP might pair as a layer with other schemes, and to conduct experiments on larger authentication databases.

7 Acknowledgements

We would like to thank Andres Fabrega for his help in scoping our project idea and encouragement towards are current direction. We are also grateful to Professors Rivest and Kalai for an engaging semester of 6.857 lectures and for striving to create the best possible student experience.

References

[1] Wenjian Luo, Yamin Hu, Hao Jiang, and Junteng Wang. Authentication by encrypted negative password. *IEEE Transactions on Information Forensics and Security*, 14(1):114–128, 2019. doi: 10.1109/TIFS.2018.2844854.

[2] Udi Manber. A simple scheme to make passwords based on one-way functions much harder to crack. *Computers & Security*, 15(2):171–176, 1996. ISSN 0167-4048. doi: https://doi.org/10.1016/0167-4048(96)00003-X. URL `https://www.sciencedirect.com/science/article/pii/016740489600003X`.

[3] Craig Webster. Securing passwords with salt, pepper and rainbows. *Barking Iguana*, 2009. URL `http://www.barkingiguana.com/2009/08/03/securing-passwords-with-salt-pepper-and-rainbows/`.