The complexity of human computation via a concrete model with an application to passwords

Manuel Blum[a,1] and Santosh Vempala[b]

[a]School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213; and [b]School of Computer Science, Georgia Tech, Atlanta, GA 30306

Contributed by Manuel Blum, February 24, 2020 (sent for review September 10, 2018; reviewed by Gilles Brassard, Whitfield Diffie, and Dawn Song)

What can humans compute in their heads? We are thinking of a variety of cryptographic protocols, games like sudoku, crossword puzzles, speed chess, and so on. For example, can a person compute a function in his or her head so that an eavesdropper with a powerful computer—who sees the responses to random inputs—still cannot infer responses to new inputs? To address such questions, we propose a rigorous model of human computation and associated measures of complexity. We apply the model and measures first and foremost to the problem of 1) humanly computable password generation and then, consider related problems of 2) humanly computable "one-way functions" and 3) humanly computable "pseudorandom generators." The theory of human computability developed here plays by different rules than standard computability; the polynomial vs. exponential time divide of modern computability theory is irrelevant to human computation. In human computability, the step counts for both humans and computers must be more concrete. As an application and running example, password generation schemas are humanly computable algorithms based on private keys. Humanly computable and/or humanly usable mean, roughly speaking, that any human needing—and capable of using—passwords can if sufficiently motivated generate and memorize a secret key in less than 1 h (including all rehearsals) and can subsequently use schema plus key to transform website names (challenges) into passwords (responses) in less than 1 min. Moreover, the schemas have precisely defined measures of security against all adversaries, human and/or machine.

humanly computability | passwords | mental algorithms | pseudorandom generators

The processing power of humans is much more limited than that of computers for simple grade school arithmetic. Although humans can do some amazing highly skilled mental jobs that computers are only now beginning to do, like write gripping stories, state hard mathematically interesting problems, discover laws of nature, etc., they (humans) are limited in their ability to do arithmetic computations in their heads.* In particular, humans have only a tiny short-term memory for performing arithmetic operations. They also do arithmetic operations much more slowly than computers. Modern complexity theory and cryptography are designed for computers (Turing machines) and/or for humans with access to computers but not for humans working alone in their head. This paper is concerned with investigating problems that might possibly be solved by humans working alone—using humanly usable algorithms (single agent) or humanly usable protocols (multiple agents). The broader problem considered here is the following.

The Problem

What functions can a human compute in his or her head that a powerful human–computer adversary cannot break† from observing a specifically limited amount of input–output behavior? Can a human transform (truly) random seeds into "pseudorandom sequences" in his or her head such that the "pseudorandom sequences" are indistinguishable from truly random sequences to a personal computer (PC) that is run for at most 10^{24} steps?‡ In this work, the adversary is presumed to know the publicly available schema, which is the humanly computable algorithm minus the private key, and to observe—possibly participate in—the public communications called for by the protocol (also, we restrict ourselves to classical rather than quantum computers; the latter might eventually require fewer steps).

The Model (Complexity in the Small vs. Asymptotic Complexity)

- Since humans are slow at arithmetic computations, at least in comparison with computers, complexity bounds described by functions such as polynomials or exponentials are not suitable for analyzing human computation and consequently, as we shall see, not appropriate for analyzing Turing machine adversaries either. We must give up thinking that an exponential function like $10^{x/10}$ is worse than a linear function like $10(x + 10)$. Indeed, the entire range of x for human computation may be $0 \le x \le 25$, and in that range, the exponential $10^{x/10}$ is smaller than the linear $10(x + 10)$. This is not a contrived example either: it is easier for a human (or computer) to decide if 91 is prime using an exponential time factoring algorithm than a polynomial time primality test.

Significance

This work presents a concrete, mathematically precise model of what humans can compute in their heads. It thereby provides a method to probe the limits of human computation (can humans generate numbers that look random to a powerful computer?) as well as to design efficient humanly computable mental algorithms for everyday tasks, such as generating passwords and making real-time decisions.

Author contributions: M.B. and S.V. designed research, performed research, and wrote the paper.

Reviewers: G.B., Université de Montréal; W.D., Stanford University; and D.S., University of California, Berkeley.

Competing interest statement: W.D. holds a patent on techniques for remembering cryptographic keys.

[1]To whom correspondence may be addressed. Email: mblum@cs.cmu.edu.

This article contains supporting information online at https://www.pnas.org/lookup/suppl/doi:10.1073/pnas.1801839117/-/DCSupplemental.

First published April 14, 2020.

*Human-oriented schemas can ask humans to use their (currently) distinctly human computing powers to generate passwords. For example, a human might be shown a photograph of a street scene and asked to count the number of humans, dogs, or bicycles in the scene. In this work, we look only at schemas that transform website names (i.e., character strings, not photographs) into passwords. Ref. 1 has schemas that accept pictorial challenges.

†Break = infer, invert . . . depends on context.

‡Why this particular number? The intent here is to have as example a specific memorable number (in this case, Avogadro's number) that is roughly the minimum size sufficient to keep adversaries at bay—adversaries that have a week's time on the fastest supercomputer, which in December 2018, can do 2×10^{17} flops per second and $\sim 12 \times 10^{22}$ flops per week. In general, the specific number will depend on current technology. While we use specific values and give justifications for them everywhere, most, like Avogadro's number, are actually placeholders for more general constants. In general, a cryptographic schema's analysis will have bounds on usability and security. Usability bounds are upper bounds on the maximum amount of work that a human should be expected to do. Security bounds are lower bounds on the minimum amount of work that an adversary must do to "break" the schema.

Therefore, instead of demanding that a schema run in polynomial or linear time, we require an exact value for the number of steps that the schema takes plus or minus a small additive constant.[§]

- Our insistence on dealing with explicit numbers instead of with formulas that are correct in the limit runs completely counter to what is done in complexity theory, but for human computation, this insistence on concrete specific constants rather than asymptotically correct formulas is the right approach. In part, this is because—while computers acquire more memory and run faster with each passing year—human brains remain the same old model (the brain still advances but through improvements in culture).
- In modern complexity theory, there is a sharp divide between polynomial time and exponential time computability. Cryptographic protocols typically assume that both users and adversaries are randomizing poly-time algorithms. Users can encrypt things in poly time that cannot be decrypted by adversaries that run in poly time. In human computation as defined here, the user is a human working without a computer for a specific bounded amount of time measured in hours, minutes, and seconds. The adversary is a human, similarly bounded, having access to a PC that can run for at most 10^{24} steps.

Our first task is to come up with a precise definition for humanly usable algorithm (henceforth called a schema) that can be used, for example, to transform challenges (website names) into responses (passwords), all within the acceptable time limits mentioned above, while remaining secure to a well-defined extent from a computationally all-powerful adversary (a Turing machine with unbounded computation time).[¶] Two other tasks are proposals for humanly computable "one-way functions" (HU-"OWFs") and humanly computable "pseudorandom generators" (HU-"PRGs"). The former are functions that humans can compute in their heads but that an adversary having a PC of clearly specified power (limited to at most 10^{24} steps) almost certainly cannot invert.

To state and verify humanly usable schemas and protocols, we first need a definition of humanly usable. For this, we define a formal model of human computation and propose two complexity measures: 1) the human cost of preprocessing (PREP), which is about memorizing the schema and generating and memorizing a private key if any, and 2) the human cost of processing (PROC), which is about using the schema and key to do the required task, our first such task being to transform challenges into responses.

We discuss three tasks in more detail. In the next section, we present the model of human computation and the associated costs with a few examples.

Password Generation. Passwords are responses to challenges (typically website names). In this paper, passwords are produced by password schemas, which are humanly computable algorithms for mapping (challenge, key) pairs to passwords. The insecurity of commonly used passwords (2–4) and the difficulty of memorizing multiple long passwords (1, 5, 6) have been discussed extensively in the literature. Passwords should be easy to produce when needed and hard for an adversary to forge even if the adversary knows the user's schema and has seen the passwords to a small number of websites. More generally, we seek schemas that are analyzable, publishable, humanly usable, secure, and self-rehearsing. Analyzable means that the schema is so precisely defined that a Turing machine can execute it. Publishable means that the schema itself (but not the user's private key) is or can be made public. Humanly usable means three things: 1) the schema itself (but not the key) must be learnable in a few minutes, 2) generation and memorization of a key should take at most an hour, preferably no more than 30 min, of the user's lifetime, and 3) generating or regenerating a password should take no more than 1 min, preferably 20 s. Regeneration is especially important because passwords in our view should never be memorized but should be regenerated when needed.

A number Q specifies security: a password schema is said to have security Q if an adversary who has seen responses to less than Q challenges—each challenge drawn at random with replacement from a well-defined sample space (a dictionary of words with associated probabilities)—has probability less than 1/10 to guess the correct response to the next challenge randomly drawn with replacement.[#] "Self-rehearsing" means that, in the process of responding to occasional random challenges, the user rehearses every aspect of the schema and key. In a recent paper (7), we established the existence of such schemas. Here, we improve on them in terms of both theoretical analysis and practical usability.[‖]

One-Way Functions. A one-way function (OWF) is a function that is efficiently computable yet not invertible by a poly-time Turing machine (8). By comparison, an HU-"OWF" is a humanly computable function that cannot be inverted in less than 10^{24} steps. We present a candidate HU-"OWF."

Pseudorandom vs. (Note the Quotes) "Pseudorandom" Generators. A standard (cryptographically secure) pseudorandom generator (PRG) is an algorithm that takes as input a random string (of digits or characters) of length n and outputs a string of length 2n** that are "virtually"[††] indistinguishable from a (truly) random string of length 2n by a poly-time Turing machine (8–10). By comparison, a "pseudorandom generator" ("PRG") is a schema that transforms n-digit input strings into 2n-digit[‡‡] output strings that, for $n \geq 20$, are "virtually" indistinguishable from random 2n-digit strings by a computer that takes no more than 10^{12} steps.[§§] An HU-"PRG," while reminiscent of a PRG, need not be a PRG. We present a candidate for an HU-"PRG."

Human Computation

For many cryptographic problems and games[¶¶] (e.g., speed chess, sudoku), human computation consists of a preprocessing

[§]Even to expect that the composition f(g(□)) of two humanly computable functions f(□) and g(□) will be humanly computable is wrong. It may be that f and g are humanly computable but that their composition is not as occurs, for example, when the human is not able to store the intermediate output of g in its limited short-term memory.

[¶]It would suffice to consider a computer that may run for no more than 10^{24} steps.

[#]Challenges randomly drawn with replacement from a dictionary with just one word (probability 1) have Q = 1 (i.e., the adversary needs to see only one challenge response pair to determine the correct response to the next challenge with probability >1/10). Note that, if a challenge repeats, we say the adversary wins, making the requirements more stringent.

[‖]An example of Q is as follows. Many people have a smallish number of passwords for all logins, typically k < 10 passwords. For the user who has a method—any method—to assign k passwords to all challenges, the adversary can try these randomly and succeed in about (asymptotically, up to a universal constant) $\sqrt{k}$ attempts; for k = 10, the expected number of attempts is $1 + 0.9 + 0.9 \times 0.8 + 0.9 \times 0.8 \times 0.7 + \ldots < 4$.

**This 2n can be replaced by n + 1, which can be extended to 2n.

[††]"Virtually" needs to be made precise.

[‡‡]The 2n cannot be replaced by n + 1 since humanly oriented algorithms are not in general composable.

[§§]Why this particular number? There are 10^{2n} random strings of length 2n but only 10^n challenges of length n for generating 10^n pseudorandom strings of this length 2n. Since the adversary knows the schema but not the challenge, she needs to try no more than 10^n challenges to prove that a given random string is not pseudorandom. To make this hard for her to do, we require that $10^n > 10^{12}$; 10^{15} would be better than 10^{12}, but the proof in this footnote does not allow us to claim 10^{15}.

[¶¶]In this paper, context determines whether "game" denotes an entire game in the usual sense or a single move in such a game. Examples of such single-move games include 1) speed chess in which the challenge or input is a chessboard position together with a color, black or white, and the response or output is that color's move; 2) sudoku in which the challenge is a typical sudoku board with digits in certain locations and the response is placement of one more digit in some location; and 3) crossword puzzles in which the input is a crossword puzzle, including its clues, partially completed and the output is a word inserted to the puzzle.

phase PREP (memorization of a public [i.e., published or at least publishable] schema plus generation and memorization of a private information or key) and a processing phase PROC (a run of the schema with its associated key on some input). We view PROC as the speedy computation (and output) of a function on a given input. This captures our motivating scenario of responding to each challenge with its associated password.

We wish to model humans for games and cryptographic problems but begin here with the more specialized model of humans for executing password schemas. For this purpose (of executing password schemas), we model a human as follows.

1) The human is a kind of PC, one in which the usual memory is replaced by two random access memories, one long term and the other short term.
2) Long-term memory is potentially infinite—upper bounded only by the usable lifetime of the human and the time that it takes to store information in that memory. Storing information (such as schema and key) in long-term memory is slow; reading information from long-term memory given a pointer to its location in memory is relatively fast. Storage in long-term memory is permanent provided that it is rehearsed on the doubling schedule described by Woźniak and Gorzelańczyk (11).[##] Spaced repetition has a long history in behavioral and experimental psychology (the comprehensive survey in ref. 12) and more recently, in neuroscience (13).
   Long-term memory is both written to and read from in the preprocessing phase. In the processing phase, long-term memory is used only for reading.
4) Short-term read–write memory is fast but tiny, typically storing two or three chunks (14), each chunk being a pointer to some item such as a digit or number, a character or word, or an image or music clip. In our model, unlike anything that we find in the psychological literature, a chunk is a well-defined object, a pointer into long-term memory.
5) In the context of passwords, the input (called the challenge) is presented as a singly linked list with a pointer in short-term memory to its leftmost start location. Whatever its location in the challenge, the pointer can be shifted one link right (but not left), read (past tense), or reset to the start location in one step. As an example, the challenge might be a word or phrase in long-term memory that the user has easy access to from left to right but not backward.

Schemas in general, not just password schemas, are algorithms intended for humans. They are used in association with information stored in permanent memory, which for passwords, includes a parameter called the key. Humans may use dice, paper, pencil, and other such tools to generate and memorize private keys and to memorize a public schema (PREP). They must thereafter execute the schema (PROC) in their heads (i.e., without using any tools outside of their head). This model will be sufficiently powerful for our password schemas. Toward the end of the paper, we briefly discuss extensions of the model.

A schema–key combination is considered to be (COMM TIME, MEM TIME, PROC TIME)-humanly usable if and only if it satisfies the following requirements.

1) COMM TIME is an upper bound on the time to learn the schema. This time includes the time to transform a few sample challenges into passwords (from a description of the humanly usable instructions) and all rehearsal time (for the life of the human). COMM TIME (including rehearsals) must be at most 10 min.
2) MEM TIME is an upper bound on the time to generate and memorize the schema's associated private key and all rehearsal time needed to maintain that memory. For passwords, this MEM TIME is required to be at most 2 h.
3) PROC TIME is an upper bound on the time to run the schema on a single input. For passwords, the PROC TIME is required to be at most 1 min.
4) A schema uses at most three (preferably at most two) pointers (chunks) into long- and short-term memory (model of a human part 3 above).[***]
5) For passwords, the schema and its associated key (both of which are stored in long-term memory) are completely self-rehearsing in that each and every instruction is run (this includes following each and every flowchart arrow) and that all elements of the key are rehearsed in a "significant fraction" of challenge–response computations.[†††]

From now on in this paper, COMM and MEM are considered to be a part of PREP. In general, not just in this paper, PREP TIME and PROC TIME must be specified to nail down human usability. In this paper, unless we say otherwise, we require that, for passwords, PREP TIME $\leq$ 2 h and PROC TIME $\leq$ 1 min. From here on, password schemas will be shortened to schemas.

Intermission. We now define the two complexity measures: 1) PREP = {(Human Complexity of creating and memorizing a key (MEM)) plus (cost of communicating and memorizing a Schema (COMM))} and 2) PROC = Human Complexity of Processing. We include COMM within PREP as it is clearly part of the preparation necessary to use a schema; we separate the two aspects of PREP, namely 1) memorizing and understanding the schema and 2) memorizing the key, as the two seem to be incomparable. Taking PREP and PROC in reverse order,

PROC(Processing Complexity) = total number of reads from long-term (permanent) memory plus total number of reads and writes to short-term memory while processing a single input.

To illustrate the measure, we examine the human complexity measure PROC of some natural operations.

1) Set a pointer to an item that is already in long-term memory (e.g., a sentence or telephone number) referred to in the schema or key: Cost = 1.
2) Move a pointer into the singly linked list for a challenge, sentence, telephone number, etc. to the right by 1 or set a pointer to the start or start + 1 or to the end or end − 1: Cost = 1.
3) Operations of + and × (mod 2, 3, 4, 5, 9, 10, 11) or $=^?$ on two single-digit operands: Cost = number of digits including the logical 0, 1 symbols created during the operation. Examples: $4 =^? 3$ and 4 + 3 (mod 10) have cost 1. 4 + 9 (mod 10) assuming that the addition consists of doing first 4 + 9 = 13 and then, 13 mod 10 = 3 has cost 2.
4) Apply a map, typically a map from letters to digits, that has been memorized as a hash function: Cost = 1.

[##]Woźniak and Gorzelańczyk (11) have shown empirically that, for an item to remain in long-term memory, the item must be rehearsed on a certain doubling schedule. Let t denote the time (after the initial memorization) between two successive successful rehearsals. Then, the item will remain in long-term memory provided that it is rehearsed at times 2t, 4t, 8t ... as measured from the time of completion of the second rehearsal.

[***]While human short-term memory is said to be able to store 7 ± 2 chunks, 2 or 3 is all that we have found to be humanly usable for PROC. Perhaps some of the other chunks are needed for other functions? We in any case assume that at most two or three chunks will be available for PROC.

[†††]"Significant fraction" is defined so that, after the total permissible PREP time has been exhausted, no additional purposeful rehearsal is needed as the natural computation of responses to random challenges will suffice to ensure that schema and key remain in permanent memory. Its definition assumes that the user will respond to random challenges at a well-defined rate.

We note that PROC is a measure of effort similar in certain ways to asymptotic complexity. Just as the actual running time of an algorithm can differ from computer to computer and even on the same computer depending on its load, the time taken for human computation can differ from one human to another and even for the same human. Unlike asymptotic complexity of algorithms running on computers, the PROC must not be estimated only for large enough inputs ("big-O"): an effective analysis must compute PROC to within a small additive constant. In later sections, we will perform such analyses for select schemas.

Next, we turn to the human complexity measure, MEM, of generating and memorizing a key, the first component of the preprocessing complexity PREP. For MEM, unlike PROC, the human may use a random number generator, paper, and pencil to create a random key and then, store it in permanent (human) long-term memory. After keys are generated and stored in permanent memory—and provided that the Woźniak and Gorzelańczyk (11) doubling rehearsal schedule (15) is followed to keep the key in permanent memory—the random number generator, paper, and pencil should no longer be needed even for rehearsal; in the case of passwords, just the normal typing of passwords would serve as sufficient rehearsal:

$$\begin{aligned} &MEM(\textit{Memorization Complexity}) = \\ &(\textit{number of tosses of a k-sided die}) \times (\log_2 k) + \\ &(\textit{number of chunks written to permanent memory} \\ &(\textit{presumably the key and the schema})). \end{aligned}$$

One measure of MEM complexity is by comparison with commonly memorized quantities. Some examples with rough estimates of their costs are given below. These costs are for long-term permanent memorizations as opposed to the short-term temporary memorizations counted in PROC.

1) Linked list memorization of a 10-digit string of chunks is equivalent to memorizing a random 10-digit phone number. Cost = number of chunks in the string (=10 for the 10-digit phone number). As an aside, after a string is memorized, it itself becomes a chunk. As a related example, consider linked list memorization of a random-looking list of chunks, such as the digits of π. Most school children learn a few digits of π, and a very few learn maybe 20 digits.[‡‡‡] Cost = number of chunks (which except in rare cases, are digits).
2) Linked list memorization of a long string of chunks (letters), roughly equivalent to memorizing the alphabet, is something that we can do by age 4 or 5 and until late in life. Cost = number of chunks in the string (=26 for the alphabet).

In both of the above cases, people memorize a linked list with a pointer to the start of the list and in the case of a longer list like the alphabet, several additional pointers into the list. Access to this linked list does not enable us to recite the list quickly from memory backward (Z Y X ... A)—although of course, we can learn to do that if we want. The cost of a linked list memorization of a string of chunks, as in 1 and 2 above, is a "fraction" of the cost of a random access memorization of a map from chunks to chunks as in 3 below. Because these two kinds of memorization are (so) different, we specify for each memorization whether it is a linked list or a random access hash. As an aside, to memorize a small amount of data such as a telephone number as a hash, we suggest memorizing that data initially as a singly linked list (e.g., as a telephone number t_1t_2 ...) and then, using that memory to enable memorizing the same data as a hash function (e.g., 1 ➔ t_1, 2 ➔ t_2 ...).

When it comes to preprocessing, humans more easily store information in a linked list than in a (random access) map. When it comes to processing, however, maps are typically more useful than linked lists.[§§§]

1) Assume that Alfa, Bravo, Charlie ... are chunks. Linked list memorization of the list Alfa ➔ Bravo ➔ Charlie ... ➔ Zulu: Cost = 26.
   Random access memorization of the map A ➔ Alfa, B ➔ Bravo, C ➔ Charlie ... Z ➔ Zulu, which is something that ham radio operators, Boy Scouts, and the military do. Cost = number of chunks in the map (2 × 26 = 52 for the ham radio map). This is twice the cost of the linked list memorization.

   Random access memorization of the Morse code map: A ➔ •–, B ➔ –•••, C ➔ –•–• ..., which typically takes days if not weeks. Cost = number of chunks in the map. In Morse code, assuming that the chunks are •, –, A, B ... Z, map C ➔ –•–• costs 5, while E ➔ • costs 2. Cost of random access memorization of the 26-letter Morse code is, therefore, =108.
4) Linked list memorization of a poem, the lyrics to a song, or the (meaningful) 272-word Gettysburg Address is the kind of thing that middle school students regularly do. Note that the rhyme in poetry, the melody in song, and the meaningfulness of prose help greatly with memorization (none of which is accounted for in our cost measure). Cost = number of chunks, which in the case of the Gettysburg Address, is the number of words =272.

The second component of PREP is COMM, the cost of communicating an algorithm to a human. In standard algorithm theory, COMM might simply be the length of the program being communicated. In the study of humanly usable algorithms, we propose

> *COMM (Communication Complexity) = length of (a precise description of) the humanly readable/humanly understandable algorithm + length of the traces of execution on enough example(s) to cover all cases (execute every instruction and take every control flow arrow) in the algorithm.*

For example, suppose that the preprocessing algorithm requires the user to memorize two randomly generated functions f_1, f_2 from the 26 letters to the 10 digits. To describe this operation, the algorithm must tell the human user exactly how to generate, memorize, and compute these functions. Since memorizing two such functions can take twice as long as memorizing one and since the two functions can be confused, a better algorithm might suggest to memorize a single function f from the 26 letters to the two-digit numbers, 00 to 99, and then, to set $f_1(x)$, $f_2(x)$ = most significant, least significant digits of f(x). These slightly longer, more detailed instructions are crucial for keeping the preprocessing time under the stipulated 2 h.

[‡‡‡]One champion memorizer, Akira Haraguchi, learned 100,000 digits of π in 10 to 15 y; 15 y will suffice if the memorizer learns roughly 25 new digits per day 5 d/wk. Assuming that 25 new digits can be memorized in 15 (concentrated) min, meaning 15 min from start to the first complete recital, and that each rehearsal of the 25 digits can be done in 1 min, the memorization time for the first day in which 25 new digits are memorized would be 15 min for the new digits + 1 min per rehearsal of these 25 digits at 15, 30, 60 min = 1, 2, 4, 8, 16 h = 15 + 7 = 22 min. To rehearse previously memorized digits on that same day would take 1 min each for the digits memorized 1 d, 2 d, 4 d, 8 d, 16 d, 1 mo, 2 mo, 4 mo, 8 mo, 1 y, 2 y, 4 y, 8 y, 16 y, 32 y, and 64 y ago, which comes to 16 min. The total is 22 + 16 = 38 min/d to achieve this awesome feat. Essentially, anyone who cares enough for π to spend 15 y at it can do this. Haraguchi cares: he views π as "the religion of his universe."

[§§§]We humans rarely notice how our brains store information. For example, most of us store our ABCs automatically in a singly linked list, not in a doubly linked list and not in a random access data structure.

Because humans have different computation rates, we count steps, which are independent of the human, rather than running time. To relate step counts to running time, we give bounds on the time that it typically takes a human to perform each step.

More precisely, we propose to use the pair <PREP, PROC> as the right measure of human effort to use a schema. This pair is typically a mix of numbers (for PREP) and formulas (a function of the challenge length, n, for PROC). We are not against using formulas for these counts; we are just against the typical emphasis on polynomial vs. exponential step counts as these distinctions are not useful for human computation.

Examples of Password Creation (and Re-Creation) Schemas

We present three schemas that illustrate the human computation model and associated complexity measures, two here and one in *SI Appendix*. For all three schemas, preprocessing requires generating and memorizing a single letter to digit map.

Letter Substitution Schema.

Preprocessing. Memorize a single random (i.e., uniformly random) letter to digit map f. The memorization of a map from letters to digits can be done with 30 min of concentrated effort (1, 16) up front and about the same amount of total additional time spent on spaced rehearsals (1-min rehearsals at the 21 successive later times: 1 h, 2 h, 4 h, 8 h, 16 h, 1 d, 2 d, 4 d, 1 wk, 2 wk, 1 mo, 2 mo, 4 mo, 8 mo, 1 y, 2 y, 4 y, 8 y, 16 y, 32 y, and 64 y).

Processing. Given a challenge (string of letters) C, run the following algorithm.

1) Set a pointer to the first letter of C
2) Repeat until the entire challenge has been processed
   - 2.1) Apply map f to current letter
   - 2.2) Output mapped value
   - 2.3) Shift pointer to next letter of challenge

Processing uses only two chunks of short-term memory: one pointer into the challenge and one for the mapped value of the current letter.

***Example*:** Suppose that the map from letters to digits is given by the position of the letter in the standard alphabetic ordering mod 10 (this is only for illustration; the reader is urged not to use this map). Specifically, A ➔ 1, B ➔ 2 . . . I ➔ 9, J ➔ 0, K ➔ 1 . . . Z ➔ 6. Then, this letter substitution schema maps GMAIL ➔ 73192 and APPLE ➔ 16625.

COMM = (description of preprocessing is less than 10 words; of processing is less than 40 words, and the description of the example is less than 60 words. Length of actual trace is 2 (initialization) + 3n steps on a challenge of length n) = 2 + 3n. Since n = 5 for the two traces, total cost is less than 10 + 40 + 60 + 2(2 + 3 × 5) < 150.

MEM = 26 tosses of a 10-sided die to generate the random key plus 26 pairs to store it has a cost of 78. The lion's share of the cost is the hour that it takes to memorize the key.

PROC = on a challenge of length n, this comes to (n reads from long-term memory to compute the f function n times) + (2n reads and writes to short-term memory) = 3n.

Modifications of this letter substitution schema might use special rules to avoid consecutive repeated letters. These include skipping a consecutive repetition or shifting up in the alphabet by one or two to avoid consecutive repetitions. An example is AAA ➔ 123.

Skip to My Lou Schema. The skip to my Lou (STML) schema (named after a popular children's song in the United States) computes $F_x(C)$ as follows.

Preprocessing. Memorize a random map x as a hash function from letters to digits.

Processing. Given a challenge C consisting of a string of letters and the map x, do the following.

1) Set SUM_V = mapped value of last letter of challenge
2) Set POINTER (current letter) to first letter of challenge
3) Repeat until POINTER shifts past last letter of challenge
   - 3.1) POINTER_V = mapped value of pointer
   - 3.2) Set SUM_V = (SUM_V + POINTER_V) mod 10
   - 3.3) If SUM_V is less than 5, output SUM_V (in any case, do not modify SUM_V)
   - 3.4) Shift POINTER from current letter in challenge by one letter to the right

Output: the string of SUM_V outputs produced by the above algorithm is $F_x(C)$.

***Example*:** Let the memorized map x be (A ➔ 1, B ➔ 2 . . . I ➔ 9, J ➔ 0 . . . Y ➔ 5, Z ➔ 6). This maps the individual letters of GMAIL to 7, 3, 1, 9, 2. The sequence of SUM_V values is 9, 2, 3, 2, 4, resulting in $F_x(\text{GMAIL}) = 2{,}324$:

$$\begin{aligned}\text{PROC} &= [\text{apply map} + \text{set SUM} + \text{shift pointer}\\ &+ \text{n}(\text{apply map} + \text{add mod 10} + \text{compare with 5}\\ &+ \text{output(maybe)} + \text{shift pointer})] = 3\\ &+ \text{n}(1 + 1.5 + 1 + 0.5 + 1) = 5\text{n} + 3.\end{aligned}$$

The output has expected length that is half the length of the challenge. To get a longer output, the user can append a fixed string to the challenge and run STML on the extended challenge. (Note that appending a fixed string to a challenge yields far better security than appending a fixed string to a password: if appended to the password, an adversary can determine the string from observation of two passwords, something she cannot do in any obvious way if the string is appended to the challenge.)

Information-Theoretic Security. We remark here that these schemas have a simple lower bound on the security parameter Q. Namely, in order to guess the response to a challenge with probability greater than 1/10, the adversary has to have seen all of the characters in the challenge in previous challenges. The expected number of challenges at which this happens is a function of the distribution of challenges. For illustration, the table below shows the Q value, its SD, and the value needed for 90% of the challenges for three different distributions: for words chosen uniformly at random from the English dictionary, the top 500 most commonly visited internet domains and random seven-letter strings (Table 1).

HU-OWFs

In complexity and cryptography theory, an OWF function F[¶¶¶] is defined to be any function (from strings to strings over some finite alphabet) such that 1) F(x) can be computed by a Turing machine on any input string x in time poly(|x|)[###] and that 2) any Turing machine that runs on input y in poly(|y|) steps (for some fixed poly) has a negligibly small probability to invert F [i.e., to find a preimage x′ such that F(x′) = y = F(x)]. OWFs are conjectured to exist and play an important role in complexity theory and cryptography.

An HU-"OWF"[****] F is a function that 1) can be computed by a humanly usable schema[††††] and 2) cannot be inverted on any but a "small" fraction of outputs by a computer that has

[¶¶¶] Note that a humanly computable password generator is not in general either a "OWF" or a "PRG."

[###] |x| = length(x).

[****] The HU-"OWF" is defined to be humanly usable by the fact that a humanly usable schema can compute its output. It is actually humanly usable in the sense that humans can compute it if the input is short enough (typically of length at most 25 but possibly more) and humans can perform the mandated operations in the mandated times (typically 1/3 to 1 s).

[††††] This puts a strict upper bound on the length of input and/or time to compute the HU-"OWF." Furthermore, to be an interesting concept, the input length, n, must have a minimum size, say n > 24, which would ensure that the adversary cannot simply try all 10^{25} possible x values.

knowledge of the HU-"OWF" algorithm but not the key and that executes a total of at most 10^{24} instructions.

Do HU-OWFs exist? We propose a candidate here based on the STML schema. Let N be the size of a sufficiently large alphabet over which the map is defined (e.g., n = 26 for the standard English alphabet and n = 100 for two-digit numbers; n = 10 will not do as it would permit an adversary to invert F by running through all possible x values). A random map x from N characters (letters and/or digits) to the 10 digits can be written as a string of N digits—the map values in a fixed ordering of the alphabet. Now generate and fix a random challenge C of sufficient length (roughly N ln N, the precise length to be determined) so that every character (typically letter or digit) is likely to be in the range. For a fixed such C, define the function F_C by $F_C(x) = F_x(C)$, the function that takes as input the N-digit string x and outputs a string y by running STML with the map defined by x on a fixed challenge string C. Note that STML is being used here to compute $F_x(C) = y$ with x variable and C held constant, not x held constant and C variable.

The computational problem for the adversary—who knows C and y (and therefore, also F_C)—is to find a string (map) x with image under F_C that is the observed y. We are not aware of any algorithm to solve this problem efficiently (i.e., to solve this inverse problem in less than 10^{24} steps).

Since the input x is now the key, not the challenge, a password schema that computes an HU-"OWF" has the important property that, from knowledge of y, an adversary cannot determine the key, x. If both x and C are unknown to the adversary, then it is hard for the adversary not only to determine x but also, to determine C. This is no big deal if the adversary knows the challenge, which is what this theory assumes. However, in practice, the user of a schema could modify challenges using some personal rule, such as the starting location (*SI Appendix*). In that case, the adversary will not know the actual modified challenge either.

HU-"PRGs"

A standard (cryptographically secure) PRG is an algorithm based on a parameter/key k and a polynomial poly, poly(n) > n, that maps a (random) input string, the challenge/seed, of length n to a random-looking output string of length poly(n), the pseudorandom string. That output "looks" random in the sense that any poly-time computer with access to random bits that has knowledge of the general algorithm for computing the PRG but not the key, k, cannot distinguish a string output by the PRG from a truly random string R of the same length with negligible probability of error.

In this paper, a "PRG" is an algorithm having a parameter/key k that maps a (random) input string, the challenge/seed, of length n to a random-looking output string of length at least 2n, the "pseudorandom" sequence.[‡‡‡‡] That output "looks" random in the sense that a computer that has knowledge of the PRG algorithm (but not the key, k) and that executes a total of at most 10^e instructions for some specified e (which may be as large as e = 24 or as small as e = 12) cannot distinguish[§§§§] a string output by the "PRG" from a truly random string R of the same length with probability of error < 1/4. While 10^{12} does not seem to be a high bar for modern cryptography, we think that it is already an interesting threshold for the security of humanly usable protocols.

[‡‡‡‡]The output of a PRG on a seed of length k is typically only required to be of length k + 1, which can be extended to longer lengths by reapplication of the PRG. Such reapplications make the PRG humanly unusable for many reasons, including that the human has no place to store the intermediate k + 1 digits, which are needed as seed to generate the next k + 2 digits.

[§§§§]"Indistinguishable" is usually up to a negligibly small 1/poly(n) factor. In human usability, this must be replaced by a concrete epsilon.

Table 1. Information-theoretic security bounds

Dictionary	Security	SD	90% value
English 50,000	6.23	1.80	4
Top 500 domains	6.60	2.03	4
Random 7 letter	7.54	1.58	6

The "PRG" is an HU-"PRG" if its output can be computed by a humanly usable algorithm and therefore, assuming that the seed is short enough, by humans that can perform the mandated operations in the mandated times (typically 1 min).

We propose a humanly computable quasi–HU-"PRG" based on STML. It is not a true HU-"PRG" for several reasons, the most important of which is that we have no proof (not even one based on reasonable assumptions) that the output is "pseudorandom." We have only a hope, which we have not been able to translate into a proof.

Our HU-"PRG" takes as input an n-long string of digits in the set {0,1,2,3,4} and as outputs a random-looking string of ~2n digits in the same set, {0,1,2,3,4}.

The intermediate work of the algorithm involves all 10 digits {0,1 . . . 9}. That work is private (except for whatever it outputs whenever it outputs it) and can be performed in the user's tiny short-term memory with the few allowable pointers plus digits from the set {0,1,2 . . . 9}.

The HU-"PRG" uses STML. When used as a password schema, STML uses a letter to digit map; for a "PRG," it uses a digit to digit map.

The basic idea is as follows.

Initialize.

1) Set SUM = last digit of the input
2) Set pointer to first digit of input
3) Until pointer drops off the challenge
   - 3.1) Set SUM = mapped SUM (comment: the map is from digits to digits(
   - 3.2) Set SUM = SUM + current digit (mod 10)
   - 3.3) If SUM is less than five, output SUM
   - 3.4) Shift pointer to next digit of input

A detailed example is given in *SI Appendix*.

A password schema that is also a PRG has some useful properties compared, say, with the schema that replaces each letter of the challenge by its hash value under the key. For one, it can make it more difficult for the adversary to get a handle on the user's key.

The HU-"PRG" that we propose is based on a key, k, which is a string randomly chosen from the 10^{10} strings of digits of length 10. The string $k = k_1\ k_2 \ldots k_{10}$ represents the hash function 1 -> k_1, 2 -> k_2 . . . 0 -> k_{10}. The schema takes as input a random seed/string of digits C of a length n (to be determined) with digits that have been randomly chosen from the set {0,1,2,3,4}. The initial output $F_k(C)$ of the PRG is the output of STML applied with key k to challenge C. This string $F_k(C)$ is rarely long enough to be the complete output of the HU-PRG as its expected length is just half the length of C. To produce a longer string, at the cost of one more (expensive) pointer, STML is run on many substrings of C, and the results are concatenated. The substrings of C have to be chosen in a humanly usable way. For a seed C of length n, let C_i be the substring of characters obtained by skipping the first i characters of C, then including the next i characters, skipping the next i, including the next i, and so on (e.g., C_1 is the substring of characters in even positions, and C_2 is the substring of characters obtained by alternately skipping two and including two). To be humanly usable (if only barely), keep track of the skip length on one's fingers.

STML HU-"PRG" 1. For a seed C of length n, the output of the HU-"PRG" 1 is the concatenation $F_k(C) \cdot F_k(C_1) \cdot F_k(C_2) \cdot \ldots \cdot F_k(C_{\lceil (n-1)/2 \rceil})$. In the first application of STML, the carry digit is the default last digit of the challenge. From the second application onward, the carry digit is the running sum mod 10 from the previous iteration (regardless of whether the digit was output).

The expected length of the output is approximately $\frac{n+\frac{n}{2}+\ldots+\frac{n}{2}}{2} = \frac{n\left(1+\frac{1}{2}\cdot\frac{n-1}{2}\right)}{2} = \frac{n(n+3)}{8}$.

***Example*:** Suppose that the seed is 31415926[¶¶¶¶] and that the key is k(i) = 3i mod 10. Then, starting with the last digit, 6, as carry and using the entire challenge as the seed, this maps in the first round to 14692577, which maps to 142. Then, for the substring of the challenge consisting of digits in even positions (1196), with carry 7 from the previous round, we get 2706, and the output is 20. Next, by skipping two and using two, we have the substring 4126. With carry 6, this maps to 2735, and the output is 23. Then, skipping 3 digits and using 3 digits, we have the substring 159. With carry 5, this maps to 638, and therefore, the output is 3. Next is the substring 5926, which with carry 8, maps to 9606, and the output is 0. Thus, the output is 142202330.

The following theorem proves that the STML HU-"PRG" 1 can be broken for challenges of length n = 10. The argument fails for n = 20.

Theorem. *STML HU-"PRG" 1 can, with high probability, be broken for challenges of length n = 10 and output 2n = 20 using at most 10^{16} operations.*[####] *(At 10^{11} operations per second, this takes at most 28 h on a laptop.)*

***Proof*:** The 2n digits of output must come from half of ∼4n locations. Guess the 2n of 4n locations from which the output comes. There are $\binom{4n}{2n} \le 2^{4n}$ possible choices ($2^{40} \sim 10^{12}$). For each choice, we get a set of 2n linear equations mod 10 in 10 variables. Of these, there will likely be n = 10 linearly independent ones. They can be solved by Gaussian elimination in $10 \cdot 10^3 = 10^4$ operations, where the factor of 10 is an upper bound on the number of steps for a single pivot. This gives a total of 10^{16} operations to find the key for all digits of the given challenge.

We now define a second, simpler candidate. It uses a single digit to digit map as before and produces an output of length twice the length of the challenge in expectation.[*****] The function, $F_x(\cdot)$, and substrings C_1, C_2 of the challenge are defined as above in STML HU-"PRG" 1.

STML HU-PRG2. On a challenge C of length n,

$$\text{Output} = C \cdot F_x(C) \cdot F_x(C_1) \cdot F_x(C_2).$$

This HU-PRG2 proceeds exactly like STML HU-"PRG" 1 but only for four passes. In four passes, a string of length n is expected to result in an output of length n + (n/2 + n/4 + n/4) = 2n.

As an example, suppose that the challenge (seed) is 31410421 (which is the first eight digits of π mod 5) and that the key is k(i) = 3i mod 10. Then, in the first pass, with the last digit 1 as carry, the challenge maps to 691420443, and the output is 1420443. The second pass maps the subchallenge 1141 with carry 3 to 0172, and the output is 012. The third pass, with carry 2, maps the subchallenge 4121 to 0156 with output 01. Thus, the entire output is 314104211 1220443 012 01.

Open Question. Can STML HU-"PRG" 2 on challenges of length n ≥ 20 be broken using less than 2^{20} operations? If not, show that any humanly computable algorithm that breaks it must take at least 2^{20} steps.

It is a tantalizing possibility to generate a longer string by allowing the human to use previously generated output in order to generate more output. We note that the seed could be a letter string and that the key could be a letter to digit map in place of a digit string and a digit to digit map as above. When the letter string has additional meaning (e.g., it is a word or phrase), then the human might not even need to have the seed in writing while running HU-"PRG" 1 and will only need to hear or see it once.

Discussion

The model and findings of this paper raise several interesting questions. We mention a few.

We have considered the complexity of human computation in the most restricted setting where all computation is in the head and the human has no access to paper, pencil, or other aids. A next step could be to consider the complexity of playing games, such as speed chess or sudoku, where one also considers the current board position but does not necessarily make or refer to notes. How do we model such computation?

More generally, how should visual input be modeled? We have so far considered inputs as challenges that are read left to right as strings. However, this does not take into account our facility for visual parsing. A good model could be useful in rigorously evaluating human–computer interfaces.

Pseudorandomness is a fascinating concept both philosophically and for practical reasons. As computation becomes more distributed, many important protocols are randomized, and it is important for interacting agents to be able to generate provably effective random bits/digits. More abstractly, can a limited computational model (such as our model for human computation) generate digits that seem random to a computer? It would be most interesting to show that our STML-based HU-"PRG" is secure against limited adversaries (e.g., finite-state machines), analogous to Nisan's celebrated PRG that fools finite-state machines.

Finding humanly usable password schemas is, in our opinion, a rich and rewarding research direction. While we have proposed (and analyzed) several schemas, we expect that there are many more to be discovered. In *SI Appendix*, we discuss some simple methods, including ways to change passwords.

Data. All data used to support the paper are included here.

ACKNOWLEDGMENTS. We thank Mr. Vempala Venkata Rao (1938 to 2020), who adopted a password schema at age 76 and provided valuable feedback.

[¶¶¶¶] For symmetry, we might (but do not) require that the seed digits be randomly chosen from {0, 1 . . . 4}.

[####] This is an upper bound. Is there a more efficient way? One might conjecture that no method can break the "PRG" in less than $10^n = 10^{10}$ steps.

[*****] Because the output has expected as opposed to guaranteed length twice that of the challenge, STML HU-"PRG" 2 is not a true "PRG."

1. J. Blocki, "Usable human authentication: A quantitative treatment," PhD thesis, Carnegie Mellon University, Pittsburgh, PA (2014).
2. B. Ives, K. R. Walsh, H. Schneider, The domino effect of password reuse. *Commun. ACM* **47**, 75–78 (2004).
3. J. Bonneau, "The science of guessing: Analyzing an anonymized corpus of 70 million passwords" in *IEEE Symposium on Security and Privacy* (IEEE Computer Society, 2012), pp. 538–552.
4. Z. Li, W. He, D. Akhawe, D. Song, "The emperor's new password manager: Security analysis of web-based password managers" in *23rd USENIX Security Symposium* (USENIX Association, 2014), pp. 465–479.
5. H. Kruger, T. Steyn, B. Medlin, L. Drevin, An empirical assessment of factors impeding effective password management. *J. Inf. Priv. Secur.* **4**, 45–59 (2008).
6. R. Shay *et al*, "Can long passwords be secure and usable?" in *CHI Conference on Human Factors in Computing Systems* (ACM, 2014), pp. 2927–2936.

7. M. Blum, S. Vempala, "Publishable humanly-usable secure password creation schemas" in *Proceedings of the Third AAAI Conference on Human Computation and Crowdsourcing*, E. Gerber, P. Ipeirotis, Eds. (AAAI Press, 2015), pp. 32–41.
8. O. Goldreich, Foundations of cryptography—a primer. *Found. Trends Theor. Comput. Sci.* **1**, 1–116 (2005).
9. M. Blum, S. Micali, How to generate cryptographically strong sequences of pseudo-random bits. *SIAM J. Comput.* **13**, 850–864 (1984).
10. A. Yao, "Theory and application of trapdoor functions" in *23rd Annual IEEE Symposium on Foundations of Computer Science* (IEEE Computer Society, 1982), pp. 80–91.
11. P. A. Woźniak, E. J. Gorzelańczyk, Optimization of repetition spacing in the practice of learning. *Acta Neurobiol. Exp. (Warsz.)* **54**, 59–62 (1994).
12. N. J. Cepeda, H. Pashler, E. Vul, J. T. Wixted, D. Rohrer, Distributed practice in verbal recall tasks: A review and quantitative synthesis. *Psychol. Bull.* **132**, 354–380 (2006).
13. E. A. Kramár *et al*., Synaptic evidence for the efficacy of spaced learning. *Proc. Natl. Acad. Sci. U.S.A.* **109**, 5121–5126 (2012).
14. G. A. Miller, The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychol. Rev.* **63**, 81–97 (1956).
15. P. Pimsleur, A memory schedule. *Mod. Lang. J.* **51**, 73–75 (1967).
16. S. Samadi, S. Vempala, A. Kalai, "Usability of humanly computable passwords" in *Proceedings of the Sixth AAAI Conference on Human Computation and Crowdsourcing*, Y. Chen, G. Kazai, Eds. (AAAI Press, 2018), pp. 174–183.