# SELECTION-BASED MINIMALISTIC PASSWORD GENERATION

Roman Maksimovich
rmaksimovich@connect.ust.hk

March 26, 2024

**Abstract:** The traditional model of password management has been failing for the past 25-30 years, with the growth in the number of passwords an average user needs to remember. To combat this problem, we introduce `pshash` — a deterministic "on-the-fly" password generation algorithm based on selecting characters from predefined sources. The selection of characters and their order are derived from two secret numbers (the "choice key" and "shuffle key") provided by the user, and the resulting character string can be used as a password. The algorithm resembles a hash function in many ways, however it is not cryptographically secure in the usual sense. Yet it still satisfies the conditions for being a *universal hash function*, and we argue that this is sufficient for secure password generation. Its main benefits are simplicity (it is implemented in under 200 lines of code) and independence (it does not draw on any pre-existing algorithms or cryptographic notions). This comes at the cost of convenience, as the user is required to memorize a relatively large amount of entropy to achieve good security. However, we argue that practice can mitigate this drawback, and the benefits of security and minimalism are worth it.

**Keywords:** Passwords, Authentication, Combinatorics, Functional Programming

## 1 Introduction

In our time, the average computer user has more than 40 different accounts which require usernames and passwords [13]. Overall, we can distinguish between three methods of managing them, in increasing levels of security and decreasing levels of convenience:

- **Memorizing passwords directly.** Much like the one-time pad encryption scheme ([9], page 40), this method theoretically provides the best possible security, assuming that all passwords are strong and have no connection to each other. Unfortunately, with a large number of passwords this assumption fails entirely. Statistical studies report that most people reuse their passwords between both important sites (e.g. banking, work-related) and less critical sites (e.g. forums, social media), and overall choose weak passwords for the sake of simplicity [10], [13], [14]. Naturally, this leads to the idea of using computer-assisted password management.
- **Using a password manager.** These are software programs that can store a user's passwords in an encrypted database, allowing the user to decrypt it using a *master password*. Examples include 1Password [1], Google Chrome [3], and Mozilla Firefox [4]. A great advantage of password managers is the ability to create random and provably strong passwords on behalf of the user. However, password managers have been criticised for insecure database storage formats [11] and auto-fill policies [16], which both leave doors for various attacks. It appears that the sources of most vulnerabilities are 1) the fact that passwords are stored on disk, albeit in an encrypted form; and 2) the tight integration of password managers with browsers, provided for the sake of convenience and some security features such as defence against phishing and spoofing [15]. Additionally, many password managers are closed-source, which hinders one's trust that the password manager will not abuse the passwords it stores.
- **Using a deterministic "on-the-fly" password generator.** These are programs that allow the user to derive a password from their identity and a secret master password, using a stateless (usually hash-based) function. An example is PwdHash [15]. This approach eliminates the problem of storage, since nothing is ever stored on disk. However, being based on well-known hash functions opens such schemes to dictionary attacks and brute-force attacks, although mitigated in [12]. Still, deterministic password generators are still more complicated than they could be, and they often require users to provide their identity, which hinders anonymity.

In this paper, we specify a new "on-the-fly" password generation algorithm `pshash` that differs from its alternatives in two important ways:

- `pshash` is extremely minimalistic (written in under 200 lines of code) and standalone, which makes it very portable and easy to implement on any platform and in any programming language that supports arbitrary-precision arithmetic (see Section 4 for a discussion of `pshash`'s implementation). The program is detached

from any external service or browser, making it invulnerable to supply chain attacks.

- Our algorithm uses a custom universal hash function instead of the "industry standards" of cryptographic hashing such as SHA-256 or Salsa20. This means that our algorithm is immune to attacks targeting these hash functions, such as dictionary attacks.

The above benefits come at the cost of convenience, which means that `pshash` is probably ill-suited for mass adoption. However, it fits perfectly for technically educated users who do not mind memorizing a couple of 20+ digit numbers for the sake of security.

# 2 The Algorithm Specification

In this section, we give a full description of `pshash`. The algorithm is a pure stateless function that takes as input three values:

- The domain name (or any other distinct identifier) of the website for which a password is required, called the *public key* and denoted $p$;
- A secret non-negative integer $k_1$ which lies in a vast range (on the order of $2^{142}$, typically), called the *choice private key*;
- Another secret non-negative integer $k_2$, which lies in a smaller range of numbers (on the order of $2^{83}$, typically), called the *shuffle private key*.

The function returns a string of characters which we call a "pseudo-hash" since it is not a full cryptographic hash but has some of its properties. `pshash` does not derive the characters from the keys directly, but instead uses them to select characters from predefined *sources*, which makes it much easier to make sure the pseudo-hash complies with a website's password requirements. The algorithm also supports different password templates, called *source configurations*. We will now consider the algorithm in detail.

## 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 $\alpha$ and $\beta$.

Let $\alpha$ 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 $\alpha$ 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 purpose of the public key $p$ is solely to allow the user to generate different passwords with the same set of private keys. Therefore, it need not be an integral part of the core *pshash* algorithm. In this and the following subsections we will build the function $\mathcal{H}(\mathfrak{L}, k_1, k_2)$ which accepts a source configuration and two private keys to produce a pseudo-hash, and we will explain the origin of the decision to use two private keys instead of one.

**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:

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

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 $\operatorname{spr}(g)$:

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

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 $(n \mid 1)! = n$, 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 special 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

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

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 building.

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 $\alpha$ or $\beta$ 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 $\alpha$ nor $\beta$ 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 $\alpha$ and $\beta$ 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 $s_1 + s_2$:

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

**Proposition 2.9.** *Let $\alpha$ and $\beta$ 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:

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

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 $\alpha$ and $\beta$, 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:

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

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*

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

*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:*

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

*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 $\alpha_1$, $\alpha_2$, 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, our statement 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 *pseudo-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 $\alpha$ 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 in our opinion.

### 2.7 The Public Key

At this stage, the only way is to create a new choice-shuffle key pair for every new occasion. 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/organization 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 $(\#^{\mathcal{MC}}(\mathfrak{L}))$. The influence of $p$ 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 supposed to be plain and easily remembered.

## 3 Security Properties

In this section, 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.

We will also assume that an adversary can perform $10^{10}$ password checks per second, i.e. 0.1 nanosecond to perform one operation such as computing a pseudo-hash or computing a key pair that produces a given pseudo-hash value. This measure is in fact extremely conservative. For example, running `openssl speed -evp sha256` to benchmark the speed of SHA256 on an 22-core Intel Ultra 9 185H CPU has yielded a speed of $10,000,000$ hashes per second, or 100 nanoseconds per hash. We assume here that `pshash` is $1,000$ times faster, whereas in fact it must be orders of magnitude slower, due to the fact that it must perform many arithmetic operations on arbitrary-precision integers. This conservative speed estimate serves as a guarantee of security for a long period of time, even when processors become a million times faster than they are today.

We will now consider various attacks on the pseudo-hashes produced by `pshash`.

### 3.1 Brute-force Attacks on Pseudo-hashes

First of all, an adversary 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 adversary 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.

### 3.2 Attacks with a Known Pseudo-hash

Having examined the `pshash` algorithm further, an 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 choosing the intermediate layer $\alpha$ between $\mathcal{MC}$ and $\mathcal{S}$). 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 $(k_1, k_2)$, and the pseudo-hash $h$ gives no preference to any of these pairs. 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.

### 3.3 Multiple Known Pseudo-hashes

Now, suppose that multiple pseudo-hashes $h_1, h_2, ..., h_n$ have been made available to the adversary, all produced with the same choice-shuffle key pair $(k_1, k_2)$ and different public keys $p_1, p_2, ..., p_n$ which are also known. Denote the number $\#^{\mathcal{MC}}(\mathfrak{L})$ by $N$. Essentially, the adversary needs to solve for $(k_1, k_2)$ the following system of equations:

$$\begin{aligned}h_1 &= \mathcal{H}(\mathfrak{L}, {}^{N}(k_1 + p_1), k_2),\\ h_2 &= \mathcal{H}(\mathfrak{L}, {}^{N}(k_1 + p_2), k_2),\\ &\vdots\\ h_n &= \mathcal{H}(\mathfrak{L}, {}^{N}(k_1 + p_n), k_2).\end{aligned}$$

Due to the argument shift function $T$, the pseudo-hash function is highly non-linear and cannot be described as an algebraic transformation of its inputs. Hence there is no way (as to our knowledge) to narrow down the space of possible solutions of the above system without simply traversing $\#^{\cap}(\mathfrak{L})$ different key pairs and checking whether they produce all the given pseudo-hashes.

In other words, we have seen that `pshash` is resistant to various brute-force attacks and dictionary attacks.

### 3.4 Supply Chain Attacks

Being a standalone program, `pshash` is not susceptible to supply chain attacks. However, this also means `pshash` does not protect from phishing and keyloggers, being completely unaware of them. Hence, `pshash` should be used with care, as it is only responsible for producing a password, and not for how it is used afterwards.

## 4 The Implementation

For the primary implementation of the $\mathcal{H}(-,-,-)$ function, we have chosen the Haskell programming language. This choice is natural because the `pshash` 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.

The primary implementation is a CLI program running in the console. It was designed without any GUI in order to make it more lightweight and portable, as well as faster. It is fully open-source, with source code available on GitHub [6], together with installation instructions.

The general invocation of `pshash` is `$ pshash PUBLIC CHOICE SHUFFLE`, providing the public key and the choice-shuffle private key-pair. To avoid passing secrets directly as command-line arguments, the user may omit any of the three arguments and provide them via `stdin`. In addition, the program has many flags and options, all of which can be viewed by invoking `$ pshash --help`. We will expand on some of the most important features below.

## 4.1 Simplified Key Input and Generation

One of the biggest disadvantages of `pshash` is the size of the private keys. For the source configuration given in Section 3, the maximum relevant choice key is on the order of $2^{142}$ and the maximum shuffle key on the order of $2^{83}$. Such long numbers are possible to remember using mnemonics, but they still take a lot of effort to commit to memory, and the cost of forgetting even a single digit is the permanent loss of all passwords derived from this key-pair. Additionally, even coming up with such big numbers is hard for a human to do. One would inevitably think of numbers relevant in their life, which makes them vulnerable to social analysis attacks.

These problems are solved in the implementation of `pshash` as follows:

- The program can generate random keys via invoking `$ pshash --gen-keys`. The keys are automatically selected from the range corresponding to the source configuration in use (see Subsection 4.4 for more information on how they are specified). This ensures that the generated keys are uniformly distributed in the relevant range, and allows the passwords generated by these keys to enjoy the security properties discussed in Section 3.
- The program supports alternative input methods for the keys. For example, it can evaluate arithmetic expressions involving addition, multiplication, and exponentiation. Instead of `52046849220153179918678225216` one may type `8234 * 91234 ^ 5` which is much easier to remember. Of course, if this input method becomes mainstream, a brute-force attack on such passwords will be much easier, since it will be enough to check all numbers of the form $a + b$, $a * b$, $a^b$ and some basic combinations of these expressions, where $a, b < 100,000$. Hence the arithmetic input method is not preferred.
  As a safer alternative, the `pshash` implementation offers an "mnemonic incantation system". By selecting the 100 most used English 2-letter syllables, we associate each 2-digit number with a syllable. For example, `00` corresponds to `or`, `37` corresponds to `mi`, and `99` corresponds to `je`. We can then apply this map to whole numbers, so `993700` becomes `jemior`, which sounds like an English word and is much easier to remember. Given the source configuration described in Section 3, the average choice key will have about 7 three-syllable words in its mnemonic incantation, while the average shuffle key will have about 4 words. An example would be

  $$59257582635437578679843077171194558385905 18 \longleftrightarrow \texttt{usjusu sodilo hiwewa quzihi agfefe riposo tobius li}$$
  $$14597701819718601692712560 \longleftrightarrow \texttt{rebifu unpuja litule jufela tu}$$

  Remembering these incantations still requires effort, of course, but it is feasible, and the nature of `pshash` is such that the security properties of passwords rely directly on the amount of entropy in the keys.
  To generate an incantation, the user may invoke `$ pshash --gen-spell`. They will be prompted for the numeric key (which can be typed as an arithmetic expression), after which the incantation will be printed as output. The `$ pshash --gen-keys` command will also automatically print corresponding incantations.

## 4.2 Error Messages

The `pshash` implementation supports a state-of-the-art error message system. Whenever the algorithm fails at any step, a helpful error message with a stack trace will be printed. For example, `$ pshash asd asd asd` will produce

```
ERROR:
|_Double trace:
  |_Trace while computing hash, getting the *choice* key:
  | |_Both numeric and mnemonic read methods failed:
  |   |_FAILED TO READ "*asd*" AS *integer*.
  |   |_UNKNOWN MNEMONIC SYLLABLE: "*as*".
  |_Trace while computing hash, getting the *shuffle* key:
    |_Both numeric and mnemonic read methods failed:
      |_FAILED TO READ "*asd*" AS *integer*.
      |_UNKNOWN MNEMONIC SYLLABLE: "*as*".
```

In shells supporting color codes, the error output can also be colored to highlight the relevant parts. Such a detailed error output is achieved via the concept of monadic data structures prevalent in functional programming languages. In essence, most functions in the `pshash` implementation source code may return either a value or an error object which keeps a trace of error messages. The error system makes `pshash` reliable in the sense that, if the program runs without errors, the user can be sure that all inputs were interpreted correctly and the resulting pseudo-hash can be safely used.

## 4.3 Key Retrieval

As mentioned before, if the final pseudo-hash $h$ and, say, the choice private key $k_1$ are given, one can solve the equation $h = \mathcal{H}(\mathfrak{L}, k_1, x)$ to find the shuffle key $k_2$ (modulo the spread of $\mathcal{S}$, of course). Similarly, the choice key (modulo the spread of $\mathcal{MC}$) can be retrieved from the shuffle key and the final pseudo-hash.

Since this can be done, there is no reason why this functionality should not be included in the implementation of `pshash`. By invoking `$ pshash -q KEYWORD`, where `KEYWORD` is either `public`, `choice`, or `shuffle`, the user will be prompted for the two remaining keys and the final pseudo-hash, after which the desired key will be printed to the console.

This functionality is useful if the user forgets one key but not the other, and has access to one of the passwords produced with that key-pair.

Moreover, the `--list` flag allows to print the first $N$ choice-shuffle key-pairs that produce a given pseudo-hash with a given public key.

## 4.4 Specifying the Source Configuration

The source configuration $\mathfrak{L}$ is not given to `pshash` as a direct argument, but optionally specified with one of the `-k`, `-n`, and `-c` flags. See `$ pshash --help` for more details.

Since different websites have different password requirements, the implementation of `pshash` provides many built-in source configuration templates. The default is the one given in Section 3, and it will be used if no other is specified. The default configuration produces passwords that are accepted by most websites, but for rare cases there are purely alphanumeric templates of different sizes, as well as pin code templates.

Additionally, the user can write a *configuration file* which specifies source configurations for different public keys. This file can then be read from a standard location if the `--impure` flag is passed to `pshash`. Note that the default behavior of `pshash` is to ignore all configuration files, to ensure that the program functions deterministically and gives the same outputs on any machine. The `-f PATH` can also be passed, which automatically implies `--impure` and will cause `pshash` to read the configuration file from `PATH`. For details on the configuration file format, see `$ pshash --help`.

## 4.5 Replacing Passwords

If one of the passwords produced by `pshash` is compromised, the safest option is to create a new random key-pair and change all passwords. However, this is obviously inconvenient. For this reason, the `pshash` implementation provides a "public key patch" mechanism. Patches can be specified with the `-p` flag, as in `$ pshash PUBLIC -p N`. This will shift all characters in `PUBLIC` by `N` modulo $128$, and then use the "patched" public key as usual.

We readily see that `$ pshash PUBLIC -p 0` is equivalent to `$ pshash PUBLIC`, and that different patches produce different final pseudo-hashes, as long as `N` is between $0$ and $128$.

This way, a user who needs a new password for, say, `spotify`, can invoke `$ pshash spotify -p 1` instead of `$ pshash spotify`. In order not to keep all patch values in their head, the user may also save them in the configuration file.

## 4.6 File Encryption

Recall that the $\mathcal{S}(-, k)$ function from Subsection 2.6 can shuffle any ordered list of distinct elements, given a key $k$. This allows us to transform the `pshash` algorithm into a simple block cipher ([9], page 93). For a number of rounds $r$ and two private keys $k_1, k_2$, we define a permutation by

$$\varphi = (\mathcal{S}(-, k_2) \circ \mathcal{S}(-, k_1))^r,$$

Applying this permutation to the list $[0, 1, 2, ..., 63]$, we receive a permuted list $[n_0, n_1, ..., n_{63}]$. Now, given a plaintext 64-byte block $P = [P_0, P_1, ..., P_{63}]$, we obtain the corresponding ciphertext block by shuffling its bytes according to the permutation:

$$C = e_\varphi(P) = [P_{n_0}, P_{n_1}, ..., P_{n_{63}}].$$

To decrypt $C$, it suffices to invert the permutation $\varphi$ and repeat the same procedure, i.e. $P = e_{\varphi^{-1}}(C)$.

As before, the security of this cipher does not correspond to the conventional notion. Given a ciphertext block $C$, one can easily generate a pair of keys and a plaintext block that will produce $C$ under the above procedure (for example, one can take $k_1 = k_2 = 0$ and calculate the corresponding plaintext). However, there is no information in $C$ about the *particular* key-pair and plaintext that were used, and the amount of possible options is equal to

$$\mathrm{spr}(\mathcal{S}([0, ..., 63], -)) = 64! \approx 2^{295},$$

which is simply the number of all possible plaintexts that correspond to the given ciphertext.

Now, to exncrypt/decrypt byte streams of length other than 64, we run our block cipher in (a slightly modified) counter mode ([9], page 118). In particular, let a plaintext $P$ of arbitrary size be given, along with two keys $k_1, k_2$. We calculate the permutation $\varphi$ based on $k_1, k_2$, and we also generate a random 64-byte block $IV$. Now, we build a sequence $(B_i)_{i=0}^{\infty}$ of 64-byte blocks as follows:

- We set $B_0$ to $IV \oplus DS$, where $DS$ is a fixed random 64-byte block generated beforehand and hard-coded in the executable;
- If $B_i$ is computed, we set $B_{i+1} = e_{\varphi}(B_i \oplus DS)$.

Now, to encrypt $P$, we simply concatenate $(B_i)$ into a stream of bytes $B$ and then XOR it with $P$, prepending the initial value $IV$ for decryption:

$$C = e(P) = IV \parallel (P \oplus B).$$

To decrypt a ciphertext $C$, we first extract the $IV$ from its first 64 bytes, yielding a shorter ciphertext $C'$, and then build the same sequence of 64-byte blocks as during encryption. Finally, we compute

$$P = d(C) = C' \oplus B.$$

This simple cipher is included in `pshash`'s primary implementation. By invoking `$ pshash -e FILE`, the user will be prompted for the destination `OUT`, as well as two keys, after which the contents of `FILE` will be shuffled and written to `OUT`. The program can read from `stdin` and write to `stdout` if `FILE` is `"stdin"` or `OUT` is `"stdout"`. Of course, both keys can be specified as arithmetic expressions or mnemonic incantations.

### 4.7 Other implementations

The `pshash` algorithm is implemented (at least in its skeleton form) in other languages and deployed on other platforms:

- In **C**, using the GNU Multiple Precision Arithmetic library [2]. This serves as a faster bare-bones implementation of `pshash` (since Haskell is often very slow), and also facilitates the in-development `pshash` desktop app [8];
- In **JavaScript**, to facilitate the `pshash` web interface [5]. This is a very important deployment location since it allows users to generate passwords from any machine that has internet access, without installing anything locally. The web interface is a static HTML page hosted via GitHub Pages, with no server-side interaction. The passwords are generated by JavaScript on the client side and can then be either displayed or copied to the clipboard;
- In **Kotlin** for the `pshash` Android application [7].

## 5 Conclusion

We have presented the `pshash` password generation algorithm that differs from its closest analogues by its simplicity and originality. The minimalistic design of `pshash` comes at the cost of user convenience, since the secrets that need to be remembered by the user are big in size. This problem is partly mitigated in the implementation of `pshash` with two alternative key input methods. The implementation has many other features as well, but it provides no integration with the browser or any external program, for the sake of independence, but at the cost of convenience. Finally, other implementations of `pshash` are available, which makes the algorithm accessible from any platform and on third-party machines.

## References


[1] 1Password Password Manager. `https://1password.com/`. Accessed: 17 Feb 2026.

[2] GNU Multiple Precision Arithmetic Library. `https://gmplib.org/`.

[3] Google. `https://www.google.com/chrome/`. Accessed: 17 Feb 2026.

[4] Mozilla Firefox. https://www.mozilla.org/. Accessed: 17 Feb 2026.

[5] Pshash Web Interface. https://thornoar.github.io/pshash-web/app/.

[6] Source code for the primary implementation of pshash. https://github.com/thornoar/pshash.

[7] Source code for the pshash android application. https://github.com/thornoar/pshash-app.

[8] Source code for the pshash desktop application. https://github.com/thornoar/pshash-gui.

[9] J. Aumasson. *Serious Cryptography, 2nd Edition: A Practical Introduction to Modern Encryption*. No Starch Press, 2024.

[10] D. Florencio and C. Herley. A large-scale study of web password habits. pages 657–666, 05 2007.

[11] P. Gasti and K. Rasmussen. On the security of password manager database formats. 09 2012.

[12] J. Halderman, B. Waters, and E. Felten. A convenient method for securely managing passwords. pages 471–479, 01 2005.

[13] A. Hanamsagar, S. Woo, C. Kanich, and J. Mirkovic. Leveraging semantic transformation to investigate password habits and their causes. pages 1–12, 04 2018.

[14] P. Kelley, S. Komanduri, M. Mazurek, R. Shay, T. Vidas, L. Bauer, N. Christin, L. Cranor, and J. Lopez. Guess again (and again and again): Measuring password strength by simulating password-cracking algorithms. pages 523–537, 05 2012.

[15] B. Ross, C. Jackson, N. Miyake, D. Boneh, and J. Mitchell. A browser plug-in solution to the unique password problem. 01 2005.

[16] D. Silver, S. Jana, D. Boneh, E. Chen, and C. Jackson. Password managers: Attacks and defenses. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 449–464, San Diego, CA, Aug. 2014. USENIX Association.