MULTISELECTION-BASED PSEUDO-HASH PASSWORD GENERATION

Roman Maksimovich
rmaksimovich@connect.ust.hk

March 26, 2024

We introduce a password generation algorithm which accepts a publicly known string (the "public key") as well as two large integer numbers ("choice private key" and "shuffle private key"), producing a random-looking string of characters (the "hash", used as a password), whose composition (i.e. number of lower-case and upper-case characters, etc.) can be set in advance. This is done by encoding list selections and permutations with positive integers, constructing the necessary list rearrangements based on the provided keys, and applying them to a predefined set of character lists (the "source configuration"). The algorithm is purely functional, which guarantees reproducibility and allows the user to generate secure passwords on the fly, without storing them in any location or having to remember them. Good levels of confusion and diffusion are achieved through so-called argument shift functions. We provide a primary implementation of the algorithm in Haskell, but implementations in C, JavaScript, and Kotlin are also available.

Keywords: Combinatorics, Security, Cryptography, Hashing, Functional Programming, Haskell

1 Introduction

The motivation behind the topic lies in the management of personal passwords. Nowadays, the average person requires tens of different passwords for different websites and services. Overall, we can distinguish between three ways of managing this set of passwords:

- **Keeping everything in one's head.** This is a method employed by many, yet it inevitably leads to certain risks. First of all, in order to fit the passwords in memory, one will probably make them similar to each other, or at least have them follow a simple pattern like "[shortened name of website]+[fixed phrase]". As a result, if even one password is guessed or leaked, it will be almost trivial to retrieve most of the others, following the pattern. Furthermore, the passwords themselves will tend to be memorable and connected to one's personal life, which will make them easier to guess.
- **Storing the passwords in a secure location.** Arguably, this is a better method, but there is a natural risk of this location being revealed, or of the passwords being lost, especially if they are stored physically on a piece of paper.
- **Using a password manager.** [1] These are software programs that create random passwords for the user and give access to them via a *master password.* This is a valid solution, especially since password managers tend to employ very secure encryption algorithms. However, this using such software implies problematic notions such as
  1. trusting the developers and administrators of the password manager, and
  2. storing passwords in a database (although encrypted).

In this paper we suggest a way of doing neither of these things. The user will not know the passwords or have any connection to them whatsoever, and at the same time the passwords will not be stored anywhere, physically or digitally. In this system, every password is a so-called "pseudo-hash" (since it does not technically adhere to the definition of a cryptographic hash, but has similar properties) produced by a fixed algorithm. The algorithm requires three inputs: one public key, i.e. the name of the website or service, and two private keys, which are arbitrary positive integers known only to the user (the initial version of the algorithm, finalized in Section 2.5, will only use one private key). Every time when requiring a password, the user will invoke the keys to re-create it from scratch. Therefore, in order to be reliable, the algorithm must be "pure", i.e. must always return the same output given the same input. Additionally, the algorithm should be published online for better accessibility, meaning that it must be robust enough so that, even if an attacker had full access to it and its mechanics, they would still not be able to guess the user's private keys or the passwords that they produce. These considerations naturally lead to exploring pure mathematical functions as pseudo-hashing algorithms and implementing them in a functional programming language such as Haskell.

2 The algorithm

There are many ways to generate pseudo-hash strings. In our case, these strings are potential passwords, meaning they should contain lower-case and upper-case letters, as well as numbers and special characters. Instead of somehow deriving such symbol sequences directly from the public and private keys, we will be creating the strings by selecting them from a pre-defined set of distinct elements (i.e. the English alphabet or the digits from 0 to 9) and rearranging them. The keys will play a role in determining the rearrangement scheme. With regard to this strategy, some preliminary definitions are in order.

2.1 Preliminary terminology and notation

Symbols A, B, C will denote arbitrary sets (unless specified otherwise). $\mathbb{N}_0$ is the set of all non-negative integers.

By E we will commonly denote a finite enumerated set of distinct elements, called a *source*. When multiple sources $E_0, E_1, ..., E_{N-1}$ are considered, we take none of them to share any elements between each other. In other words, their pairwise intersections will be assumed to be empty.

The symbol "#" will be used to denote the number of ways to make a combinatorial selection. For example, $\#^m(E)$ is the number of ways to choose m elements from a source E with significant order.

The expression $[A]$ will denote the set of all ordered lists composed from elements of the set A. We assume that all elements in a list are distinct. Every list can therefore be considered a source. The subset $[A]_m \subset [A]$ will include only the lists of length m. Extending this notation, we will define $[A_0, A_1, ..., A_{N-1}]$ as the set of lists $\alpha = [a_0, a_1, ..., a_{N-1}]$ of length N, where the first element is from A_0, the second from A_1, and so on, until the last one from A_{N-1}. Finally, if $\alpha \in [A]$ and $\beta \in [B]$, the list $\alpha \uplus \beta \in [A \cup B]$ will denote the concatenation of lists α and β.

Let α be a list. By $|\alpha|$ we will denote its length, while $\alpha : i$ will represent its i-th element, with the enumeration starting from $i = 0$. By contrast, the expression $\alpha \,!\, i$ will denote the list α without its i-th element. All sources are associated with the ordered list of all their elements, and so expressions like $E : i$ and $|E|$ also have meaning for a source E.

Let $k \in \mathbb{N}_0, n \in \mathbb{N}$. The numbers ${}^n k, {}_n k \in \mathbb{N}_0$ are defined to be such that $0 \leq {}^n k < n$ and ${}_n k \cdot n + {}^n k = k$. The number ${}^n k$ is the remainder after division by n, and ${}_n k$ is the result of division.

For a number $n \in \mathbb{N}$, the expression (n) will denote the semi-open integer interval from 0 to n, $(n) = \{0, ..., n-1\}$.

Let $n, m \in \mathbb{N},\ m \leq n$. The quantity $n!/(n-m)!$ will be called a *relative factorial* and denoted by $(n \mid m)!$.

Consider a function f of many arguments $a_0, a_2, ..., a_{n-1}$. Then with the expression $f(a_0, ..., a_{i-1}, -, a_{i+1}, ..., a_{n-1})$ we will denote the function of one argument a_i where all others are held constant.

2.2 Enumerating list selections

The defining feature of the public key is that it is either publicly known or at least very easy to guess. Therefore, it should play little role in actually encrypting the information stored in the private keys. It exists solely for the purpose of producing different passwords with the same private keys. So for now we will forget about it. In this and the following subsection we will focus on the method of mapping a (single, for now) private key $k \in \mathbb{N}_0$ to an ordered selection from a set of sources in an effective and reliable way.

Definition 2.1 (First-order choice function)**.** Let E be a source, $k \in \mathbb{N}_0$. The *choice function of order 1* is defined for E and k as the following one-element list:

$$\mathcal{C}^1(E, k) = \left[E : {}^{|E|}k\right].$$

It corresponds to picking one element from the source according to the key. For a fixed source E, the choice function is periodic with a period of $|E|$ and is injective on the interval $(|E|)$ with respect to k. Injectivity is a very important property for a pseudo-hash function, since it determines the number of keys that produce different outputs. When describing injectivity on intervals, the following definition proves useful:

Definition 2.2. Let A be a finite set and let $f \colon \mathbb{N}_0 \to A$ be a function. The *spread* of f is defined to be the largest number n such that, for all distinct $k_1, k_2 \in \mathbb{N}_0$, the following implication holds:

$$f(k_1) = f(k_2) \implies |k_1 - k_2| \geq n.$$

This number exists due the A being finite. We will denote the spread of f by $\operatorname{spr}(f)$.

Trivially, if $\operatorname{spr}(f) \geq n$, then f is injective on (n), but the converse is not always true. Therefore, a lower bound on the spread of a function serves as a guarantee of its injectivity on a certain interval. Furthermore, if $\operatorname{spr}(f) \geq n$ and f is bijective on (n), then f is periodic with period n and therefore has a spread of exactly n. We leave these facts as a simple exercise for the reader.

Proposition 2.3. *Let* $f\colon \mathbb{N}_0 \to A,\ g\colon \mathbb{N}_0 \to B$ *be two functions such that* $\operatorname{spr}(f) \geq n$ *and* $\operatorname{spr}(g) \geq m$. *Define the function* $h\colon \mathbb{N}_0 \to [A, B]$ *as follows:*

$$h(k) = [f({}^{n}k),\ g({}_{n}k + T({}^{n}k))],$$

where $T\colon \mathbb{N}_0 \to \mathbb{N}_0$ *is a fixed function, referred to as the* ***argument shift function.*** *It is then stated that* $\operatorname{spr}(h) \geq nm$.

Proof. Assume that $k_1 \neq k_2$ and $h(k_1) = h(k_2)$. Since h returns an ordered list, the equality of lists is equivalent to the equality of all their respective entries:

$$\begin{gathered} f({}^{n}k_1) = f({}^{n}k_2), \\ g({}_{n}k_1 + T({}^{n}k_1)) = g({}_{n}k_2 + T({}^{n}k_2)). \end{gathered}$$

Since f is injective on (n), we see that ${}^{n}k_1 = {}^{n}k_2$. Consequently, it follows from $k_1 \neq k_2$ that ${}_{n}k_1 \neq {}_{n}k_2$ and so ${}_{n}k_2 + T({}^{n}k_1) \neq {}_{n}k_2 + T({}^{n}k_2)$. We then utilize the definition of $\operatorname{spr}(g)$:

$$\begin{aligned} |\,{}_{n}k_1 + T({}^{n}k_1) - {}_{n}k_2 - T({}^{n}k_2)| &\geq m, \\ |\,{}_{n}k_1 - {}_{n}k_2| &\geq m, \\ \left| \frac{k_1 - {}^{n}k_1}{n} - \frac{k_2 - {}^{n}k_2}{n} \right| &\geq m, \\ \left| \frac{k_1 - k_1}{n} \right| &\geq m, \\ |k_1 - k_1| &\geq nm, \end{aligned}$$

q.e.d. ■

With this proposition at hand, we have a natural way of extending the definition of the choice function:

Definition 2.4. Let E be a source with cardinality $|E| = n$, and let $k \in \mathbb{N}_0,\ 2 \leq m \leq n$. The *choice function of order* m is defined for E and k recursively as follows:

$$\mathcal{C}^m(E, k) = [E : {}^{n}k] \uplus \mathcal{C}^{m-1}(E', k'),$$

where $E' = E \,!\, {}^{n}k$ and $k' = {}_{n}k + T({}^{n}k)$, while $T\colon \mathbb{N}_0 \to \mathbb{N}_0$ is a fixed argument shift function.

Proposition 2.5. *Let* E *be a source with cardinality* n. *Then the choice function* $\mathcal{C}^m(E, -)$ *of order* $m \leq n$, *as a function of k, has a spread of at least* $(n \mid m)!$.

Proof. We will conduct a proof by induction over m. In the base case, $m = 1$, we notice that $(n \mid m)! = n$, and the statement trivially follows from the definition of $\mathcal{C}^1(E, k)$.

Let us assume that the statement is proven for all choice functions of order $m - 1$. Under closer inspection it is clear that the definition of $\mathcal{C}^m(E, k)$ follows the scheme given in Proposition 2.3, with $\mathcal{C}^1(E, -)$ standing for f and $\mathcal{C}^m(E', -)$ standing for g. The application of the proposition is not entirely trivial, and we encourage the reader to consider the caveats. Thus, we can utilize the statement of the proposition as follows:

$$\operatorname{spr}(\mathcal{C}^m(E, -)) \geq \operatorname{spr}(\mathcal{C}^1(E, -)) \cdot \operatorname{spr}(\mathcal{C}^{m-1}(E, -)) \geq n \cdot (n - 1 \mid m - 1)! = (n \mid m)!,$$

q.e.d. ■

The above result is especially valuable considering the fact that there are exactly $(n \mid m)!$ ways to select an ordered sub-list of length m from a list of length n, meaning that $\mathcal{C}^m(E, -)$ is not only injective, but also surjective on the interval $((n \mid m)!)$. This makes it a bijection

$$\mathcal{C}^m(E, -)\colon ((n \mid m)!) \to [E]_m,$$

and therefore a periodic function with a spread of exactly $(n \mid m)! =: \#^m(E)$.

These properties make the choice function a fine candidate for a pseudo-hash map. Suppose that the source E is composed from lower-case and upper-case characters, as well as special symbols and digits:

$$E = \texttt{"qwertyuiopasdfghjklzxcvbnmQWERTYUIOPASDFGHJKLZXCVBNM0123456789!@\#\$\%"}$$

The choice function gives us a way to enumerate all possible ways to select a sub-list from E. What is more, these selections can be made more "random" and unpredictable by means of complicating the argument shift function T.

A reasonable practice is to set $T({}^{n}k)$ to the ASCII value of the character $E : {}^{n}k$. This way, each chosen character will influence the choice of the next, creating what is called a “chaotic system”, where its behavior is fully determined, but even small changes to inputs eventually produce large changes in the output. Here is a little input-output table for the choice function of order 10 with the specified source and shift function:

input	output
123	`41BeGs9$Dd`
124	`52NgJfZIk7`
125	`63MfHs9$Da`
126	`740VbDo6@u`
127	`851Br469$S`

There is, however, a serious problem. This selection method does not guarantee that the chosen 10 symbols will contain lower-case and upper-case characters, as well as digits and spacial symbols, all at the same time. Since the choice function is bijective, there is a key that produces the combination `"djaktpsnei"`, which will not be accepted as a password in many places, because it contains only one category of symbols.

2.3 Elevating the choice function

Definition 2.6. Let $\mathfrak{L}$ be list of pairs (E_i, m_i), where E_i are sources, $|E_i| = n_i,\ m_i \leq n_i$, for $i \in (N)$. The *elevated choice function* $\overline{\mathcal{C}}(\mathfrak{L}, -)$ corresponding to these data is defined for a key $k \in \mathbb{N}_0$ by means of the following recursion:

$$\overline{\mathcal{C}}(\mathfrak{L}, k) = [\mathcal{C}^{m_0}(E_0, {}^{n_0}k)] \uplus \overline{\mathcal{C}}(\mathfrak{L}\,!\,0, {}_{n_0}k + T({}^{n_0}k)),$$

where T is an argument shift function. The base of the recursion is given when $\mathfrak{L}$ is empty, in which case $\overline{\mathcal{C}}([], k) = []$. Otherwise, for every key k, its image under $\overline{\mathcal{C}}(\mathfrak{L}, -)$ is an element of

$$\operatorname{cod}\left(\overline{\mathcal{C}}(\mathfrak{L}, -)\right) = [[E_0]_{m_0}, [E_1]_{m_1}, \ldots, [E_N]_{m_N}]\,.$$

In this context, the list $\mathfrak{L}$ will be called a *source configuration.*

In other words, the elevated choice function is a “mapping” of the choice function over a list of sources, it selects a sub-list from every source and then composes the results in a list, which we call a *multiselection.* A trivial application of Proposition 2.3 shows that the spread of $\overline{\mathcal{C}}(\mathfrak{L}, -)$ is at least

$$\prod_{i=0}^{N-1} \operatorname{spr}(\mathcal{C}^{m_i}(E_i, -)) = \prod_{i=0}^{N-1} (n_i \mid m_i)!, \tag{1}$$

where E_i, n_i, and m_i compose the configuration $\mathfrak{L}$. In fact, due to the rule of product in combinatorics, we see that the expression in (1) directly corresponds to the number of possible multiselections from $\mathfrak{L}$, or $\#^{\overline{\mathcal{C}}}(\mathfrak{L})$ for short. Therefore, $\overline{\mathcal{C}}(\mathfrak{L}, -)$ is bijective on the interval $\left(\#^{\overline{\mathcal{C}}}(\mathfrak{L})\right)$ and periodic with period $\#^{\overline{\mathcal{C}}}(\mathfrak{L})$.

This solves the problem with lacking symbol categories — now we can separate upper-case letters, lower-case letters, numbers, etc., into different sources and apply the elevated choice function, specifying the number of symbols from each source. However, there are two issues arising:

- The result of the elevated choice function will be something like `"amwYXT28@!"`, which is not a bad password, but it would be nice to be able to shuffle the individual selections between each other instead of lining them up one after another.
- Despite the fact that the argument shift function makes the password selection chaotic, the function is a bijection, which means that it can be reversed. With sufficient knowledge of the algorithm, an attacker can write an inverse algorithm that retrieves the private key from the resulting password. This is a deal breaker for our function, because it defeats the purpose — you may as well have one password for everything. The way to solve this problem is to make the choice function artificially non-injective, or non-collision-free, in a controlled way. In such case, many different keys will produce the same password, and it will be impossible to know which one of them is the correct one. This violates the common non-collision property of hash functions, but it is necessary given the nature of the function we are developing.

We will solve one problem at a time.

2.4 The merge function

Proposition 2.7. *Let $f\colon A \times \mathbb{N}_0 \to B$ and $g\colon B \times \mathbb{N}_0 \to C$ be functions such that $\operatorname{spr}(f) \geq n$ and $\operatorname{spr}(g) \geq m$, where the spread is taken with respect to the second argument. Assume also that g is* ***absolutely injective*** *with respect to* *the first argument, that is,*

$$\forall (b_1,k_1),(b_2,k_2) \in B \times \mathbb{N}_0, \;\; g(b_1,k_1) = g(b_2,k_2) \implies b_1 = b_2.$$

Define the function $h\colon A \times \mathbb{N}_0 \to C$ *by*

$$h(a,k) = g(f(a, {}^{n}k), {}_{n}k + T({}^{n}k)),$$

where T *is an argument shift function. It is then stated that* $\operatorname{spr}(h) \geq nm$ *with respect to* k*.*

Proof. Take an element $a \in A$ and let k_1, k_2 be distinct numbers such that $h(a,k_1) = h(a,k_2)$. It means that

$$g(f(a, {}^{n}k_1), {}_{n}k_1 + T({}^{n}k_1)) = g(f(a, {}^{n}k_2), {}_{n}k_2 + T({}^{n}k_2)).$$

Since g is absolutely injective, we see that $f(a, {}^{n}k_1) = f(a, {}^{n}k_2)$, which means that ${}^{n}k_1 = {}^{n}k_2$, since f is injective on the interval (n). Now, since the first argument of g in the above equation is the same, we can use the definition of spread for the function g:

$$\begin{aligned} |\,{}_{n}k_1 + T({}^{n}k_1) - {}_{n}k_2 - T({}^{n}k_2)| &\geq m, \\ |\,{}_{n}k_1 - {}_{n}k_2| &\geq m, \\ \left|\frac{k_1 - {}^{n}k_1}{n} - \frac{k_2 - {}^{n}k_2}{n}\right| &\geq m, \\ \left|\frac{k_1 - k_2}{n}\right| &\geq m, \\ |k_1 - k_2| &\geq nm, \end{aligned}$$

q.e.d. ■

Definition 2.8. Let E_1, E_2 be two sources, m_1, m_2 be numbers such that $m_i \leq |E_i|$ for $i = 1,2$. Define the *merge function of order 2*,

$$\mathcal{M}^2\colon [E_1]_{m_1} \times [E_2]_{m_2} \times \mathbb{N}_0 \to [E_1 \cup E_2]_{m_1+m_2},$$

with the following recursive procedure: for $\alpha \in [E_1]_{m_1}, \; \beta \in [E_2]_{m_2}, \; k \in \mathbb{N}_0$, consider two cases:

1. Either α or β is empty, that is, $m_1 = 0$ or $m_2 = 0$. Then we set $\mathcal{M}^2(\alpha,\beta,k)$ to $\alpha \uplus \beta$.
2. Neither α nor β is empty. Then we will assume that the merge function is already refined for $(\alpha\,!\,0, \beta, -)$ and $(\alpha, \beta\,!\,0, -)$. Let s_1 be the spread of the function $\mathcal{M}^2(\alpha\,!\,0,\beta,-)$ and s_2 be the spread of $\mathcal{M}^2(\alpha,\beta\,!\,0,-)$. Finally, denote the remainder ${}^{(s_1+s_2)}k$ by k'. The merge of α and β with key k and an argument shift function T is defined as
$$\mathcal{M}^2(\alpha,\beta,k) = \begin{cases} [\alpha : 0] \uplus \mathcal{M}^2(\alpha\,!\,0, \beta, k' + T(k')), & \text{if } k' < s_1, \\ [\beta : 0] \uplus \mathcal{M}^2(\alpha, \beta\,!\,0, k' + T(k')), & \text{otherwise.} \end{cases}$$

The merge function takes two lists and combines them together in one, in such a way that the order of elements in each of the two lists is not disturbed. For example, the merge of $[1,2,3]$ and $[a,b,c]$ with a certain key could be $[1,a,b,2,c,3]$. We will now derive some properties of $\mathcal{M}^2$. If s_1 and s_2 are as defined above, we immediately see that $\mathcal{M}^2(\alpha,\beta,-)$ is periodic with period $s_1 + s_2$, since it depends only on $k' = {}^{(s_1+s_2)}k$. Moreover, it is clear from the definition that $\mathcal{M}^2(\alpha,\beta,-)$ is injective on the interval (s_1+s_2), which means that its spread equals exactly s_1+s_2:

$$\operatorname{spr}(\mathcal{M}^2(\alpha,\beta,-)) = \operatorname{spr}(\mathcal{M}^2(\alpha\,!\,0,\beta,-)) + \operatorname{spr}(\mathcal{M}^2(\alpha,\beta\,!\,0,-)). \tag{2}$$

Proposition 2.9. *Let* α *and* β *be lists from* $[E_1]_{m_1}$ *and* $[E_2]_{m_2}$ *respectively. Then the spread of the corresponding merge function* $\mathcal{M}^2(\alpha,\beta,-)$ *with respect to the key* k*, is equal to* $(m_1+m_2)!/(m_1! \cdot m_2!)$*.*

Proof. We will conduct a proof by induction over the sum $m_1 + m_2$. The base case is provided by the situation when either m_1 or m_2 is zero. Now, assume that $m_1, m_2 \neq 0$ and for all similar pairs with sum $m_1 + m_2 - 1$ the statement is proven. Recall (2), which we can now transform due to the induction hypothesis:

$$\begin{aligned} \operatorname{spr}(\mathcal{M}^2(\alpha,\beta,-)) &= \frac{((m_1-1)+m_2)!}{(m-1)! \cdot m_2!} + \frac{(m_1+(m_2-1))!}{m_1! \cdot (m_2-1)!} = \\ &= \frac{m_1 \cdot (m_1+m_2-1)! + m_2 \cdot (m_1+m_2-1)!}{m_1! \cdot m_2!} = \frac{(m_1+m_2)!}{m_1! \cdot m_2!}, \end{aligned}$$

q.e.d. ■

Using basic combinatorics, we can see that the number $(m_1 + m_2)!/(m_1! \cdot m_2!)$ corresponds to the number of all possible ways to merge the lists α and β, which we will denote by $\#^{\mathcal{M}}(\alpha, \beta)$. It means that the function $\mathcal{M}^2(\alpha, \beta, -)$ is in fact surjective, and therefore bijective, on the interval from zero to its spread. Now we will expand its definition to an arbitrary number of lists.

Definition 2.10. Let $\overline{\alpha} = [\alpha_0, \alpha_1, ..., \alpha_{N-1}$ be a list of lists, where $N \geq 2$ and $\alpha_i \in [E_i]_{m_i}$. Define the *merge function of order N* recursively as follows:

$$\mathcal{M}^N(\overline{\alpha}, k) = \begin{cases} \mathcal{M}^2(\alpha_0, \alpha_1, k), \quad N = 2, \\ \mathcal{M}^2(\alpha_0, \mathcal{M}^{N-1}(\overline{\alpha} \,!\, 0, {}^s k), {}_s k + T({}^s k)), \quad N > 2, \end{cases} \tag{3}$$

where s is the spread of the function $\mathcal{M}^{N-1}(\overline{\alpha} \,!\, 0, -)$, and T is an argument shift function.

Proposition 2.11. *Let $\overline{\alpha} = [\alpha_0, \alpha_1, ..., \alpha_N]$ be a list of lists with $\alpha_i \in [E_i]_{m_i}$. Then we have*

$$\operatorname{spr}(\mathcal{M}^N(\overline{\alpha}, -)) = \frac{\left(\sum_{i=0}^{N-1} m_i\right)}{\prod_{i=0}^{N-1} m_i!}. \tag{4}$$

Proof. We conduct a proof by induction over N. If $N = 2$, the result follows from Proposition 2.9. Now, assume that the result holds for all merge functions of order $N - 1$. Note that (3) follows the definition scheme assumed in Proposition 2.7, and by applying its result, we see that (4) holds for $\mathcal{M}^N(\overline{\alpha}, -)$. ■

2.5 Merging the multiselection

Definition 2.12. Let $\mathfrak{L}$ be a source configuration of pairs (E_i, m_i) for $i \in (N)$, and let k be a non-negative integer. We define the *merged choice function* corresponding to $\mathfrak{L}$, as follows:

$$\mathcal{MC}(\mathfrak{L}, k) = \mathcal{M}^N\left(\overline{\mathcal{C}}(\mathfrak{L}, {}^n k),\ {}_n k + T({}^n k)\right),$$

where n is the spread of the elevated choice function $\overline{\mathcal{C}}(\mathfrak{L}, -)$, while T is a fixed argument shift function.

In other words, the merged choice function selects N lists from the configuration via $\overline{\mathcal{C}}$, and then merges them together using $\mathcal{M}^N$. We have therefore solved our first problem — the password resulting from this new pseudo-hash function will have different categories of characters mixed together. What is more important, there is no information lost in the process, which is illustrated by the following proposition:

Proposition 2.13. *Let $\mathfrak{L}$ be a source configuration of length N. The following lower bound takes place for the spread of the merged choice function:*

$$\operatorname{spr}(\mathcal{MC}(\mathfrak{L}, -)) \geq \operatorname{spr}(\overline{\mathcal{C}}(\mathfrak{L}, -)) \cdot \operatorname{spr}(\mathcal{M}^N(\overline{\alpha}, -)) = \prod_{i=0}^{N-1} (n_i \mid m_i)! \cdot \frac{\left(\sum_{i=0}^{N-1} m_i\right)!}{\prod_{i=0}^{N-1} m_i!}, \tag{5}$$

where $\overline{\alpha}$ is the multiselection arising from the application of $\overline{\mathcal{C}}$, while n_i are the lengths of the sources E_i in the configuration $\mathfrak{L}$.

Proof. Since the merge function $\mathcal{M}^N$ preserves the order of the lists it merges, we can see that it is absolutely injective with respect to its first argument, $\overline{\alpha}$. Indeed, no matter how the multiselection $\overline{\alpha}$ is merged, all elements of, say, E_0, can be read from the resulting merged list in the order that they were originally. In other words, the list $\alpha_0 \in \overline{\alpha}$ can be reconstructed from the output of $\mathcal{M}^N(\overline{\alpha}, k)$ for all k. And so can α_1, α_2, etc. Here we are actively using the fact that all elements across all sources $E_0, E_1, ..., E_{N-1}$ are distinct. Now that the absolute injectivity of M^N has been established, the statement of the immediately follows from an application of Proposition 2.7. ■

It can once again be shown using combinatorics, that the final expression in 5 is, in fact, the total number of ways to choose a multiselection from $\mathfrak{L}$ and merge it (the reader should not hesitate to check this). Therefore, we conclude that the function $\mathcal{M}(\mathfrak{L}, -)$ is bijective on the interval from zero to its spread, and therefore periodic with the period of its spread.

2.6 Subverting invertibility

We still have one problem left: our current pseudo-hash function is bijective, and it can be inverted relatively easily to retrieve the private key from the source configuration and the final pseudo-hash. To prevent this, we will have to make our function a bit less injective — artificially add inputs that produce the same output, in order to make the function harder to invert.

Definition 2.14. Let $\mathfrak{L}$ be a source configuration consisting of pairs (E_i, m_i) for $i \in (N)$, with $n_i = |E_i|$. Let $k_1, k_2 \in \mathbb{N}_0$ be two numbers referred to as *choice key* and *shuffle key*. Define the *hash* corresponding to these inputs as follows:

$$\mathcal{H}(\mathfrak{L}, k_1, k_2) = \mathcal{C}^{\sum m_i}(\mathcal{MC}(\mathfrak{L}, k_1), k_2).$$

This definition can be re-written in a more concise way by defining the *shuffle function* $\mathcal{S}(\alpha, k)$ for a list α as $\mathcal{C}^{|\alpha|}(\alpha, k)$, and letting the source configuration be the varying argument:

$$\mathcal{H}(-, k_1, k_2) = \mathcal{S}(-, k_2) \circ \mathcal{MC}(-, k_1).$$

This pseudo-hash function makes a multiselection from every source in the configuration, then merges them together with the merge function, and finally reshuffles the resulting list.

We call $\mathcal{H}(\mathfrak{L}, -, -)$ a "pseudo-hash" here because it does not adhere to the formal definition of a cryptographic hash. Still, such naming is somewhat justified, given that $\mathcal{H}$ is a mapping that is impossible to invert, with a uniform distribution of outputs (given a uniform distribution of keys). The reason $\mathcal{H}$ is not a proper hash function is the fact that for each "hash" it produces, one may easily construct key pairs that result in this hash. However, there is no way to pinpoint the pair that was actually used. We will now discuss the properties of $\mathcal{H}(\mathfrak{L}, -, -)$ for a given source configuration $\mathfrak{L}$:

- **Injectivity.** For any $k_2 \in \mathbb{N}_0$, the function $\mathcal{H}(\mathfrak{L}, -, k_2)$ is injective on the interval from zero up to

  $$\#^{\mathcal{MC}}(\mathfrak{L}) := \operatorname{spr}(\mathcal{MC}(\mathfrak{L}, -)) = \prod_{i=0}^{N-1} (n_i \mid m_i)! \cdot \frac{\left(\sum_{i=0}^{N-1} m_i\right)!}{\prod_{i=0}^{N-1} m_i!}.$$

  This is because $\mathcal{MC}(\mathfrak{L}, -)$ is injective on this interval, and $\mathcal{S}(-, k_2)$ is a bijection. With respect to the shuffle key k_2, the function $\mathcal{H}(\mathfrak{L}, k_1, -)$ is injective on the spread of $\mathcal{S}$, which is

  $$\#^{\mathcal{S}}(\mathfrak{L}) := \operatorname{spr}(\mathcal{S}(\alpha, -)) = \#^{|\alpha|}(\alpha) = (|\alpha| \mid |\alpha|)! = |\alpha|! = \left(\sum_{i=0}^{N-1} m_i\right)!,$$

  where $\alpha = \mathcal{MC}(\mathfrak{L}, k_1)$. Therefore, the number of relevant key pairs for the pseudo-hash function, denoted by $\#^{(k_1,k_2)}(\mathfrak{L})$, is

  $$\#^{(k_1,k_2)}(\mathfrak{L}) = \#^{\mathcal{S}}(\mathfrak{L}) \cdot \#^{\mathcal{MC}}(\mathfrak{L}) = \prod_{i=0}^{N-1} (n_i \mid m_i)! \cdot \left(\left(\sum_{i=0}^{N-1} m_i\right)!\right)^2 \cdot \left(\prod_{i=0}^{N-1} m_i!\right)^{-1}. \tag{6}$$

- **Collisions.** However, the number in (6) does not equal the number of all possible values of $\mathcal{H}$. When applying $\mathcal{S}$ after $\mathcal{MC}$, we are changing the order of elements in each source twice. That is, the information about the order of these elements, stored in the output of $\overline{\mathcal{C}}$, is lost after this output is reshuffled with $\mathcal{S}$. The number $\#^{\mathcal{H}}(\mathfrak{L})$ of all possible outputs of $\mathcal{H}(\mathfrak{L}, -, -)$ is the number of ways of choosing m_i elements from E_i (unordered), multiplied by the number of ways to reorder them as one list. We therefore recognize that

  $$\#^{\mathcal{H}}(\mathfrak{L}) = \prod_{i=0}^{N-1} \binom{n_i}{m_i} \cdot \left(\sum_{i=0}^{N-1} m_i\right)!.$$

  Now, due to the fact that all resulting hashes are equally likely for k_1, k_2 within their respective injectivity intervals, we can calculate the number of different (k_1, k_2) pairs that produce the same hash in a fixed configuration, $\#^{\cap}(\mathfrak{L})$:

  $$\#^{\cap}(\mathfrak{L}) = \frac{\#^{(k_1,k_2)}(\mathfrak{L})}{\#^{\mathcal{H}}(\mathfrak{L})} = \frac{\prod_{i=0}^{N-1} (n_i \mid m_i)! \cdot \left(\left(\sum_{i=0}^{N-1} m_i\right)!\right)^2 \cdot \left(\prod_{i=0}^{N-1} m_i!\right)^{-1}}{\prod_{i=0}^{N-1} \binom{n_i}{m_i} \cdot \left(\sum_{i=0}^{N-1} m_i\right)!} = \left(\sum_{i=0}^{N-1} m_i\right)!.$$

With respect to each of the two keys, $\mathcal{H}$ is injective, but in combination they clash together and, in a sense, encrypt each other, erasing the trace to the original pair of keys. It does mean that the pseudo-hash function now requires two private keys instead of one, but it is a minor disadvantage.

2.7 Producing different hashes with the same private keys

The premise of this entire discussion was that one requires many passwords for different purposes. Right now, the only way is to create a new primary-secondary key pair for every new occasion. Considering how large these numbers tend to be, you may as well try to remember the hashes themselves. This is where the public key comes into play. The public key, denoted p, is an integer defined by the particular context in which the algorithm is invoked. We will assume it to be the name of the website for which we require a password, like "google", for example, converted into a number by treating the characters in the string as digits in base-128. This number acts on the choice private key by shifting it modulo $\#^{\mathcal{MC}}(\mathfrak{L})$:

$$k_1' = {}^{\#^{\mathcal{MC}}(\mathfrak{L})}(k_1 + p)$$

and then this new choice key is plugged into the pseudo-hash function along with the shuffle key. Due to the injectivity of $\mathcal{H}$ with respect to k_1, we see that different public keys produce different output hashes, as long as they remain in the interval $(\#^{\mathcal{MC}}(\mathfrak{L}))$. Its influence on k_1 is simple and predictable, but it doesn't have to be complex, since the public key is not directly responsible for any encryption. Instead it is designed to be easily remembered.

2.8 Security properties

In this subsection, we will use a specific source configuration $\mathfrak{L}$, particularly the following:

i	E_i	n_i	m_i
0	"ckapzfitqdxnwehrolmbyvsujg"	26	8
1	"RQLIANBKJYVWPTEMCZSFDOGUHX"	26	8
2	"=!*@?$%#&-+^"	12	5
3	"1952074386"	10	4

Therefore, every password produced with this configuration will contain a total of 25 symbols, 8, 8, 5 and 4 from their respective categories.

Now, suppose that an attacker with perfect knowledge of the pseudo-hashing algorithm and a supercomputer that can perform 10^{10} password checks per second, decides to crack the passwords produced with the above source configuration and a fixed choice-shuffle key pair (k_1, k_2).

First of all, they might try to brute-force a given password by iterating over all 25-letter strings made of 8 lower-case letters, 8 upper-case letters, 5 special characters, and 4 digits. They would have to check $\#^{\mathcal{H}}(\mathfrak{L})$ combinations, which in our case is

$$\#^{\mathcal{H}}(\mathfrak{L}) = 6,296,585,738,425,733,189,152,569,035,980,800,000,000,000 > 2^{142}.$$

Finding the correct string would take the supercomputer about

$$9,976,338,090,389,847,753,239,424 > 10^{24}$$

years, or about

$$723,604,706,635,950 > 10^{14}$$

ages of the Universe. Here we calculate the time it would take to find the correct entry with at least 50% confidence, that is, the time to traverse half of the space of possible values.

By examining the algorithm, the attacker might try to brute-force the choice-shuffle key pair instead of the final pseudo-hash. The number of all relevant key pairs for our configuration is

$$\begin{aligned}\#^{(k_1,k_2)}(\mathfrak{L}) = {}& 97{,}667,663,944,563,885,279,694,955,400,689,334,\\ & 503,676,436,493,107,200,000,000,000,000,000 > 2^{225}.\end{aligned}$$

There is no possibility of traversing this number of values.

Now let us assume that one of the passwords generated with $\mathfrak{L}$ and (k_1, k_2) has been compromised, for example due to a security leak on a website it was used to log into. Denote this leaked password by h. The attacker knows that there are exactly $\#^{\cap}(\mathfrak{L})$ different key pairs that generate the pseudo-hash h. They can generate any number of these key pairs at leisure (by substituting the intermediate layer α of their choosing). However, even if a key pair (k_1', k_2') generates h, it need not generate any other pseudo-hash that was generated by the original key pair (k_1, k_2). Therefore, the attacker still needs to identify the original key pair among the potential ones, of which there is

$$\#^{\cap}(\mathfrak{L}) = 15,511,210,043,330,985,984,000,000 > 2^{83}.$$

Traversing half of this number of key pairs would take about

$$24,576,029 > 10^{7}$$

years. It is, of course, almost infinitely better than brute-forcing the final pseudo-hash (it won't even take a single age of the Universe), but it is still pretty much impossible. However, if this is an issue, one can always use a password with, say, 30 symbols instead of 25, and its security properties will be considerably stronger.

3 The implementation

To implement the $\mathcal{H}(\mathfrak{L}, -, -)$ function, we have chosen the Haskell programming language. This choice is natural because the pseudo-hash algorithm does not involve any mutation and hence fits perfectly into the functional paradigm. Additionally, implementing it in a statically typed functional language ensures a degree of reliability due to a powerful type system and the inability to mutate values. Haskell also has a mature package ecosystem and first-class support for arbitrary-precision integers, which is necessary for computing the spreads of the functions involved in our algorithm.

3.1 The `pshash` program

`pshash` (from "pseudo-hash") is the name of the implementation of the algorithm specified in the previous section. The main program has no GUI, it is only a command-line utility. This decision was made for the sake of simplicity, speed, and portability. The Haskell source is available at `https://github.com/thornoar/pshash`, together with installation instructions and other details.

The CLI accepts three arguments: the public key and two private keys, returning a hash using its builtin source configuration. On demand, the user may pass a different configuration using one of the `-k`, `-n`, and `-c` command-line options, but this is not recommended, since it will have to be done every time. Instead, it is suggested to download the source, change the default configuration in the code, and then compile it to a new executable. Alternatively, one can write a configuration file and invoke `pshash` with the `--impure` flag to read it. You can run `pshash --help` to see all available options and their behavior.

3.2 Other implementations

The pseudo-hash algorithm is implemented in other languages:

- **C** for the in-development desktop app, source at `https://github.com/thornoar/pshash-gui`;
- **JavaScript** for the web interface, available at `https://thornoar.github.io/pshash-web/app/`;
- **Kotlin** for the Android application, source at `https://github.com/thornoar/pshash-app`, with download links hosted at `https://thornoar.github.io/pshash-web/get/`.

For more information, see the respective GitHub repositories.

References

[1] S. Oesch, S. Ruoti, J. Simmons, and A. Gautam. "It basically started using me:" An observational study of password manager usage. 2022. In *Proceedings of the 40th ACM CHI Conference on Human Factors in Computing Systems*. Available at `https://doi.org/10.1145/3491102.3517534`.