

Human Computation with an Application to Passwords

Manuel Blum

Santosh Vempala, Jeremiah Blocki

Elan Rosenfeld, Michael Wehar, Samira Samadi, Bhiksha Raj



Human Computation with an Application to Password Generation

- This talk is about **human computation**. My goal is to **apply CS ideas to what humans can compute in their heads**.
- (I hope to extend this eventually to allow writing. At first only required writing will be allowed: think Crossword Puzzles, Sudoku, and various Crypto Protocols. In that case, what's written is then available for further computation and output. There is no eraser and no other paper/pencil.)
- My running **Example** will be **Password Generation**. In this example, I would like all computation to be done entirely in the head. This means no paper or pencil. Even the generated password is to be invisible.

Password Generation

Major point: password generation is best viewed as a function from

challenge → **response**


website → **password**



I require functions to be humanly computed
(computed in one's head w/o paper pencil)



Password Generation

- I need to show you that **it is possible for a normal human being (me) to generate – and regenerate – random looking passwords quickly on the fly.**
- I do this by applying a (publishable) **humanly usable algorithm** that I call a **(public) schema**, which uses a randomly chosen **(private) key** as parameter, to the website name. 
- I want you to observe that I can use **schema + key** to generate a random-looking password response to any website **quickly** this way.

Password Generation

For this demonstration I need a **volunteer** – someone whom you trust is not in cahoots with me – to run my **Python Program**. This is to prove to you that I'm not cheating, and to check my work (because I make mistakes).

I also need:



1. Someone to **give me a 5 or 6 letter word** typical of a website name, like AMAZON or JASON, maybe your own name, to use as challenge;
2. Someone to write challenge and my response on board behind me. You will see that I can create **passwords w/o paper, pencil** or notes.

Do you see any pattern?

MAGIC1 → 436115 278376

MAGIC2 → 292849 405629

MAGIC3 → 984239 642019

MAGIC4 → 123404 881985

MAGIC5 → 311575 154499

MAGIC6 → 609092 393253

MAGIC7 → 040782 529504

MAGIC8 → 778658 766869


MAGIC9 → 565927 917135

An Easy-to-Use but Insecure Algorithm for Generating Passwords

If you know the alphabet **A B C D E F ... X Y Z**, then you can map each letter of the challenge (website) to the next letter in the alphabet:

Example: **EBAY** → **FCBZ**

Unfortunately, **AAAA** → **BBBBB** 

Also, **unfortunately**, if all of you did this, you would all have the same passwords. 

The weakness is that this public **schema** does not use a private **key**!

A Public Schema for Generating Passwords must use a secret random KEY

For example, memorize a random phone number as KEY.

Example: Key = (246) 357-0189.

Use Key as shift:

EBAY → E+**2** B+**4** A+**6** Y+**3** E+**5** B+**7** A+**0** ...

→ (GFG) BJI-AZMK (= GFGBJIAZMK)

AAAA → (CEG) DFH-AB I J (= CEGDFHAB I J)

Now AAAA is not a problem 😄 and

Short challenges → **10 letter passwords**



To do this, you must memorize a phone number.

Are you willing to memorize a phone # you use frequently?

Many different Algorithms for Generating Passwords are possible

For example, memorize a random permutation on the letters, like "THE QUICK BROWN FX JMPS V LAZY DG."

EBAY → QRZD 😄 but

AAAA → ZZZZ OUCH !! 😞

To fix the AAAA and password length problems, as already suggested, memorize a random phone number like (246) 357-0189, then use it as a shift.



How much harder is it to memorize and use "THE QUICK BROWN FX..." than to memorize the key to a password vault? **It's slightly harder.** A social security number? **Slightly harder.** How about 100 passwords? **A lot harder !!**

To satisfy password requirements

start every password with **aA1@bB2\$**

Attaching **aA1@bB2\$** to every password gets you nearly **100%** from every password checker.

To test a password schema,

use not one but **BOTH** of the following tests:

1. Check if **Google** recognizes (typical) passwords that the schema generates.
2. Use a **password checker** like passwordmeter.com

Caution: Run tests 1 and 2 on faux passwords!

This completes PART ONE of this talk

PART TWO: Human Usability

PART TWO: Human Usability

I will shortly present **A Model for Human Computation**.

En route, you may rightly ask: **How will this model help me?** Will it help me solve crossword puzzles or Sudoku better? **Will it help me play a better game of chess?**

NO. The model of human computation won't help you play chess better, just as the model of a Turing machine won't help you do matrix inversion faster.

In **LOGIC**, the importance of the **Turing machine** is in distinguishing what is computable from what is not.

In **THEORETICAL COMPUTER SCIENCE**, the importance of the **Turing machine** is in distinguishing what is efficiently solvable from what is not.

In this **COG SCI COMP SCI** area, the **model of human computation** is for distinguishing what is humanly computable from what is not, and for estimating the human effort required to do what is humanly computable.

What the Model of Human Conscious Computation Does

It provides a way to **estimate human usability** of a schema without having to test 100 people for each new schema or for each modification of a schema.*

Of course, we need to validate our model experimentally, but we can then **count steps and prove human usability mathematically**.

The model is needed to **prove the impossibility** of schemas (humanly usable algorithms) for certain problems. Example: are zero knowledge protocols achievable with humanly usable schemas?

How the Theory of Human Conscious Computation Differs from Computer Science Theory

1. A major **difference between computers and humans**: besides that **computers are much faster than humans** in doing arithmetic operations, **computers are constantly improving in speed and memory, whereas human replacements are the same old model**. On this account,
2. When counting steps to determine the complexity of schemas, **specific constants are crucial**. $O(n)$ is not good enough. Step counting must specify constants much more precisely, e.g. use $c \cdot n + O(1)$ for a specific c rather than $O(n)$.

Human Conscious Computation versus Computer Science Theory

3. Guiding us is the following **Major Open Problem**: can **humans compute 1-way functions (in their head) that a powerful supercomputer cannot invert?** Think about it. Do you think it possible? Is it even imaginable?
4. Similarly, can a human generate pseudo-random numbers that are indistinguishable from truly random numbers? More formally, can a human in a few minutes transform 20-digit random nos into 40-digit pseudo-random numbers that a supercomputer working for an hour cannot distinguish from truly random 40-digit nos?

A Model of Human Conscious Computation

For human usability, I need to present a **model of human conscious computation***. This model is a Turing machine **with two distinct memories** in place of tape.



-
- * **Unconscious computation**: ride a bike.
 - * **Conscious computation**: multiply a 2-digit number by a 1-digit number in your head. Recall image of the Mona Lisa.

The two memories of our Turing machine are:

1. **Long term or permanent memory (LTM)**, which is hard to write (can only be written to slowly) but easy to read (can be read quickly). **Retention** in **LTM** requires rehearsals on Wozniak's doubling schedule.
2. **Short term or working memory (STM)**, which is easy to read & write, but tiny: **7±2** chunks. Storage is fleeting: items in **STM** are pushed out by new deposits.

As computer scientists you recognize STM as cache (**but STM is much more than cache**).



Long term memory LTM

- **Long term permanent memory (LTM)** is **virtually infinite** but permanence requires rehearsals on Wozniak's doubling schedule. **LTM** is used to store both Schema and Data.
- **Virtually infinite** means that **LTM** can store info (like a secret key) for life - until the human can no longer use it (the password) anyway.
- **References:**
 - **Piotr Wozniak** (not Steve Wozniak) **Super Memo:**
 - Magician **Juan Tamariz** teaches **how to memorize** a random 1:1 map from the 52 cards to $\{1,2,3,\dots,52\}$ in 3 hours.

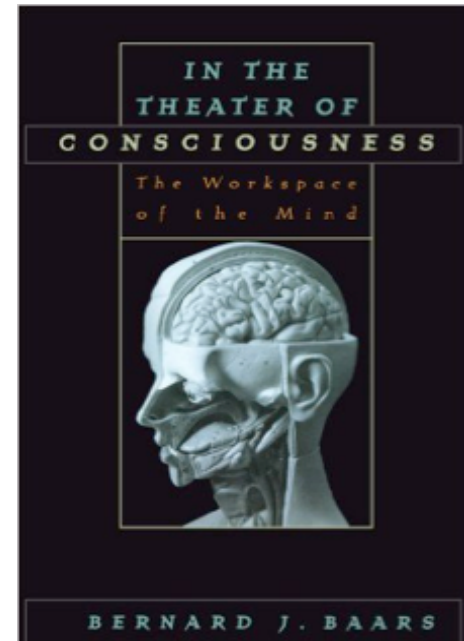


Short Term Memory STM

What I write in BLUE, fascinating as it is, is not strictly part of the model. Description of the model is in black.

From the literature, the contents of STM are what we are consciously aware of.

- **STM** is the stage in the **theater of consciousness (Bernard Baars)**.
- The contents of **STM** are actors performing on that stage.
- Audience members are the many **processes of the brain**. They pay attention to the stage.
- Any audience member with relevant info can pass it back to the stage*.



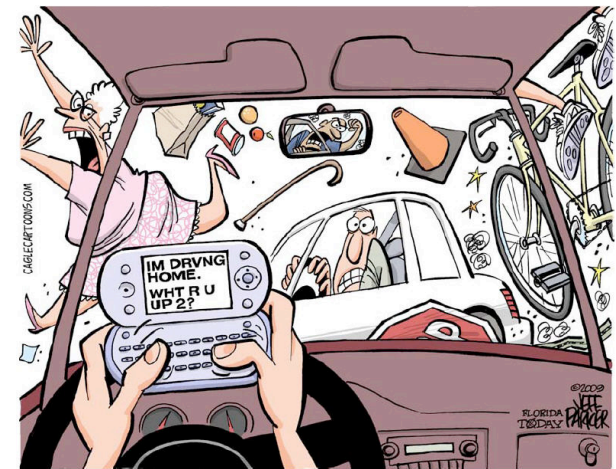
*In this sense, **STM** is much more than cache.

The items in **STM** are what psychologists call **chunks**.

- **Chunks** are **pointers into memory**. 💡
- **STM** can send info to and receive info from wherever that info is located in memory.

Question: why does short-term memory have room for only 7 ± 2 chunks, not more? Are there reasons why this number is so small... ?

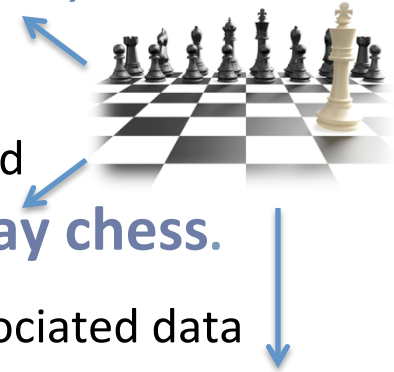
Is it really better to pay full attention to a few important things than divided attention to many more things?



The CS model: PREParation and PROCessing

Games and crypto-problems (Crossword Puzzles, **speed chess**, Password generation) have **2 parts: PREP and PROC.**

- 1. PREParation** has to do with
 1. obtaining/generating data, and
 2. writing schema and data into permanent **LTM**. E.g. **Learn/play chess.**
- 2. PROCessing** has to do with applying schema (in **LTM**) with associated data (in **LTM**) to the input and generate output. **Observe board; make move.**



Our measure of human effort is human compute time.

- 1. #PREP** is the preparation time – including rehearsals – to **write** schema and data into **LTM**. **For password generation, we place an upper bound of at most 3 hours, preferably 1 hour – including rehearsals – on what is permissible #PREP.**
- 2. #PROC** is the processing time it takes to apply the schema to the input and to generate the output. **For password generation, we place upper bound of at most 6 secs/letter, preferably 2 secs/letter, on permissible #PROC.**

PREP

PREparation stores the **SCHEMA** and associated **DATA** in **LTM**.

For solving **crossword puzzles**, the **SCHEMA** is the set of instructions one somehow acquires – by googling, reading, talking to friends, or thinking about the problem. Typical instructions include: “search first for short words,” and “an answer could be a phrase.”

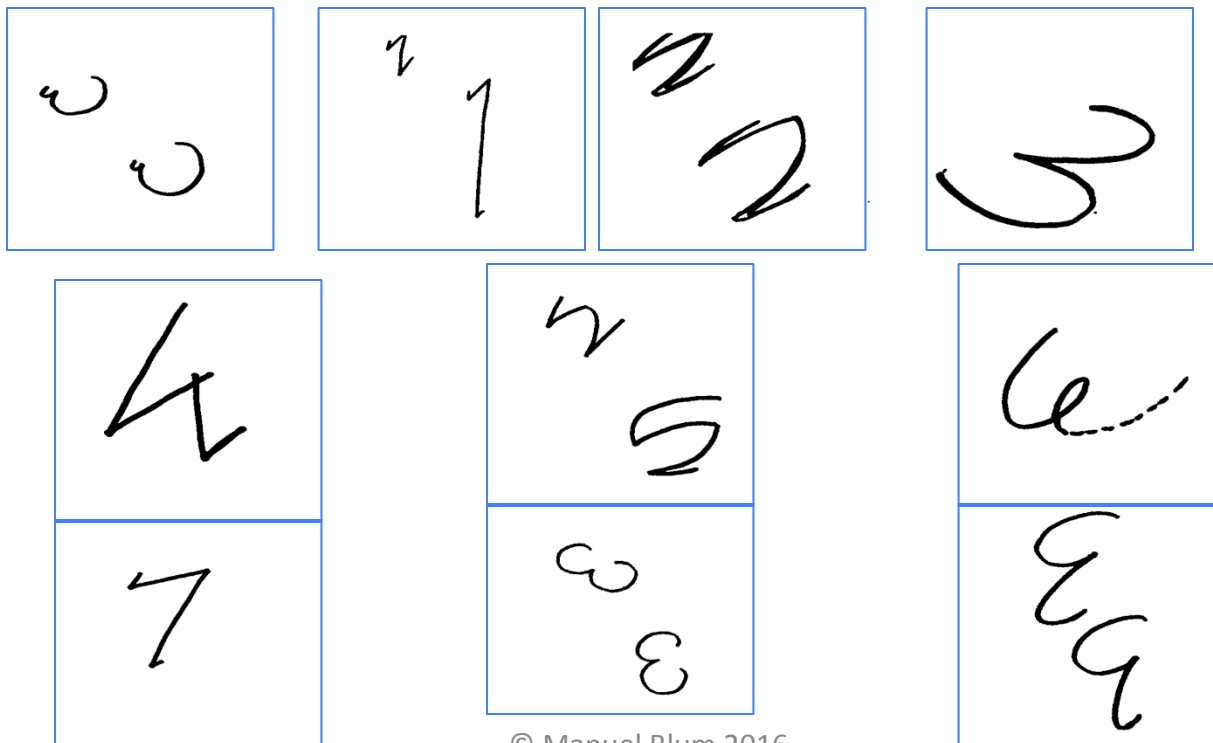
The **DATA** includes such stuff as a dictionary of short words committed to memory.

For **password generation**, the **SCHEMA** is a humanly usable (public i.e. publishable) algorithm and the **DATA** is a private **KEY** (a parameter of the schema).

Properties of PREP

Instructions must be as detailed for humans as they are for computers. **For example**, for humans to memorize a random map from letters to digits, I would create a 26x10 chart of letters to digits, then show only the 26 cells that must be memorized. Here are:

W -> 0, 1, 2, 3, 4, 5, 6, 7, 8, 9



Properties of PREP

A major challenge to schema designer is to **make required memorizations easy and fun.**

For example, to memorize 2 independent random functions from the standard 26 letters → 10 digits can be confusing.

- Instead, memorize a single random function

f : 26 letters → 2-digit numbers.



Then set $f_1 = \text{msd}(f)$ and $f_0 = \text{lsd}(f)$.

- Another possibility is to let F map capital letters and f map lower case letters to digits.

To estimate **PREP** time, **point out a familiar memorization activity of the same sort.**



Best is a scholarly study to measure time.

Definition of #PREP

#PREP := a triple consisting of

1. Work to **generate random numbers**, e.g. (number of tosses of a k -sided die) $\times (\log_2 k)$ for generation of numbers in $\{1, \dots, k\}$.
2. Work to **store data** in permanent memory. This depends crucially on the data structures used for storage, which must be specified. **Singly linked lists are much cheaper to memorize than hash functions.** State work as “a singly-linked list of length L ” or “a hash function of N items”.
3. Work to **store schema** in usable form in permanent memory, e.g. **number of lines of code** in the schema.

PROC

PROCessing (e.g. transforming challenge → response) is a fast execution of schema. For this reason, it does **NOT** write to **LTM**:

It may read and write to STM.

It may use the pointers in **STM** to **read** – but **not** write – **LTM**.

PROCessing uses a pointer from **STM** into schema (stored in **LTM**) plus it uses pointers into relevant data.

In password computation, **PRO**Cessing the schema includes cycling thru the challenge (the website name) and outputting the response (the password).

Our model requires schema have at most 3 pointers in STM.

This requirement is necessary for the schema to be humanly usable.

For **computation of a password, I bound the total number of steps a schema may take – so that the schema takes at most a minute to transform a 10 letter challenge into a 10 letter password.**

Definition of #PROC

#PROC := total number of steps to execute the given schema. This number counts the total number of atomic operations executed by the schema, e.g. *jump or conditional jump, reset a given pointer in STM to head of a given data structure in LTM, read an item in STM or LTM, write or rewrite a variable in STM, such as $x := x+1$.*

Properties of Schemas and STM

4.15. 2-3 pointers*. Pointers into a singly-linked list can shift right (or circle round to the start of the list) but they cannot shift left*.

4.17. The model enables to count **PRO**Cessing steps.

Steps are simple: each takes at most 1 sec.

**Typical
steps**

- Shift a pointer into a linked list, be it challenge or data, by one symbol to the right. **1**
- Reset a pointer into challenge to start or end of challenge. **1**
- Add or multiply 2 digits mod 10. **1**
- Add or multiply 2 digits mod 9 or 11. **2**

***Whence come these properties and rules?
From constructing password schemas.**

Properties of Schemas

Instructions must be as detailed for humans, as they are for computers.

- For example, instructions for storing data in **LTM** must assert whether the data is stored in a singly linked list (like the alphabet) or random access (like names of faces).
- In password generation, discuss possible **starting locations** in a challenge. For example,
 - Start 1 past 1st vowel.
 - Start 2 past 1st letter that has a vertical i.e. B D E F H
 - Start 1 past 2nd letter that rhymes with eee i.e. B C D E G P T V Z.
 - Start from 1st occurrence of a letter in SHRDLU.

In general, spell out all possibilities from which a random selection must be made.


Example 1: Telephone Schema

PREP: Store Schema. Create and memorize a random hash function (key) **h** mapping 26 letters \rightarrow 10 digits. Create and memorize random hash function (key) **T** mapping 10 digits \rightarrow 10 digits. For human usability, view **T** initially as a random 10-digit Telephone # **T** = 412 596-4063.

Treat challenge as a circular linked list.

PROC: (Input Challenge = CMU)

Evaluate Schema on Challenge:

	Steps	
• c := 1 st letter of Challenge (= C M U), a linked list	1	
• t := 1 st digit of Telephone # (T = 4 1 2...)	1	
• Repeat until T is exhausted (10 times):	1	
○ h (c) := mapped value of c.	1	
○ sum := h (c) + t mod 10	1	
○ Output sum	1	
○ Increment c	1	
○ Increment t	1	
		Roughly 20 to 60 secs
		
	Total =	53

Example 2: **STML (Skip-To-My-Lou) Schema;** **not to be confused with** **STM (Short Term Memory).**

- **STML** is our current best suggestion for humanly usable and secure (in a manner to be explained) “Pseudo Random Generators (**PRG**)”, 1-way functions, and Password Generators.
- It’s definition depends on the **STML** Subroutine.

6. The STML (Skip-To-My-Lou) Subroutine

- **Preprocessing:** Memorize a random f : letters \rightarrow digits.

The STML subroutine is easier to understand if the input, e.g.

AMAZING, is replaced by its mapped value: $f(\text{AMAZING}) = 1316947$

- **Processing:** Given a challenge consisting of a string of letters - now viewed as a string of digits - apply the following algorithm:
 1. Set **SUM** = last digit of challenge
 2. Set current digit (pointer) = first digit of challenge
 3. Repeat until current digit shifts past last digit of challenge:
 1. Set **SUM** = **SUM** + current digit (mod 10)
 2. If **SUM** is at least 5, output **SUM**
 3. Shift current digit in challenge one digit to the right.

STML Idea in an Example

- **Example:** We assume letters are hashed to digits.
- Suppose the challenge is **31415926**.
- The running sum is $6+3=9$, $9+1=0$, $0+4=4$, $4+1=5$... resulting in **31415926** \rightarrow **90450917** \rightarrow **9597**
- **Processing measure of complexity** = [apply map + set SUM + shift pointer + $(n-1) \times$ (apply map + add mod 10 + compare to 5 + output (maybe) + shift pointer)] = $3 + (n-1)(1+1.5+1+0.5+1) = \mathbf{5n-2}$.

STML Subroutine in an Example

- **Example:** We assume letters are hashed to digits,

123 456 7890
↓↓↓ ↓↓↓ ↓↓↓↓

T = 758 429 - 1360, T is memorized as hash function.

Suppose the challenge (post hashing) is **31415926**.

- The running sum is $T(6)+3=2$, $T(2)+1=5+1=6$, $T(6)+4=3$, $T(3)+1=9$... resulting in **31415926 → 59867978**
- The following computation not only **counts steps**, but also **shows in which order operations are performed**: **Processing measure of complexity** = [apply map + set SUM + shift pointer + $(n-1) \times$ (apply map + add mod 10 + compare to 5 + output (maybe) + shift pointer)]
= $3 + (n-1)(1+1.5+1+0.5+1) = 5n-2$ **steps**. For $n=10$, $5n-2 = 48$, time t is in the range **16 secs < t < 48 secs**.

6. The STML (Skip-To-My-Lou) Subroutine

- **Preprocessing:** Memorize a random **f**: letters \rightarrow digits.

Memorize a random **T**: digits \rightarrow digits.

The STML subroutine is easier to understand if the input, e.g.

AMAZING, is replaced by its mapped value: **f: AMAZING \rightarrow 1316947**

- **Processing:** Given a challenge consisting of a string of letters - now viewed as a string of digits - apply the following algorithm:
 1. Set **SUM** = last digit of challenge
 2. Set current digit (pointer) = first digit of challenge
 3. Repeat until current digit shifts past last digit of challenge:
 1. Set **SUM** = **T(SUM)** + current digit (mod 10)
 2. If **SUM** is at least 5, output **SUM**
 3. Shift current digit in challenge one digit to the right.

The $n \rightarrow 2n$ STML "PRG"

Wlg, we assume that the random map x from the 26 letters to the 10 digits has already been applied to the challenge, C , so C is a random seed/string of digits of a length n (to be determined).

The initial output $F_x(C)$ of the PRG is the output of Skip-To-My-Lou applied with key x to challenge C . This string $F_x(C)$ is not long enough to be the complete output of the PRG. Indeed, its expected length is just half the length of C . To produce a longer string, at the cost of one more (expensive) pointer, the idea is to run STML on many substrings of C and concatenate the results. 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, C_2 is the substring of characters obtained by alternately skipping two and including two, and so on... stopping when $n/2$ digits are skipped.

The STML (Skip To My Lou) Schema

Theorem: An adversary with an ordinary laptop who knows that the user is using STML on integer challenges of length 10, mapping 10 letter challenges to 20 letter passwords, can with high probability invert a password in just a few hours.

Open Question: Is it possible for a modern (2016) supercomputer working for a week to invert a 40 digit password?

Theorem (Michael Wehar): The STML schema, which takes n digit challenges to $2n$ digit responses, can recover a/the challenge from the response in $O(n^{13})$ steps). Consequently, STML is not what cryptographers would call a true pseudo-random sequence generator.

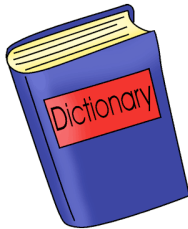
Part three Security

Part three Security

- **Big difference between computers and humans:** besides that they are much faster than humans in doing arithmetic operations, computers are constantly improving in speed and memory, while human replacements are the same old model. For this and other reasons,
- When counting steps to determine the complexity of schemas, **specific constants are important.** $O(n)$ is not good enough. Step counts must specify constants better, e.g. $c \cdot n + O(1)$ for specific c is better than $O(n)$.
- 1-way functions are examples. A typical 1-way function is multiplication of 2 primes. But any two numbers that a human can multiply in their head can be factored in a fraction of a second by a laptop. The whole question whether humans can compute 1-way functions (in their head) that a powerful supercomputer cannot invert is open. Is it possible? Is it even imaginable? Santosh's **STML** suggests a candidate schema to do it.

To deal with security, we define a measure of the Quality Q of a Schema.

The definition of a Schema's **Quality** assumes that challenges are sampled from a **Distribution** of challenges called the **Dictionary**, \mathbf{D} .



Examples: \mathbf{D} may be the set of all strings of length 10 on the 26 letters, each with probability $1/26^{10}$, or \mathbf{D} may be the set of all website names with their associated probabilities of being called.

Password Schema Quality Q

- Password Schema **Quality Q** is then defined in terms of a game between an "all-powerful" **adversary** and a trusted **judge**.
- The **adversary** is a Turing machine that is "**all-powerful**" in the sense that **1.** it has unbounded but finite running time, and **2.** it knows the public Schema, but **3.** it does **not** know the private Key.
- The **judge** knows the private key. Initially, the **judge** draws a random sample from **D** and gives it to the **adversary**. After each such draw, the **adversary** makes 10 guesses at the password. If none is correct, then the **judge** gives the **adversary** the correct password and draws another challenge. This repeats until adversary makes a correct guess.
- **Q** is defined to be the expected number of draws with replacement from **D** until the adversary guesses correctly.

The Quality Q of various Schemas

For the schema that maps each letter (like X) to a digit (like 4), independent of the location of X in the challenge (so $X \rightarrow 4$ implies $XXX \rightarrow 444$), we know that $Q = 7$. The lower bound ($Q \geq 7$) is info theoretic. The upper bound ($Q \leq 7$) is proved by supplying an optimal program for the adversary, due to **Elan Rosenfeld**.

The Quality Q of various Schemas

For the schema that maps each letter (like X) to a digit that depends solely on the letter and its position in the challenge (e.g. $XX \rightarrow 41$, $YY \rightarrow 59$; so $XY \rightarrow 49$), we know $Q = 14$. The lower bound of 14 is info theoretic, and the upper bound of 14, which is harder to prove, is again due to **Elan Rosenfeld**.

Future Directions

Major Open Question: Can a human generate pseudo-random sequences in her head?

More formally: Does there exist a publishable schema such that a human with just **A FEW HOURS PREP** (to learn the **public schema** and to generate and memorize a **private random key**) and **A FEW MINUTES PROC** per input seed can transform a dozen **private random 20-digit numbers/seeds** into corresponding **40 digit numbers** such that a 2016 supercomputer running for an hour cannot distinguish the dozen 40-digit pseudo-random numbers from a dozen truly random 40 digit numbers?

Manuel's principal interest is to extend the model of human computation to applications in which the computation is **not** done entirely in the head.

Santosh's principal interest is to investigate how far one can go in doing cryptography in one's head. He looks to construct secure **humanly usable Pseudo Random Generators, 1-way functions, Trapdoor functions**, etc.

