On rearrangement hashing with Haskell

Roman Maksimovich

Abstract

In this paper I introduce and develop a mathematical method of producing a cryptographic hash of adjustable length, given a public key and a private key. The hashing is done through encoding selections and permutations with natural numbers, and then composing the hash from a set of source strings with respect to the permutations encoded by the keys. The attempts to construct a suitable integer-to-selection mapping leads to interesting mathematical definitions and statements, which are discussed in this paper and applied to give bounds on the reliability of the hashing algorithm. An implementation is provided in the Haskell programming language (source available at `https://github.com/thornoar/password-hash`) and applied in the setting of password creation. In the paper, the details of the implementation are discussed, as well as the connections between it and the corresponding mathematical model.

March 24, 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, one 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. It is usually unclear, however, how this software works and whether it can be trusted with one's potentially very sensitive passwords. After all, guessing the password to the password manager is enough to have all the other passwords exposed.

In this paper I 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 cryptographic hash produced by a fixed hashing algorithm. The algorithm requires two inputs: the public key, i.e. the name of the website or service, and the private key, which is an arbitrary positive integer known only to the user. Every time when retrieving a password, the user will use 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 working, 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 hashing algorithms and implementing them in a functional programming language such as Haskell.

2 The theory

There are many ways to generate 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 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. By $|E|$ we will denote the cardinality of a source E, and $E:i$ will represent its i-th element, with the numeration starting from $i = 0$. On the opposite, the expression $E\,!\,i$ will denote the set difference $E \setminus \{E:i\}$.

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. 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 $\alpha +\!\!+ \beta \in [A \cup B]$ will be the concatenation of lists α and β.

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 $(n \mid m)!$.

If f is a function of many arguments $a_0, ..., a_{n-1}$, the expression $f(a_0, ..., a_{i-1}, -, a_{i+1}, ..., a_{n-1})$ will represent 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 key. It exists solely for the purpose of producing different passwords with the same private key. So for now we will forget about it. In this and the following subsection we will focus on the method of mapping a 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. Let E be a source, $k \in \mathbb{N}_0$. The *choice function of order 1* is defined as the following one-element list:

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

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 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 $k_1, k_2 \in \mathbb{N}_0$, $k_1 \neq k_2$, the following implication holds:

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

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

Trivially, if $\operatorname{spr}(f) \geqslant n$, then f is injective on (n), but the inverse is not always true. Therefore, a lower bound on the spread of a function serves as a guarantee of its injectivity. Furthermore, if $\operatorname{spr}(f) \geqslant n$ 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 2.3. *Let* $f\colon \mathbb{N}_0 \to A$, $g\colon \mathbb{N}_0 \to B$ *be functions such that* $\operatorname{spr}(f) \geqslant n$ *and* $\operatorname{spr}(g) \geqslant m$. *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* $\operatorname{spr}(h) \geqslant nm$.

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:

$$f({}^{n}k_1) = f({}^{n}k_2), \tag{1}$$

$$g({}_{n}k_1 + T({}^{n}k_1)) = g({}_{n}k_2 + T({}^{n}k_2)). \tag{2}$$

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 can then proceed to utilize the definition of spread for the function g:

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

■

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 $|E| = n$, $k \in \mathbb{N}_0$, $2 \leqslant m \leqslant n$. The *choice function of order* m is defined recursively as

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

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* $\mathcal{C}^{m}(E, k)$ *of order* $m \leqslant n$, *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 $\mathcal{C}^{1}(E, k)$ 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:

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

q.e.d. ■

The preceding result is especially valuable considering the fact that there are exactly $(n \mid m)!$ 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 $((n \mid m)!)$. 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 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's 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 $T({}^nk)$ to the ASCII value of the character $E : {}^nk$. 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:

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 placed, because it contains only one category of symbols. Fortunately, there is a solution.

2.3 Elevating the choice function

Definition 2.6. Let α be the 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:

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

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

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

In this context, the list α 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 will call a *multiselection*. A trivial application of proposition 2.3 shows that the spread of $\overline{\mathcal{C}}(\alpha, -)$ is at least

$$\prod_{i=0}^{N-1} \operatorname{spr}(\mathcal{C}^{m_i}(E_i, -)) = \prod_{i=0}^{N-1} (n_i \mid m_i)! \tag{3}$$

where E_i, n_i, and m_i compose the configuration α. In fact, due to the rule of product in combinatorics, we see that the expression in (3) directly corresponds to the number of possible multiselections from α, or $\#^{\overline{\mathcal{C}}}(\alpha)$ for convenience. Therefore, $\overline{\mathcal{C}}(\alpha, -)$ is bijective on the interval $\big(\#^{\overline{\mathcal{C}}}(\alpha)\big)$ and periodic with period $\#^{\overline{\mathcal{C}}}(\alpha)$.

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 it to make the choice function artificially non-injective, or overlapping, 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 both problems at the same time.

2.4 The hash function

Definition 2.7. Let α be a source configuration consisting of pairs (E_i, m_i) for $i \in (N)$. For two numbers $k_1, k_2 \in \mathbb{N}_0$, define their *hash* by the following expression:

$$\mathcal{H}(\alpha, k_1, k_2) = \mathcal{C}^{\Sigma m_i}\left(+\!\!+\overline{\mathcal{C}}(\alpha, k_1), k_2\right),$$

where $+\!\!+ \colon [[A]] \to [A]$ is list concatenation. This definition can be re-written in a more readable way by defining the *shuffle function* $\mathcal{S}(E, k)$ as $\mathcal{C}^{|E|}(E, k)$ and letting the source configuration α be the varying argument:

$$\mathcal{H}(-, k_1, k_2) = \mathcal{S}(-, k_2) \circ (+\!\!+) \circ \overline{\mathcal{C}}(-, k_1).$$

The hash function makes a selection from every source in the configuration, then concatenates all these selections, and finally reshuffles the resulting source. The two keys k_1 and k_2 used to make a selection will be called the *choice key* and the *shuffle key* respectively.

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 $\mathcal{H}(\alpha, k_1, k_2)$ for a given source configuration α:

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

$$\#^{\overline{\mathcal{C}}}(\alpha) = \prod_{i=0}^{N-1}(n_i \mid m_i)!$$

  This is because $\overline{\mathcal{C}}(\alpha, -)$ is injective on this interval, and $\mathcal{S}(-, k_2)$ is injective as well. With respect to the shuffle key k_2, the hash function is injective on the spread of $\mathcal{S}$, which is

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

  Therefore, the number of relevant key pairs for the hash function, denoted by $\#^{(k_1,k_2)}(\alpha)$:

$$\#^{(k_1,k_2)}(\alpha) = \#^{\mathcal{S}}(\alpha) \cdot \#^{\overline{\mathcal{C}}}(\alpha) = \prod_{i=0}^{N-1}(n_i \mid m_i)! \cdot \left(\sum_{i=1}^{N-1} m_i\right)!$$

- **Overlapping.** However, this number does not equal the number of all possible values of $\mathcal{H}$. When applying $\mathcal{S}$ after $\overline{\mathcal{C}}$, 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 concatenated and reshuffled with $\mathcal{S}$. Since there are $m_i!$ ways to reorder every sub-list chosen by $\overline{\mathcal{C}}$ from E_i, the amount of lost information accounts for a total of $\prod_{i=0}^{N-1}(m_i!)$, which will be denoted by $\#^{\cap}(\alpha)$. It is the number of pairs (k_1, k_2) that produce the exact same hash. Hence, we see that the number of possible outputs of the hash function, given a configuration α, is

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

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.

2.5 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 choice-shuffle pair for every new occasion. You may as well try to remember the hashes themselves. This is where the public key comes into play. The public, denoted p, is an integer number defined by the situation and available to the public. We will assume that it is the name of the website for which the password is required, like `"google"`, for example, converted into a number by treating the characteers in the string as digits in base-128. This number acts on the choice private key by shifting it modulo $\#^{\overline{\mathcal{C}}}(\alpha)$:

$$k_1' = {}^{\#^{\overline{\mathcal{C}}}(\alpha)}(k_1 + p),$$

and then this new choice key is plugged into the 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 $\left(\#^{\overline{\mathcal{C}}}(\alpha)\right)$. Its influence on k_1 is simple and predictable, but it doesn't have to be complex, since the public key is not responsible for any encryption. Instead it is designed to be easily remembered.

2.6 Counting ages of the Universe

In this subsection, we will use a specific source configuration α, particularly the following:

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

Therefore, every password produced with this configuration will contain a total of 30 symbols, 10, 10, 5 and 5 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 nanosecond to check one password or key. What's more, they got their hands on the 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 30 symbols. Well, they will have to check $\#^{\mathcal{H}}(\alpha)$ combinations, which in our case equals exactly

$$\#^{\mathcal{H}}(\alpha) = 1,493,683,548,205,755,958,753,683,829,702,969,075,433,472,000,000,000.$$

Cracking it would take the supercomputer about $47,331,975,441,914,340,970,572,896,272,384$ years, or about 3 billion of millions of millions of ages of the Universe.

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

$$\begin{aligned}\#^{(k_1,k_2)}(\alpha) = \ & 283,235,154,137,060,638,511,193,623,530,464, \\ & 898,307,082,375,546,273,792,000,000,000,000,000\end{aligned}$$

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 30-symbol passwords because of a security leak on the website you were registered to. They have also dug ears deep into the 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

$$\#^{\cap}(\alpha) = 189,621,927,936,000,000$$

different (k_1,k_2) pairs.