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. 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.
- **Using a password manager.** 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 TODO, 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 key or the passwords that it produces. In other words, the algorithm has to adhere to Kerchhoff's principle. 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 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 $0 \leq {}^n k \leq n$ 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 $\mathrm{spr}(f)$.

Trivially, if $\mathrm{spr}(f) \geq n$, then f is injective on (n), but the converse is not always true. Therefore, a lower bound on the spread of a function server as a guarantee of its injectivity on a certain interval. Furthermore, if $\mathrm{spr}(f) \geq n$ 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* $\mathrm{spr}(f) \geq n$ *and* $\mathrm{spr}(g) \geq m$*. 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* $\mathrm{spr}(h) \geq nm$*.*

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:

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

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

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

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.