On rearrangement hashing with Haskell

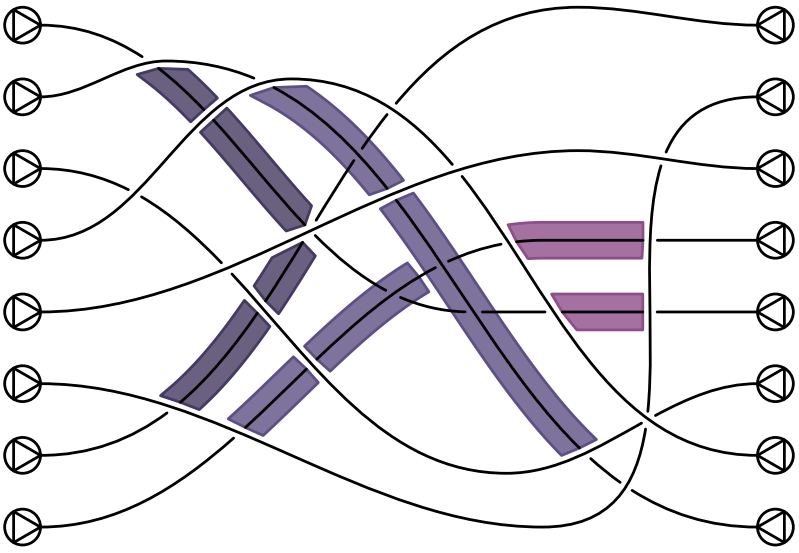
Roman Maksimovich

Keywords

Combinatorics
Hash
Security
Haskell
Functional programming

Abstract

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”), 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 pure, 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. We provide a primary implementation of the algorithm in Haskell, but implementations in C, Javascript, and Kotlin are also available.



March 26, 2024

Contents

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 two 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. There is, after all, a limit to one's imagination.
- **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. Currently, various "password managers" are available, which are software programs that will create and store your passwords for you. This is a good solution, but it comes with the necessity to trust the password manager not to have security vulnerabilities, not to mention that the password manager itself will have access to one's passwords, and a party inside the team developing the password manager will be able to get ahold of them.

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 pseudo-hash (since it does not adhere to the definition of a cryptographic hash, but the idea is similar) 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 retrieving 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 must be robust enough so that, even if a hacker had full access to it and its mechanics, they would still not be able to guess the user's private key or the passwords that it produces. 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 theory

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 understand 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 pair-wise intersections will be assumed to be empty.

The symbol "#" will be used to describe 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 the 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 be the concatenation of lists α and β.

Let α be a list. $|\alpha|$ 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 thus expressions such as $E : i$ and $|E|$ 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 $0 \leqslant {}^n k < n$ 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 represent the semi-open integer interval from 0 to n: $(n) = \{0, 1, ..., n-1\}$.

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

Consider a function f of many arguments a_0, a_1, ..., 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 2.2.1. (First-order choice function): Let E be a source, $k \in \mathbb{N}_0$. The *choice function of order 1* is defined 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 2.2.2. Let A be a finite set and let $f : \mathbb{N}_0 \to A$ be a function. The *spread* of f is defined to be the largest number n such that, for all $k_1, k_2 \in \mathbb{N}_0$, $k_1 \neq k_2$, the following implication holds:

$$f(k_1) = f(k_2) \Longrightarrow |k_1 - k_2| \geqslant n.$$

This number exists due to A being finite. We will denote this number by $\mathrm{spr}(f)$.

Trivially, if $\mathrm{spr}(f) \geqslant n$, 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. Furthermore, if $\mathrm{spr}(f) \geqslant n$ and f is bijective on (n), then f is periodic with period n and therefore has a spread of exactly n. We leave this as a simple exercise for the reader.

Proposition 2.2.1. *Let $f : \mathbb{N}_0 \to A$, $g : \mathbb{N}_0 \to B$ be functions such that $\mathrm{spr}(f) \geqslant n$ and $\mathrm{spr}(g) \geqslant m$. Define the function $h : \mathbb{N}_0 \to [A, B]$ as follows:*

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

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

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 corresponding elements:

$$\begin{aligned} f({}^n k_1) &= f({}^n k_2), \\ g({}_n k_1 + T({}^n k_1)) &= g({}_n k_2 + T({}^n k_2)). \end{aligned}$$

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 ${}_n k_1 + T({}^n k_1) \neq {}_n k_2 + T({}^n k_2)$. We then utilize the definition of $\mathrm{spr}(g)$:

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

q.e.d. ∎

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

Definition 2.2.3. Let E be a source with cardinality $|E| = n$, $k \in \mathbb{N}_0$, $2 \leqslant m \leqslant n$. The *choice function of order* m is defined recursively as

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

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

Proposition 2.2.2. *Let E be a source with cardinality n. Then the choice function $\mathcal{C}^m(E, k)$ of order $m \leqslant n$, as a function of k, has a spread of at least $(n \,|\, m)!$.*

Proof: We will conduct a proof by induction over m. In the base case, $m = 1$, we notice that $(n \,|\, m)! = n$, 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 2.2.1, with $\mathcal{C}^1(E, k)$ standing for f and $\mathcal{C}^{m-1}(E', k')$ standing for g. The application of the proposition is not straightforward, and we encourage the reader to consider the caveats. Thus, we can utilize the statement of the proposition as follows:

$$\begin{aligned} \operatorname{spr}(\mathcal{C}^m(E, k)) &\geqslant \operatorname{spr}(\mathcal{C}^1(E, -)) \cdot \operatorname{spr}(\mathcal{C}^{m-1}(E', -)) \geqslant \\ &\geqslant n \cdot ((n-1) \,|\, (m-1))! = (n \,|\, m)!, \end{aligned}$$

q.e.d. ∎

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

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

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

These properties make the choice function a fine candidate for a pseudo-hash mapping. Suppose that the source E is composed from lower-case and upper-case Latin 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. Fortunately, there is a solution.

2.3. Elevating the choice function

Definition 2.3.1. Let $\mathfrak{L}$ be be a list of pairs (E_i, m_i), where E_i are sources, $|E_i| = n_i$, $m_i \leqslant n_i$, for $i \in (N)$. The *elevated choice function* 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 is an element of

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

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 2.2.1 shows that the spread of $\overline{\mathcal{C}}(\mathfrak{L}, -)$ is at least

$$\prod_{i=0}^{N-1} \operatorname{spr}(\mathcal{C}^{m_i}(E_i, -)) = \prod_{i=0}^{N-1} (n_i \,|\, m_i)!, \tag{2.1}$$

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 (2.1) 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 $\big(\#^{\overline{\mathcal{C}}}(\mathfrak{L})\big)$ 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, a hacker 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 2.4.1. *Let $f : A \times \mathbb{N}_0 \to B$ and $g : B \times \mathbb{N}_0 \to C$ be functions such that $\operatorname{spr}(f) \geqslant n$ and $\operatorname{spr}(g) \geqslant m$, 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,*

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

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

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

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

Proof: Let an element $a \in A$ and let k_1, k_2 be distinct numbers such that $h(a, k_1) = h(a, k_2)$. That implies,

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

Since g is absolutely injective, we see that $f(a, k_1) = f(a, k_2)$, which means that ${}^n k_1 = {}^n k_2$, since f is injective on (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:

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

q.e.d. ∎

Definition 2.4.1. Let E_1, E_2 be two sources, m_1, m_2 be numbers such that $m_1 \leqslant |E_1|$ and $m_2 \leqslant |E_2|$. Define the *merge function of order 2,*

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

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 set $\mathcal{M}^2(\alpha, \beta, k)$ to be equal to $\alpha \uplus \beta$.
2. Neither α nor β is empty. Then we will assume that the merge function is already defined 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

$$\mathcal{M}^2(\alpha, \beta, k) = \begin{cases} [\alpha : 0] \uplus \mathcal{M}^2(\alpha \,!\, 0,\ \beta,\ k' + T(k')), & k' < s_1 \\ [\beta : 0] \uplus \mathcal{M}^2(\alpha,\ \beta \,!\, 0,\ k' + T(k')), & \text{otherwise,} \end{cases}$$

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 what they are in the above definition, 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 is equal exactly to $s_1 + s_2$:

$$\operatorname{spr}(\mathcal{M}^2(\alpha, \beta, -)) = \operatorname{spr}(\mathcal{M}^2(\alpha \,!\, 0, \beta, -)) + \operatorname{spr}(\mathcal{M}^2(\alpha, \beta \,!\, 0, -)). \tag{2.2}$$

Proposition 2.4.2. *Let α and β be elements of $[E_1]_{m_1}$ and $[E_2]_{m_2}$ respectively. Then the spread of the corresponding merge function, with respect to the key k, is equal to $\frac{(m_1+m_2)!}{m_1! \cdot m_2!}$.*

Proof: We will conduct a proof by induction over the sum $m_1 + m_2$. The base case is provided by the situation where 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 (2.2), which we can now transform due to the induction hypothesis:

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

$$= \frac{m_1 \cdot (m_1 + m_2 - 1)! + m_2 \cdot (m_1 + m_2 - 1)!}{m_1! \cdot m_2!} = \frac{(m_1 + m_2)!}{m_1! \cdot m_2!},$$

q.e.d. ∎

Using some combinatorial logic, we can see that the number $\frac{(m_1+m_2)!}{m_1! \cdot m_2!}$ 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, as usual, expand its definition beyond only two lists.

Definition 2.4.2. Let $\overline{\alpha} = [\alpha_0, \alpha_1, ..., \alpha_{N-1}]$ be a list of lists, where $N \geqslant 2$ and $a_i \in [E_i]_{m_i}$. Define the *merge function of order* N recursively as follows:

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

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

Exercise 2.4.1. Using induction and Proposition 2.4.1, prove the following result for the spread of the merge function:

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

where $\overline{\alpha} = [\alpha_0, \alpha_1, ..., \alpha_{N-1}] \in \Big[[E_0]_{m_0}, [E_1]_{m_1}, ..., [E_{N-1}]_{m_{N-1}}\Big]$.

2.5. Merging the multiselection

Definition 2.5.1. 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:

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

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

$$\operatorname{spr}(\mathcal{MC}(\mathfrak{L}, -)) \geqslant \operatorname{spr}\Big(\overline{\mathcal{C}}(\mathfrak{L}, -)\Big) \cdot \operatorname{spr}(\mathcal{M}^N(\overline{\alpha}, -)) = \prod_{i=0}^{N-1} (n_i \,|\, m_i)! \cdot \frac{\left(\sum_{i=0}^{N-1} m_i\right)!}{\prod_{i=0}^{N-1} m_i!}, \qquad (2.3)$$

where $\overline{\alpha}$ is the multiselection arising from the application of $\overline{\mathcal{C}}$, while n_i are the lengths of 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 present statement immediately follows from an application of Proposition 2.4.1. ■

It can once again be shown using combinatorics, that the final expression in (2.3) 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 $\mathcal{MC}(\mathfrak{L}, -)$ is bijective on the interval from zero to its spread, and therefore periodic with the period of its spread.

2.6. Double encryption

However, we still have one problem left: our current pseudo-hash function is bijective, and it can be reverse-engineered relatively easily to retrieve the private key from 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 2.6.1. Let $\mathfrak{L}$ be a source configuration consisting of pairs (E_i, m_i) for $i \in (N)$, where $n_i = |E_i|$. Let $k_1, k_2 \in \mathbb{N}_0$ be two numbers, referred to as the *primary (or choice) key* and the *secondary (or 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 readable 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 $\mathfrak{L}$ 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.

Note that the term "hash" is used loosely here, as it may not adhere to the formal definition of a cryptographic hash. Still, such naming is somewhat justified, given that $\mathcal{H}$ is designed to be a uniformly distributed encryption mapping that is very hard to invert. We will now discuss the properties of $\mathcal{H}(\mathfrak{L}, k_1, k_2)$ for a given source configuration $\mathfrak{L}$:

- **Injectivity.** $\mathcal{H}$ is injective with respect to the choice key k_1 on the interval from zero up to

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

  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 pseudo-hash function is injective on the spread of $\mathcal{S}$, which is

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

  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:

$$\#^{(k_1,k_2)} = \#^{\mathcal{S}}(\mathfrak{L}) \cdot \#^{\mathcal{MC}}(\mathfrak{L}) = \prod_{i=0}^{N-1} (n_i \,|\, m_i)! \cdot \left(\left(\sum_{i=0}^{N-1} m_i\right)!\right)^2 \cdot \left(\prod_{i=0}^{N-1} m_i!\right)^{-1}. \tag{2.4}$$

- **Collisions.** However, the number in (2.4) 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 of all possible outputs of $\mathcal{H}$ 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, $\#^{\cap}(\mathfrak{L})$:

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

$$= \left(\sum_{i=0}^{N-1} m_i\right)!$$

With respect to each of the two keys, $\mathcal{H}$ is an injective function, but in combination they clash together and, to a degree, 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.

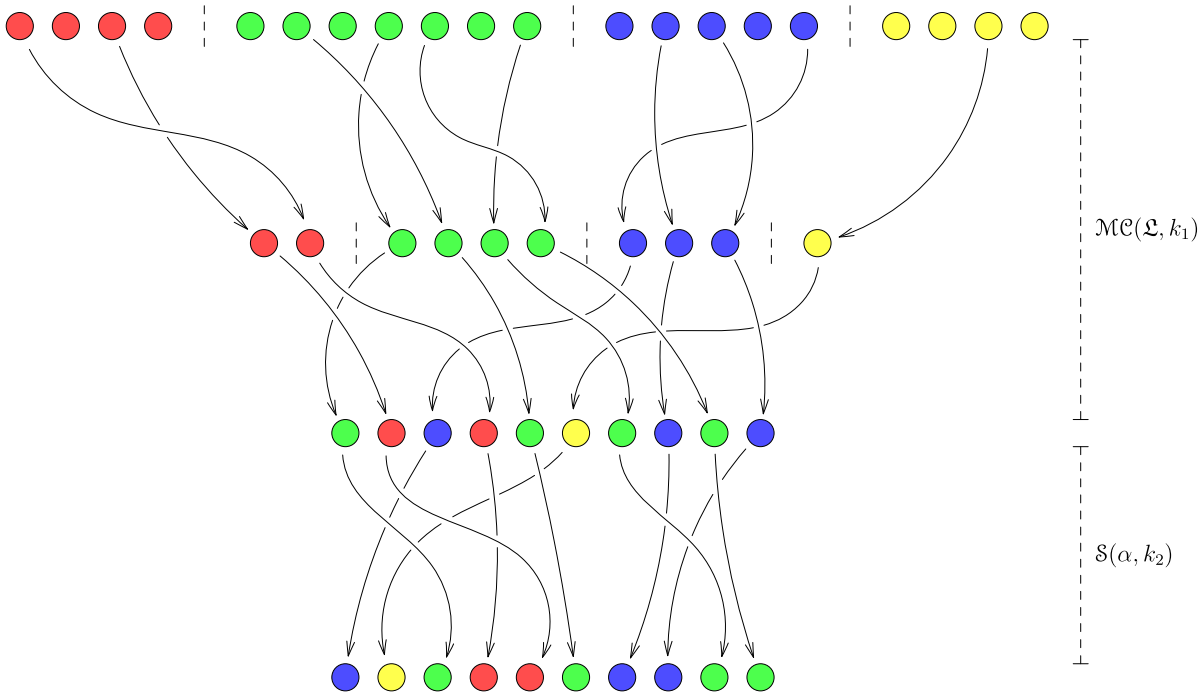


Figure 1: An illustration of the working of the pseudo-hash function

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 number defined by the environment and available to the public. We will assume that it is 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 $\left(\#^{\mathcal{MC}}(\mathfrak{L})\right)$. 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. Counting ages of the Universe

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	10
1	`"RQLIANBKJYVWPTEMCZSFDOGUHX"`	26	10
2	`"=!*@?$%#&-+^{}"`	14	6
3	`"1952074386"`	10	6

Table 1: An example of a source configuration

Therefore, every password produced with this configuration will contain a total of 32 symbols, 10, 10, 6 and 6 from their respective categories.

Now, let's imagine that you have inserted your two private keys into the function and got a password out of it. A sophisticated hacker sets their mind to crack your password whatever it takes. They are very smart and they have a supercomputer that can perform 1,000,000,000,000 password checks in a second, or one picosecond to check one password or key. What is more, they got their hands on the pseudo-hash function and the configuration you use, so they can try to reverse-engineer your password.

First, they read into the configuration and see the structure of the password. They decide to brute-force it by checking every relevant combination of 32 symbols. Well, they will have to check $\#^{\mathcal{H}}(\mathfrak{L})$ combinations, which in our case equals exactly

$$\#^{\mathcal{H}}(\mathfrak{L}) = 4,681,868,099,431,597,288,493,491,203,991,195,290,886,471,680,000,000,000 \approx 4 \cdot 10^{54}.$$

Cracking it would take the supercomputer about

$$148,359,447,468,489,290,599,901,768,793,980,928 \approx 10^{35}$$

years, or about

$$10,760,821,605,025,697,027,325,952 \approx 10^{25}$$

ages of the Universe.

Okay, thinks the hacker, no luck. They dig a little deeper into the algorithm and find that your password depends on two private keys. The number of all pairs of such keys is

$$\begin{aligned}\#^{(k_1,k_2)}(\mathfrak{L}) = 1,231,943,871,416,597,272,872,197,263,679,034,862,\\ 284,237,334,545,743,786,828,619,158,565,735,628,\\ 800,000,000,000,000,000 \approx 10^{90}\end{aligned}$$

This one is going to take billions of billions times longer than the previous one.

Okay, thinks the hacker, the night is young. They are able to get their hands on one of your 32-symbol passwords because of a security leak on the website you were registered to. They have also dug ears deep into the pseudo-hash function and understood how it works to the tiniest detail. Now, they want to retrieve your private keys to be able to generate your passwords to all other websites and services you use. They see that your password starts with a `'W'`. They know it was shuffled from some other position by the shuffle function, $\mathcal{S}(-,k_2)$, but they don't know what the password was before the shuffling. What they do know is that your password is produced by

$$\#^{\cap}(\mathfrak{L}) = 263,130,836,933,693,530,167,218,012,160,000,000 \approx 2 \times 10^{35}$$

different (k_1,k_2) pairs. The hacker can (relatively) easily retrieve any one of these pairs by substituting the intermediary layer α (after the application of $\mathcal{MC}$, but before $\mathcal{S}$) of their choosing, and then solve for the two keys. The problem is that if the keys they receive differ from the true keys, they will not produce the correct password given a different public key. Therefore, to get the key to another website, the hacker would need to go through all of $\#^{\cap}(\mathfrak{L})$different combinations, which will take them about $8,338,113,067,333,812 \approx 8 \times 10^{15}$ years, or $604,780$ ages of the Universe. It is, of course, almost infinitely better than brute-forcing the password from scratch (after all, the resulting hash does give the hacker a lot of information about the original keys), but it is still pretty much impossible.

After that, the hacker can go to sleep and forget about cracking your password, because they have nothing left to try. And you can go to sleep knowing that your accounts are safe.

2.9. Some advice

All the above calculations are based on the premise that the hacker will not get your key pair among the first couple of hundreds, that they will have to deplete the entire spread of the pseudo-hash function to find your choice-shuffle pair. Therefore, if your pair is something like $(14,245)$, the password will not take a single millisecond to be broken. It is therefore advisory that the keys be very large — somewhere in the middle of the spread. You can practice your memory and remember two large numbers, if you are not afraid of forgetting them (and you should be). Or, you can take the factorial, or a power, of a smaller number and remember that number instead.

But anyway, how can *you* get some of those hashes yourself?

3. The implementation

3.1. A brief introduction to Haskell

Haskell, named after mathematician Haskell Curry, is one of the most renown functional programming languages. The term "functional" here refers to the following principles:

- **Function purity.** While in standard procedural languages a function is usually just a macro, i.e. a piece of code that can be run multiple times, in Haskell and other functional languages functions are understood mathematically. They are abstract and immutable objects that accept inputs and return outputs. They do not have access to any external variables that are not functions themselves. That property is called *purity*.
- **Stateless programming.** Ever since the model of Turing machines, the idea of state has been integral to programming. It implies that a program can be in different states, which affects its behavior. This does, however, make the program potentially more unpredictable, since, even given the same input twice, it may return different outputs based on its state. In functional programming, state does not exist. All functions and variables are constant and are not allowed to change. As a result, a Haskell program, much like a mathematical function, is guaranteed to give the same result with the same input.

It is obvious that the hashing algorithm, which is basically a mathematical function and which must return the same hash given the same keys, is best implemented in a functional programming language. Haskell has been chosen because of its simple (yet unconventional) syntax, as well as amazing flexibility and a developed infrastructure for working with numbers and strings. Particularly, it has the ability to store integer numbers of *arbitrary length* (not just, say, 8 bytes). This is crucial given the monstrous numbers seen in Section 2.8.

3.2. The `pshash` program

`pshash` (from "password-hash") is the name of the implementation of the pseudo-hash function described in the previous section. There is no front-end or GUI, it is only a command-line utility designed to be simple and lightweight. The source, written in Haskell, is available at https://github.com/thornoar/pshash. The code repeats the definitions explained in the previous section almost verbatim — such is the benefit of functional programming.

The program 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 `-d`, `-s`, 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. You can run `"pshash --help"` to see all available options.

! Important notice !

The program is *extremely sensitive to changes*. Thus, before using it to create passwords, make sure that the configuration is set to your liking, save the executable and do *not* change it afterwards. Any small change, and all your passwords will be lost. You will have to undo the change to retrieve them. Additionally, it is advised to store the program openly on the internet, in case you lose your local copy. This way, you can access it from anywhere with an internet connection. Currently, there are pre-compiled binaries for Linux (NixOS, Debian, and Arch), as well as Windows and Android, all hosted at https://github.com/thornoar/pshash-bin. There is also a web interface at https://thornoar.github.io/pshash/web/app/.

3.3. Other implementations

The pseudo-hashing algorithm is implemented in other languages, namely C (to provide 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/), and Kotlin (for the `pshash-app` Android app, source available at https://github.com/thornoar/pshash-app).