Publishable Humanly Usable Secure Password Creation Schemas*

Manuel Blum and Santosh Vempala

CMU, Pittsburgh PA 15213 and Georgia Tech, Atlanta GA 30332
mblum@cs.cmu.edu, vempala@cc.gatech.edu

Abstract

What can a human compute in his/her head that a powerful adversary cannot infer? To answer this question, we define a model of human computation and a measure of security. Then, motivated by the special case of password creation, we propose a collection of well-defined password-generation methods. We show that our password generation methods are humanly computable and, to a well-defined extent, machine uncrackable. For the proof of security, we posit that password generation methods are public, but that the human's privately chosen seed is not, and that the adversary will have observed only a few input-output pairs. Besides the application to password generation, our proposed Human Usability Model (HUM) will have other applications.

Introduction

Passwords are maps from *challenges* (e.g., website names) to *responses* (strings of characters). The ideal password is hard to crack, i.e., an adversary that knows the password generation method, but not the human's privately chosen seed, and that knows only a few challenge-response pairs, will ***provably*** (as opposed to probably) have little clue about the response to a new challenge. Even the knowledge of a half dozen challenge-response pairs will give a combination supercomputer-human adversary little chance to respond correctly.

Constructing a password is a trivial problem for a computer. All it needs to do is store a table containing a completely random string for each challenge of interest. It can then respond to any challenge by table look-up. In this case, an adversary who knows all but one challenge-response pair has negligible information for responding to the missing challenge.

But this does not work for humans [Shay et al., 2014]. Indeed, people commonly use passwords that are too simple [Bonneau, 2012; Cranor, 2014] and/or they use the same password for many different websites [Ives et al., 2004] and/or they use password vaults that, if cracked, expose all their passwords [Li et al., 2014]; see [Blocki, 2014] for a detailed discussion and references. People are mostly unwilling to make the effort to generate and memorize random responses to 100 challenges, so this password generation method is ***not*** humanly usable. The motivating problem for this paper is the following:

> Are there any *humanly usable, secure* and *publishable* password generation methods?

Informally, human usability has two requirements:

Preprocessing Time (PT1): Any initial long-term memorization should take at most 1 hour, preferably less than 20 minutes; future rehearsals should take at most 2 minutes each for a total of no more than 1 hour over the user's lifetime.

Processing Time (PT2): Generation of a 10-character password should be done entirely in the user's head (without the use of paper, pencil, phone), and should take at most 30 seconds, preferably less than 20 seconds.

With these stringent requirements, what can a human do? What level of security can be achieved for password generation?

In the next section, we formally define the space of allowable password schemas. Following that, we give several candidate schemas. Then we introduce our Human Usability Model (HUM) to define and quantify the space of *human* algorithms. The subsequent section defines security parameters. Armed with these measures, we analyze the proposed schemas. This will let us state our (existence) theorem for good password schemas. As a note of caution, we also demonstrate a natural schema that is not humanly usable (an impossibility theorem).

Password schemas: definitions and desiderata

Passwords are maps from challenges to responses. We also use the word **password** (as is commonly done) to denote the response to a challenge[1]. With this definition, a challenge is typically a website name, e.g. AMAZON,

* PHUSPCS, pronounced "phoospics" as in "toothpicks".

[1] Hopefully, this dual use of the word will not be cause for confusion.

AMEX, EBAY; the associated response is the password. To login to a website such as AMAZON, for example, the user must enter -- besides the user's login name -- the password response to the challenge AMAZON.

A **password schema** consists of a sample space of allowable challenges called the **dictionary** and a **set of instructions** for transforming challenges in the dictionary into passwords. That set of instructions typically has two stages, of which the second is the **processing stage** for transforming challenges into passwords (about which we say more later). The first is the **preprocessing stage**, which explains:

1. what must be memorized (e.g., a random map from 26 characters to 10 digits), and
2. what operations are to be performed on the challenge (e.g., a direct map from the characters in the challenge to the corresponding digits in the response) to create the password.

Both 1 and 2 above must include explicit directions for memorizing and for operating on challenges so that a human can do these in a well-defined acceptable amount of time.

The goal is to have a schema for creating and generating passwords that is 1) *humanly usable,* 2) *self-rehearsing,* 3) *secure,* 4) *publishable* and 5) *analyzable.* Here are the meanings of these terms, in reverse order:

Analyzable means that the schema is so explicitly well-defined that a Turing Machine, in practice a computer without access to the web, can follow its directions. By comparison, most currently recommended schemas for constructing passwords are "soft": they are not algorithms. "Take a sentence and concatenate the first letters of all words to create the password," while good advice as far as it goes, invariably does not say how the sentence is to be chosen; on the rare occasions when it does say how, it does not say it in a way that a Turing Machine can generate the password. Precise (Turing Machine) algorithms are necessary for security and human usability to be analyzed.

Publishable means that the schema for generating and recalling passwords is publicly available for everyone to know. In particular, the adversary knows exactly how the user creates responses (passwords) to challenges (website names); the only unknown to the adversary is the particular random choices made by the user in the preprocessing stage. This selection is private (to the user).

Note that analyzable and publishable are not the same: many schemas have been published that are not sufficiently precise to be analyzable. Schemas that are analyzable are not in general published.

Secure means that an adversary cannot generate a user's password to a new website, a website whose user password she has not seen.

Security for passwords schemas is measured very differently from security for most (other) cryptographic schemas. With most cryptographic schemas, security can be based on the fact that the adversary has limited, typically polynomial time bounded, computational power. In password schemas the adversary is a human with a supercomputer, while the user is a human without a computer. As the adversary has vastly greater power than the human user, proofs of security are generally information theoretic: the adversary is assumed to be capable of computing anything computable.

In this work, complexity considerations do arise, but only in the rather severe limits they place on the user. Even finite automata are too powerful to model the computations that a human can do in his head. The only satisfactory proof that a human can compute something in a given amount of time is to see a human do it. Equivalently, we can specify the steps that the human must perform to generate passwords, estimate the time for each, then put this all together to estimate the time it will take the human to do his part.

Self-rehearsing means that in the process of computing passwords for frequently entered websites, the user gets all the practice he needs to compute passwords for infrequently entered websites. In other words, computing the passwords for commonly visited sites rehearses both the algorithm (schema) and the maps that are used to construct the passwords for the other sites.

In our experience, at typical rates of password generation, all 22 letters except Z, X, Q, J are rehearsed sufficiently often to be remembered without additional rehearsals. These are the self-rehearsed letters. As for the Z, X, Q, J maps above, they can either be deliberately rehearsed, or all have the same map. In addition, only lines of code in the password algorithm that are used for the most frequently visited websites will be rehearsed often enough to be remembered without additional rehearsals.

Humanly usable means that a human can learn the schema, then use it to generate passwords, and that all this can be done satisfying certain time requirements (given earlier) for preprocessing and processing. In a later section, we will present a Human Usability Model. It defines a set of operations that humans can perform and gives measures for the costs of these operations.

The Dictionary: A password-generation schema must define the sample space of (allowable) challenges. Examples include:

1. All 3-letter strings of letters, with every 3-letter string equally likely.
2. A standard English dictionary of 10,000 or so words, each with its probability of occurrence in ordinary English text.

3. The top 500 currently most popular websites, each with its probability of being logged into.

Sample schemas

Most of our schemas will involve implicit or explicit character-to-character maps. We write letters of the challenge as capitals A, B, C… and their map values (A)f, (B)f, (C)f… as lower-case letters (A)f=a, (B)f=b, (C)f=c…. We use (A)f instead of f(A) because human computation requires first retrieving A and then applying f.

Our first set of schemas use a letter-to-digit map.

Digit Schema 1 (DS1)

◆**Preprocessing:** Memorize a random letter-to-digit map f (we discuss the important issue of how to do this later).

◆**Processing:** Apply the map to the distinct letters of the challenge in the order they appear.

Example: f(AMAZON) = (A)f (M)f(Z)f (O)f (N)f = amzon (Note: the lower-case letters represent digits; we write f after the letter it is applied to; we skipped the second A).

Caution: short challenges, or ones with many repeated letters will lead to short responses, e.g., f(AAA) = (A)f = a. This is not a favorite schema.

In what follows, we use $+$ and $\bullet$ to denote addition and multiplication mod 10.

Digit Schema 2 (DS2)

◆**Preprocessing:** Memorize a letter-to-digit map f.

◆**Processing:**

For a challenge $A_0A_1 \dots A_{n-1}$,

1. Output $b_0 = (A_{n-1})f + (A_0)f$
2. For $i = 1 \dots n-1$:
   Output $b_i = b_{i-1} + (A_i)f$

Remark 1: A variant of the above schema is to use a random-looking starting point for processing. This can be done by first computing a starting location $start = (A_0)f + (A_{n-1})f$ and starting at that location with $b_0 = A_{start-1} + A_{start}$, then wrapping around the challenge when needed to get $b_1 \dots b_{n-1}$.

Digit Schema 3 (DS3)

◆**Preprocessing:** Memorize a random letter-to-digit map f and a random digit permutation g.

◆**Processing:** For a challenge $A_0A_1 \dots A_{n-1}$,

1. Output $b_0 = ((A_{n-1} + A_0)f)g$
2. For $i = 1 \dots n-1$:
   Output $b_i = (b_{i-1} + (A_i)f)g$

The next few schemas do not use any arithmetic.

Word Schema 1 (WS1)

◆**Preprocessing:** Memorize a letter-to-word map, i.e., pick a word for each starting letter A-Z. For this we suggest that the user first choose a topic (e.g., "Animals"), then generate a list of 10 or more words from that topic for each starting letter (using the internet if he wishes) picking one of those 10 words at random as his map value. If he cannot find 10 words for some letters, he can widen the topic (e.g., "Animals and Birds"). For rare letters such as Q and X, he can simply pick a random word. Let f map a letter to the first TWO consonants in the word following the starting letter. E.g., if A ➔ AARDVARK, then (A)f=RD.

◆**Processing:** Given a challenge, apply the map f to the first k distinct letters of the challenge, or to all letters of the challenge if it is of length less than k. We suggest k=4.

Example: Suppose

A ➔ AARDVARK
M ➔ MONGOOSE
Z ➔ ZORILLO
O ➔ ORANGUTAN

Then, f(AMAZON) = (A)f (M)f (Z)f (O)f=RDNGRLRN.

Letter Permutation Schema 1 (LP1)

◆**Preprocessing:** Pick a *random* permutation of the 26 letters A-Z, and memorize the sequence. For this we suggest the user recite the permutation to a tune, e.g., the standard alphabet song:

ABCD EFG
HIJK LMNOP
QRS and TUV
W and X, Y and Z…

For example, if the random permutation is XBUDVOWTNRPGAHFJSZYLIMQKEC, then the tune could go:

XBUD VOW
TNRP GAHFJ
SZY and LIM
Q and K, E and C…

Let f map a letter to the next letter in this sequence, with the last mapping to the first.

◆**Processing:** Given a challenge, apply the map f to each distinct character of the challenge.

Note: The user does not have to make a pass through the challenge to determine which letters have not occurred earlier. On challenge MAMAMIA, for example, the user will recall, on the second and third occurrences of M and A, that he already printed (M)f and (A)f respectively.

Caution: It is advised that each challenge in the user's dictionary should contain at least 4 distinct characters.

Remark 2: For the above schemas that output characters, a random-looking start could be obtained by fixing a rule such as: "start one past the second vowel" or "start at the vowel before the last consonant".

Remark 3: Many real-life websites require that passwords contain special characters and/or both capital and lower-case letters. To comply with this, we suggest that the user employ a fixed rule for all passwords, such as: "capitalize

the first letter and terminate the password with @1". The purpose of this is not to increase the security but to have self-rehearsing methods for making passwords legal.

Letter Permutation Schema 2 (LP2)

◆**Preprocessing:** Pick a random word with a great many different letters. In a table of the 26 letters, mark off those letters that appear in the word. Pick another random word that uses many unmarked letters (and possibly some marked ones) and mark off those letters in the table. Pick a third word that uses many unmarked letters and mark those letters off the table. Memorize the three words in the string word1-word2-word3, and a final string4 consisting of letters (if any) that do not appear in any word, e.g., QXZ if all letters except Q, X and Z appear in the three chosen words. Let f map each letter in the challenge to whichever consonant follows its first occurrence in the sequence word1-word2-word3-string4.

◆**Processing:** Apply f to the distinct letters of the challenge. If a letter is a repeat, map it to the next letter in the sequence, wrapping around the unused letters if needed.

Example: ANTIVIRAL-MOHAWKBUG-ESCAPED are the three words and FJQXYZ is string4. Then, f(AMAZON) = (A)f(M)f((A)f)f(Z)f (O)f(N)f = NHTFHT.

The above schemas can be extended to multiple maps (letter-to-digit or letter-to-letter), allowing for proportionally more security, needing proportionally more preprocessing, but only slightly more processing. We call these *multi-map schemas*, and provide one illustration.

[MMS1]

◆**Preprocessing.** Memorize m letter-to-letter maps, $f_0, f_1, .., f_{m-1}$.

◆**Processing.** Given a challenge of length n, compute t = n (mod m), apply the t[th] map f_t to the challenge.

Remark 4: There are several ways the user might choose a meta-map to decide which of his maps to apply to a challenge. E.g., he could use a letter-to-digit map and compute $t = (A_{n-1})f + (A_0)f$ to determine which map to use.

All the above schemas can be applied by making one or at most two passes over the challenge. In contrast, the following schema appears to require substantially more memorization and far too many passes.

Random Matrix Schema (RM)

◆**Preprocessing:** Memorize an n x n matrix M of randomly chosen digits and a random letter-to-digit map f.

◆**Processing:** given a challenge $A_0A_1 \ldots A_{n-1}$,

for $i = 0 \ldots n-1$:

Output $b_i = \sum_{j=0}^{n-1}(A_j)f \bullet M(i,j)$.

The Human Usability Model (HUM)

We model the human as a finite state machine with both long-term and short-term memory. The long-term memory is written in the **preprocessing** stage, when the human stores the function(s) that must be memorized. That long-term memory is used in read-only mode together with a tiny, short-term, read-write memory in the **processing** stage to transform any given challenge to its response. The state diagram of the machine that does this, i.e., accesses both types of memory in order to transform challenge into response, is described in part in the preprocessing stage and in part in the processing stage of each password schema. The state diagram must be learnable in minutes, not hours, from a description of the code and a few examples. It must also be completely self-rehearsing in that every line of code defining that state diagram must be used in a substantial fraction (quarter? half? the larger the fraction, the more self-rehearsing is the code) of challenge-response computations. A password schema that chooses its starting location in the challenge by randomly hashing the first letter of the challenge to the start location would not be self-rehearsing and therefore would not be permissible.

More on Long- and Short-term Memory

For the purpose of transforming challenges into responses, human memory consists of a large long-term relatively permanent memory and a small short-term working memory.

1. Long-term memory is unbounded except for

- the (substantial, proportional) time and effort needed to acquire it (up to a concentrated hour or so),
- the relatively small time and effort (in spaced rehearsals) required to maintain it (measured in minutes over a lifetime), and
- the length of one's (mentally healthy) life.

2. Long-term memory (for our purpose) consists of sequences of text. The storage of these texts may be based on visual, auditory, sensory, motor, experiential and other memories, all of which can play a role in computing and remembering passwords. Since this work is concerned with transforming challenge strings into response strings, only the memory required to store strings of characters (rather than, say, visual images), and the rules/password schemas for transforming them are described here.

3. Just as humans typically store their memory of the alphabet A, B… Z in a singly-linked list, which can typically only be recited left-to-right from the starting point and a few other entry points in the list, the input challenge words are streamed into the machine looking much like a singly-linked list in long-term memory. The human has direct

access to the input characters in the order they arrive. Short-term memory enables going back at most one character. The human also has pointer access to the first, second, last and last-but-one characters of the input sequence.

4. Working memory is severely bounded. In password computations, it typically consists of 2 characters (2 digits, 2 letters, or 1 letter and 1 digit) and a pointer into one or at most two sequences, these being typically the challenge itself and at most one additional sequence.

5. Working memory fades quickly. Items in working memory that are needed but not recalled for more than 2 or 3 operations must be reconstructed.

We are now ready to define our measure of the cost (or perhaps more precisely, the effort) of human computation.

> HUM Processing Complexity = total number of reads from and writes to working memory.

The total combined Preprocessing Complexity (PC1) and Processing Complexity (PC2) is a vector: HUM = <PC1, PC2>.

Complexity of some natural operations

- Working memory for letters and digits is a stack of two elements (letters or digits). The cost of accessing the top (most recent) entry is 1, and that of accessing the bottom entry is 2.
- Retrieving a sequence from recently retrieved long-term memory, i.e., retrieving a pointer to the beginning of the sequence, has Cost=1.
- Following a pointer into a recently-retrieved sequence in long-term memory, moving it 1 step to the right, resetting the pointer to start/start+1 or to end/end-1 has Cost=1.
- Operations of =, + and x (mod 2,3,4,5,9,10) on two single-digit numbers. Cost = number of digits created (not necessarily all written) during the operation.

Example: 4+3 (mod 10) = 7 has cost 1; 4+9 (mod 10) = 3 has cost 2.

- Operations of =, + and x (mod 11) on two single-digit numbers has Cost = number of digits created (not necessarily written) before applying the mod operation + 1 for the mod operation on numbers > 11.

Example: 4+3 (mod 11) has cost 1; 4+9 (mod 11) has cost 3.

- Operations of =, + and x (mod 2,3,4,5,9) on two single-digit numbers are treated similarly.
- Applying a map such as a character-to-digit map is the same as following a pointer and has Cost =1.

Remark 1: The HUM only indicates proportional effort for a human, the actual time will vary with the human. This is analogous to asymptotic complexity, which counts the number of operations, while the running time depends on the machine executing the algorithm and its current load. An important difference between human and machine computation counts is that, since challenges typically have length $n \approx 12$ or less, care must be taken to estimate all constants, not just accept their big **O** or big **Ω** dependence on n.

Remark 2: A major reason for specifying the operations and their costs is to determine, albeit roughly, the expected human usability of proposed password schemas. These computed values should be confirmable experimentally.

Cost of memorization

How are we to measure the cost of preprocessing, i.e. long-term memorization? First, fix an upper bound on how much one can memorize in a single sitting and how many sittings one is willing to do per day. While this can vary from person to person, most people can memorize one 7-digit phone number in a single sitting. An efficient way to retain the number in long-term memory is the following: taking T = 15 minutes, a phone number memorized at time t should be rehearsed at times t+T, t+2T, t+4T, t+8T,…, a roughly "doubling" schedule [Pimsleur 1967; Woźniak & Gorzelańczyk 1994]. These purposeful rehearsals of a phone number can stop once the required time-between-rehearsals exceeds the expected time-between-successive-uses of the phone number.

Suppose the cost of memorizing one 7-digit phone number is C, and the cost of rehearsing a 7-digit phone number is c, $c \ll C$. Then we observe that n phone numbers can be memorized at a cost[2] of $Cn + cn \cdot (\log n)$. If no more than n phone numbers are ever memorized, the next sitting will require just $c \cdot (\log n)$ time, a big drop from $(C + c \cdot (\log n))$, and this $c \cdot (\log n)$ cost will then decrease slowly to zero.

Security measures

The human usability of a password schema is measured by its preprocessing and postprocessing (processing) complexities **PC1** and **PC2**. We define additional measures **Q** and **K** to evaluate the security of password schema.

1. Quality **Q** is formally defined by the password game below. Informally, Q is the expected number of randomly chosen challenges and associated responses that an adversary must see in order to have sufficient information to

[2] To see this, suppose a new phone number is memorized in each of sittings k = 1…n, and that in sitting k in which we memorize the k^{th} telephone number, that sitting's work also includes the log k rehearsals, each costing c, sufficient to retain all previously memorized phone numbers. The cost for this k^{th} sitting will be $C + c \cdot (\log k)$. That is at most $Cn + cn \cdot \log n$ total for all n phone numbers, reckoned on the sitting in which we memorize our last (n^{th}) phone number.

generate the correct response in one of at most 10 tries to the next randomly chosen challenge. Our Q's are typically in the range of 6 to 10. By comparison, cryptographic zero–knowledge protocols have $Q = 10^{n-1}$ for integer passwords of length n.

2. Exponent **K** is an integer defining a probability, typically $1/10^K$ for an integer response to a challenge and $1/36^K$ for a response consisting of letters and digits. This $1/10^K$ is the probability that an adversary (Alice) can guess the correct response to a new challenge once she has seen (in previously observed challenge-response pairs) all but one of the characters in the current challenge. For example, suppose the schema maps letter strings (challenges) of length n to digit strings (responses) of length 2n by hashing each letter in the challenge to a pair of randomly chosen digits. Further suppose that on challenge EBAY, the adversary has previously seen E, B, A, but not Y. Then K = 2, indicating that the correct response to EBAY is any one of $10^K = 100$ equally likely possibilities.

The Password Game (for determining Q)

Recall that the design of a Password schema includes in its specification a sample space of permissible challenges, the dictionary. The game is played between the User (he) of the schema and the Adversary (she) under the watchful eye of a trusted Judge. There are no winners or losers, just a determination of Q.

Until the Adversary's 10 responses to a challenge is guaranteed to contain the correct one:

- The Judge selects a challenge at random (with repetition) from the dictionary.
- User and Adversary independently and privately supply responses to the Judge. Here,
  1. the User supplies the unique correct response to the judge;
  2. the Adversary (without seeing the User's response) guesses/computes 10 possible responses, which she supplies to the Judge.
  3. If none of the 10 responses are correct, then the Judge gives the adversary the correct response.

End Until

Return the expected number of challenges, **Q**, up to and including the one on which the Adversary supplied a correct response.

Analysis of human usability and security for select schemas

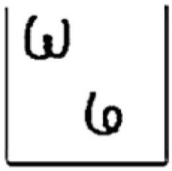
[DS1]

Usability.

Postprocessing: This schema requires the user to apply his map to the first letter of the challenge, output the value, and shift his "pointer" to the next letter of the challenge, each operation costing 1 unit in our model. Thus, for an n-letter challenge, the Human Usability Measure is HUM = #(apply map, output, shift pointer in challenge)·n = 3n, where #(x, y, z) denotes the number of steps to perform operations x, y, z in this order. The average time it takes to do this is about n seconds on an n letter challenge. This comes to 10 seconds for a 10-letter challenge, which is well within the desired 20 seconds (and the required 1 minute) per password to satisfy requirement PT2.

Preprocessing: The preprocessing complexity for memorizing a letter-to-digit map, while significant, can be made less than 1 hour by use of an appropriate memorization technique. Specifying a technique that upper bounds the human's memorization or computation time is an important aspect of human computability, akin to specifying an algorithm to achieve a certain run time.

After the user has tossed a 10-sided die 26 times to create a random map from 26 letters to 10 digits – an amount of time that we do not count toward our maximum 3-hour requirement – he can memorize his map. This he does by using a 26 row x 10 column table in which each entry prescribes a visual way to memorize the associated letter to digit map. For example, row W column 6 shows how to remember that the chosen map takes W ➔ 6:



The authors of this paper can do an initial memorization (up to the first successful recall of the entire map) in a concentrated 15-30 minutes using this table. Once the user has completed an initial memorization of the material, he can set a timer to rehearse his map using the [Pimsleur, 1967; Woźniak & Gorzelańczyk, 1994; Blocki et al., 2013] recommended doubling schedule, which rehearses at intervals of 15 min, 30 min, 1 hr, 2 hrs, 4 hrs etc. The first day is hell, but after that the rehearsals are short and (relatively) painless. Assuming each rehearsal takes 2 minutes, the total time spent over the user's lifetime is at most an additional 45 minutes, which comes to a total of one hour. This satisfies the PT1 time requirement.

Security. The adversary can respond correctly to a challenge only if she has seen all letters in the challenge in previous challenges. If she hasn't seen even one letter, the chance of guessing the response to a single letter of the challenge is at most 1/10, thus K=1. The quality measure Q will be the expected number of challenge-response pairs up to (and including) the first challenge all of whose letters have appeared in earlier challenges. This value depends on the dictionary. The table below shows the Q value (expec-

tation), its standard deviation and its 90% value (i.e., the length of challenge at which the probability that Q is at least this value is at least 0.9) for three dictionaries: a large English dictionary, the top 500 domain names and random 7-letter strings. All estimates of Q were done using 100,000 trials with repetition.

Dictionary	Q	STD	90%-value
English 50K	6.23	1.80	4
Top 500 domains	6.60	2.03	4
Random 7-letter	7.54	1.58	6

Table 1. Q values

For a dictionary of random strings, Q depends only slightly on the length of the challenge, as shown in the next table.

1	2	3	4	5	6	7	8	9	10
7.09	8.89	9.12	8.82	8.39	7.96	7.54	7.12	6.97	6.91

Table 2. Q for random strings

[DS2]

Usability. Preprocessing for this schema is the same as [DS1], requiring the user to memorize a letter-to-digit map. **Processing** involves applying this map, adding mod 10, outputting and shifting pointer. More precisely,

HUM = go to last letter, apply map, go to first letter, apply map, add mod 10, output; then repeat n-1 times: (apply map, add mod 10, output, shift pointer)

= 6.5 + (n-1)(4.5) = 4.5n+2.

Here we have counted 1.5 per addition mod 10, since some additions create 2 digits while others create one, and both types are of the same number in expectation.

Security. From knowledge of the schema and a challenge-response pair, the adversary can recover all the map values of letters in that challenge: $f(A_i) = b_i - b_{i-1}$. Thus the parameters K and Q are the same for [DS2] as for [DS1].

[DS3]

Usability. **Preprocessing**: This schema needs the user to memorize both a letter-to-digit map as before (15-30 min up front, 1 hour total over the user's lifetime), and also a random permutation on the digits 0-9. He does this by choosing a random cyclic permutation of the digits, i.e., a single 10-digit number with no repeated digits. The map takes each digit to the next digit, and the last digit to the first. This is as hard as memorizing a single 10-digit phone number, and takes 2 minutes up front and less than 10 min over the course of the user's lifetime. We note that with only the 10-digit number memorized, access to an input digit takes longer initially – about 5 steps on average. The user has to scan multiple digits to find the right one. Each time he does this, however, he rehearses his map: this will soon give random access to the digit map.

The **processing** complexity is HUM = #(go to last letter, apply f, go to first letter, apply f, add mod 10, switch to g, apply g, output) + #(select next letter, apply f, add mod 10, switch to g, apply g, output, shift pointer)·(n-1)

= 8.5 + (n-1)(7.5) = 7.5n+1.

The "switch to g" is for computing g after the user has memorized the map as a linked list but not yet as a random access hash. Its cost would initially be 5 but would drop eventually to 1.

Security. A computationally unbounded adversary can consider all possible cycles on 10 digits (9! =362,880 of them), and decode the user's maps as follows: whenever she notices an inconsistency in the values of f for a particular g, that g is eliminated. The quality Q is the number of challenges up to and including those for which the surviving f, g pairs all give the same response. In our evaluations, the Q for [DS3] is about the same as for [DS1] and [DS2]. Nevertheless, [DS3] provides more security in at least one sense: the adversary needs to see multiple challenge-response pairs before she can determine the value of f for even a single letter.

[WS1]

Usability. Preprocessing: The user has to create a map from letters to words of a topic, perhaps aided by a word generator, and then rehearse his map. Creating the map takes 30mins or so, but after that, the map is surprisingly easy to rehearse and to remember since the starting letter together with the topic gives a strong clue to the word. Budgeting 2min per rehearsal, with spaced repetition at doubling intervals, we get a total of at most 45min over the user's lifetime.

Processing: Recalling that we process k letters of the challenge and output 2k letters, the processing complexity is HUM = #(apply f, output two consonants, shift pointer on the challenge)·k = 4k = 2n. This is the lowest complexity per output character of all the schemas we discuss.

Security. As in the case of [DS1], an adversary has to have seen each of the first k characters of a challenge in previous challenges to be able to respond correctly. However, each challenge-response pair reveals only k values, where k is typically less than the length of the challenge. Using k=4 (responses of length 8) gives a significantly higher value of the quality Q compared to [DS1, DS2, DS3] that use the entire challenge (Table 1).

Dictionary	Q	STD	90%-value
English 50K	7.25	2.02	4
Top 500 domains	7.56	2.24	4
Random 4-letter	8.82	2.21	6

Table 3. Q values for 4-letter challenges

The security parameter K is between 1 and 3. If there is one letter in the new challenge that the adversary has not seen, he has to map it to two letters. If these two letters were both completely random then his chance would be only $1/21^2 = 1/441 < 1/10^{2.6}$. However, since we output consecutive consonants of a word, and consonants are not equally likely, the true probability is a bit higher.

[LP1]

Usability. **Preprocessing:** Memorizing a random letter permutation by singing it to a tune takes 5 mins initially and 1 min per rehearsal, giving a total of 20 additional minutes over the user's lifetime.

Processing.

HUM = #(apply map, output, shift pointer)·n = 3n.

We remark that while we have charged a cost of 1 for applying the "next-letter" map, this could initially take significantly longer, as the user would have to recite (sing) an expected 13 letters to compute his hash. With time, however, he will get faster. This schema will be truly self-rehearsing if and only if the user recites the entire permutation each time he generates a password.

Security. The quality Q is the same as in previous schemas that use all n letters of a challenge (Table 1).

[LP2]

Usability. This schema has the distinct advantage of very low preprocessing complexity --- once the user has chosen his three words, he has practically got them already memorized! The processing complexity is similar to LP1, converging to HUM = 3n, after sufficient practice.

Security. The quality Q is potentially lower, since an adversary who knows the schema, after seeing some letters, could try to guess the user's words, which come from a significantly smaller set of possibilities than all letter permutations.

[RM]

Usability. This schema, introduced primarily to demonstrate the tradeoff between usability and security, requires a huge **preprocessing** effort, random access memorization of an n x n matrix of random digits, for n about 8, 9, 10, with the user learning a pointer to a value in each position (i,j) of the matrix. Unlike all the previous schemas, which we have successfully implemented on ourselves, we have not done or even tried to do this one. We do believe it is possible, given routine memory feats such as learning the digits of π up to hundreds of digits. The **processing** complexity, at least for the natural human algorithm, can be computed as follows: for each output digit, make one pass over the challenge: apply the matrix map, apply the letter-to-digit map, multiply mod 10, add mod 10, then shift the pointer into the challenge. Thus, for an n-digit output,

HUM = #((apply M, retrieve f, apply f, multiply, add, shift pointer)·n, output, shift M pointer)·n

$= n(7n + 2) = 7n^2 + 2n$.

Here we counted 2 per multiplication since multiplying two single digits typically results in two digits. This implementation uses n passes over a challenge of length n. The resulting processing complexity is the highest of all the schemas we have considered. Is there a faster human algorithm? In a later section, we will prove that the number of passes needed by *any* human algorithm grows linearly with the length of the challenge.

Security. The adversary has to know the entire matrix M and the user's map on the letters of the next challenge, to be able to output the correct response with probability more than 1/10. Each challenge-response pair gives n equations. Thus Q is at least n. An unbounded adversary could keep track of all 10^{26} possible letter-to-digit maps f, and eliminate them as she finds inconsistencies. A rough lower estimate of Q is n + (26/n) = 11.25 for n=8.

[Multi-map schemas]

Usability. The preprocessing complexity is m times the preprocessing complexity of a single map. For letter-to-digit maps he can memorize a single map from letters to m-digit numbers. For letter-to-word maps, he can pick different topics for each map.

While processing, first the user has to compute the map number t. Then HUM = n+1 if he uses $t = n\ (mod\ m)$. If he uses $t = f(A_0) + f(A_{n-1})$, then HUM=5 for identifying the map. The remaining processing cost is that for a single map schema. For example, for [DS1], whose processing cost for a single map is 3n, the processing cost of a multi-map schema would be 3n+5 with the latter rule for t.

Security. This schema has a Q with expected value that is nearly m times the Q-value for a single map. To see this, suppose for argument that it takes exactly Q challenge-response pairs for a single map. If we think of the k maps as k bins and each challenge as a ball thrown into a random bin, then the game stops when one of the bins reaches Q balls. With k bins, we stop as soon as one of them reaches Q balls, which will happen slightly before all k of them do.

Schema	Preprocessing	Processing HUM	Q
DS1	15+45	3n	6-7
DS2	15+45	4.5n+2	6-7
DS3	17+55	7.5n+1	6-7
WS1	30+45	2n (=16)	7-8
LP1	5+20	3n	6-7
LP2	1+1	3n (eventually)	3-4
MWS1	m(30+45)	3n+5	6·m
RM	??	$7n^2 + 2n$	11

Table 4. Comparison of Schemas

The ultimate proof that a schema is humanly usable is to see a human do it within the prescribed constraints. To do this, and to validate the HUM, we tested ourselves on most of these schemas. These results are in the tables and charts below. Just as a computer's timing depends on what other processes it is executing, a human's speed also depends on several factors such as energy level, distraction etc. To avoid these confounds, we timed for all the schemas in one sitting, using 11 challenges in random order.

	N	DS1	DS2	DS3	WS1	LP1	LP2
CMU	3	3	6	16	5	5	6
APPLE	5	4	6	22	6	10	8
GMAIL	5	4	9	19	6	8	18
DELTA	5	4	6	14	6	7	19
CHASE	5	5	7	16	6	14	15
AMAZON	6	7	10	24	7	15	17
GATECH	6	6	8	23	6	14	24
PAYPAL	6	5	8	20	7	9	17
DROPBOX	7	8	11	22	6	14	18
PNCBANK	7	6	13	25	7	18	21
WELLFARGO	10	10	18	29	5	17	35

Table 5. Timings for H2

We compared these observed timings to their HUM values by plotting the data for [DS1], [DS2], [DS3] and [WS1] against the length of the challenge (Figure 1). The HUM for [WS1] is 8 for a challenge with at least k=4 distinct letters, i.e., constant.

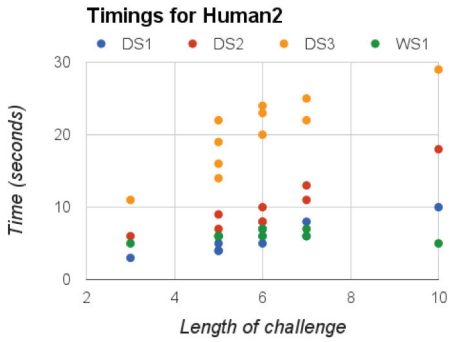


Figure 1. Timing comparison in one sitting

Using the same challenges, we timed 8 other human subjects on [DS1] and [LP1], with each schema timed in a separate sitting. The average timings are shown in Figure 2. The two schemas have comparable timings, consistent with their HUM values. We expect them to get even closer as the users rehearse their letter permutations.

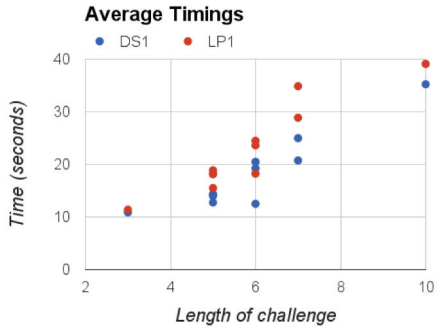


Figure 2. Average times of 8 participants

A human password schema theorem…

Based on the previous section, we assert the following.

THEOREM 1. There exist publishable PASSWORD SCHEMAS that are
1. WELL-DEFINED (i.e., implementable unambiguously on a Turing machine),
2. MACHINE UNCRACKABLE to a well-defined extent by unbounded Turing machines (i.e., information- theoretically, based on the Q value), and
3. HUMANLY USABLE as witnessed by a feasible human algorithm with a bounded HUM, and demonstrated by at least one normal dedicated human being with at most 1 hour of preprocessing, and at most 30 seconds of processing (i.e., shown empirically).

…and an impossibility theorem

Here we show that any human algorithm that implements the [RM] schema has to make a prohibitively large number of passes over the challenge, even for some simple M.

THEOREM 2. There exist $n \times n$ matrices M such that for any $k \geq 1$, any algorithm that computes $C \cdot M$ by making at most k passes over any given n-digit challenge C must use a working memory of size at least n/k digits.

The theorem is tight: we can compute n/k digits of output in each pass using k digits of working memory. However, since the working memory allowed in our human usability model is a small constant (2 digits), at least $n/2$ passes are needed to compute $C \cdot M$ for a challenge of length n.

Proof of Thm 2. Let M be the matrix that "reverses" challenge X, i.e., $Y = X \cdot M = (X_n, X_{n-1}, \ldots, X_1)$. First we prove the bound for any 1-pass algorithm. A 1-pass algorithm that scans X only once cannot output Y_1 till it sees the last digit of X. Consider the working memory contents S at that point. The algorithm is able to compute Y from just S and X_n. From Y we can reconstruct X since M is full rank.

So S and X_n together determine X, which means S must have at least n-1 digits, and along with reading X_n (which is essential), this is a total of n digits of memory.

To analyze multi-pass algorithms, we take the challenge X to be a random digit sequence. For a proof by contradiction, suppose that the algorithm uses only s digits of memory for $s < n/k$. Then by the above argument, after one pass, the algorithm can output at most s digits of Y, and these must be a subset of the s-prefix of Y. In the second pass, consider the point at which the digit Y_{s+1} is output. After this, due to the space restriction, at most s-1 digits can be output in the second pass, so at most a 2s-long prefix of Y is output after two passes. This conclusion clearly holds if Y_{s+1} is not output in the second pass. We repeat this argument, using $Y_{(i-1)s+1}$ in the i^{th} pass, concluding that k+1 passes are needed. ■

We remark that for a random matrix M, the above proof technique shows that with probability at least 9/10 (over the choice of M), any 1-pass algorithm needs memory of size at least n. This is because the last entry of the first row is nonzero with probability 9/10; in this case, the algorithm can start its output only after reading the last digit, at which point it needs as much memory as its total output.

This lower bound technique applies to many natural matrices. For other matrices, a human algorithm making only one or two passes can compute the product $C \cdot M$. E.g., DS2 uses the matrix with 1's on and below the diagonal and zeros above it. This raises two questions for future research: for which matrices can $C \cdot M$ be computed by a human algorithm? Which of these gives the most security?

Further considerations

Using letter frequencies. The frequency of letters in English words is highly skewed, and the first 10 letters by frequency make up more than 70% of all occurrences. Thus, an easier memorization option that works nearly as well as a complete map is to learn a map for just the first ten most frequent letters: E, T, A, O, I, N, S, H, R, D. In the case of a letter-to-digit map, the corresponding digits could be read off as a single phone number. For a letter-to-letter map, we recommend mapping only to consonants, then inserting letters to make the 10 consonants form words and sounds that are more familiar and therefore easier to remember.

Changing passwords. Passwords sometimes have to be changed. How should a user who is using one of the password generation methods described here change his password(s)? The simplest, oft-suggested method to change passwords is to attach an extra character at the beginning or end of the password and cycle through its chosen range of values, e.g., 0,1,2,0,1,2…. This has a serious problem: a new password that differs from the old one in only one digit will not be particularly secure.

A more effective method to change passwords is to set challenge$^+$ equal to one of challenge0, challenge1, … challenge9 (or 0challenge, 1challenge, …). These challenge$^+$s will turn into passwords that **provably** look quite different from each other. This is in fact a desideratum for password generation methods.

Acknowledgements

We are deeply grateful to Jeremiah Blocki, Lenore Blum, Anupam Datta, Adam Kalai, V. V. Rao and Haofeng Zhang for helpful discussions, and to many others for trying out some of these schemas. This work was partially supported by the NSF.

References

Blocki, J. 2014. Usable Human Authentication: A Quantitative Treatment. Ph.D. Thesis. *CMU-CS-14-108.pdf*

Blocki, J., Blum M. & Datta, A. 2013. Naturally Rehearsing Passwords. *ASIACRYPT* (2): 361-380.

Bonneau, J. 2012. The science of guessing: analyzing an anonymized corpus of 70 million passwords. *IEEE Symposium on Security and Privacy (SP),* 538–552.

Bonneau, J., Herley, B., Oorschot, P. C. v., and Stajano, F. The Quest to Replace Passwords: A Framework for Comparative Evaluation of Web Authentication Schemes 2012. *IEEE Symposium on Security and Privacy,* 553 – 567.

Cranor, L.F. What's wrong with your pa$$w0rd? 2014. *TED talk*, http://www.ted.com/talks/lorrie_faith_cranor_what_s_wrong_with_your_pa_w0rd/transcript?language=en

Li, Z., He, W., Akhawe, D. and Song, D. 2014. The Emperor's New Password Manager: Security Analysis of Web-based Password Managers. *USENIX Security 2014:* 465-479.

Ives, B., Walsh, K. R., and Schneider, H. 2004. The Domino Effect of Password Reuse. *Communications of the ACM,* April 2004, 47(4), 75-78.

Kruger, H., Steyn, T., Medlin, B. and Drevin, L. 2008. An empirical assessment of factors impeding effective password management. *Journal of Information Privacy and Security*, 4(4):45–59.

Pimsleur, P. 1967. A Memory Schedule. *The Modern Language Journal*, 51(2), 73—75.

Shay, R., Komanduri, S., Durity, A. L., Huh, P. (S.), Mazurek, M. L., Segreti, S. M., Ur, B., Bauer, L., Christin, N. and Cranor, L. F. 2014. Can long passwords be secure and usable? *CHI 2014:* 2927-2936.

Woźniak, P. A., & Gorzelańczyk, E. J. 1994. Optimization of repetition spacing in the practice of learning. *Acta Neurobiologiae Experimentalis*, 54, 59-62.