# Random Seed Encryption

Alexander R. Frieder[*]
21,876,528 782,847,088,000[†]

April 1, 2016

## Abstract

We present a novel form of encryption, wherein the encryption step is computationally expensive while the decryption step is more or less trivial. We will describe how the scheme works, proving its correctness along the way. We will then show that this encryption scheme has many uses, for example, prevention of texting while intoxicated or sender identification over email. We will also acknowledge the limitations imposed upon our beautiful scheme by the cruel harsh mistress of reality. Finally, we will discuss the immeasurable impact this algorithm could, nay, will have upon the world.

## 1 Introduction

It is universally understood among nearly all experts in the field of cryptography [1] that any good cryptographic encoding function has four fundamental properties:

- it is quick to compute the hash value for any given message
- it is infeasible to generate a message from its hash
- it is infeasible to modify a message without changing the hash
- it is infeasible to find two different messages with the same hash

[*]email: alex.frieder@gmail.com
[†]This is the encryption of the author's name using C's default long-seeded PRNG.

We, however, like to go against the grain in terms of what is "universally understood". Thus, we posed the question: what if the first two conditions were switched?, that is, what if we had a hash function where it was nearly infeasible to compute the hash value, but quick to generate a message from its hash? Thus Random Seed Encryption (RSE)(not to be confused with the lesser known RSA) was born.

We have come up with one such function and will spend the rest of this article analyzing and disucssing it, with applications in encrypting words, though any datatype could theoretically be encrypted.

## 2 Encryption Algorithm

The encryption algorithm takes a word and calculates two associated numbers for it: the length, $\ell$, and a seed, $s$, to a pseudorandom number generator (PRNG) such that $s$, is the smallest seed such that the next $\ell$ numbers generated, when converted to letters, spell the word. These two numbers are mapped to a single number, which is the encryption of the word.

### 2.1 PRNG to Letter Conversion

In general, PRNGs output either a random real number between 0 and 1 or a random integer between 0 and $N$. Any function that takes one of these numbers as input and deterministically returns a letter, where the probability across all numbers in the domain of getting any given letter is approximately $\frac{1}{26}$ will suffice. However, since constructive proofs are all the rage these days, we will briefly discuss our chosen functions.

To convert a random real $0 \leq r < 1$ into a letter, partition the space $[0, 1)$ into 26 sections, $[0, \frac{1}{26})$, $[\frac{1}{26}, \frac{2}{26})$, $\ldots$, $[\frac{25}{26}, 1)$. If your random number falls into the $i$th section, then your letter is the $i$th letter. This clearly works. Even more trivial is the proof that every letter has an equal probability.[1]

To convert a random integer $0 \leq n < N$ into a letter, take $i = n \bmod 26$ and use the $i$th letter. For $N \geq 2500$, the difference from $\frac{1}{26}$ is less than 1%.[2]

Now we know how to convert random numbers to random letters and we can move on to finding the right random numbers for our specific letters.

## 2.2 Finding the Correct Seed

PRNGs are not actually random and honestly, they are not fooling anyone. PRNGs just produce a very very very very long list of bits that are converted into numbers in some range. And this list of bits is always the same! The trick used is to start at different points in the list so it at least seems random. This starting location is called the seed.

If you set a PRNG to a specific seed and generate a list of random numbers, then reset to the same seed and generate another list of random numbers, the two lists will be identical. Our goal is to find a specific seed that will produce our word.

For example, if you set the default Python PRNG to seed 1,944,062 and generate five random numbers, converting them to letters as discussed above, you get "hello", which is a nice message many people would want to send, probably. How do we find seed 1,944,062?

Well, you can set the seed to 0, see that the word of length 5 generated at this seed is "vtkgn", which is not "hello", set the seed to 1, see that the word of length 5 generated is "dwtgm", and repeat until you set it to 1,944,062.

Surely this naïve brute force algorithm is just the first in a set of stepping stones to some more intelligent algorithm, no? No. The whole point of a PRNG is that it seems random. Knowing these five letters tells you nothing about the next five, or where any given five letters will be.

There is no way to determine where some given letters will appear in a sufficiently complex (known in the technical world as "cryptographically secure") PRNG without trying every possibility until they are found.[2][3] That is what makes this algorithm so powerful.

## 2.3 Merging the Seed and Length

Having the correct seed, $s$, is not enough; you also need the length of the word, $\ell$. We could of course just encrypt every word as two numbers, but that means we need twice the space. Instead we will use the well-known bijection between $\mathbb{N} \times \mathbb{N}$ and $\mathbb{N}$, $f(x, y) = 2^x(2y + 1)$. We will use $f(\ell, s)$ since $\ell \ll s$ and this function grows exponentially in the first argument, but linearly in the second. Any other bijection between $\mathbb{N} \times \mathbb{N}$ and $\mathbb{N}$ will work, but we use this one since it is clearly the best.[3] Thus, we now have all the machinery necessary to take a word and fully encrypt it into a single number.

# 3 Decryption

Decrypting a word given the encrypted number is extremely easy. Looking at the bijection we used, the length of the target word is exactly the number of times we can divide 2 into the encrypted number. The remaining odd number is one more than twice the seed.

For example, if our encrypted number is 124,420,000, we halve it as many times as we can. 32 divides this number evenly while 64 does not. Thus the length is $\log_2 32 = 5$.

[1] Proof left as an exercise for the reader.

[2] Proof left as an exercise for the reader. But actually it is true.

[3] This result is probably shown in this citation. It is reallllllly long and we reallllllly do not want to read it. But it is there. we promise.

We are left with 3,888,125. The seed is thus $\frac{3{,}888{,}125-1}{2} = 1,944,062$.

Now that we have our seed and length, we set our random number generator to that seed and generate 5 random letters. In this case, we get the letters, in order, “h”, “e”, “l”, “l”, and “o”, which spells “hello”. Thus the number 124,420,000 uniquely encodes the word “hello”.

## 4 Discussion

We have fully described how and why this encryption method works, but as with all theoretical algorithms, there are some limitations due to the realities of physics and computers.

Every PRNG has some finite period, after which the PRNG starts repeating values. This places a hard upper bound on the number of words we can encode with a given PRNG. Additionally, we are further bound by the number of unique values we can use for the seed.

For example, when the seed is an integer, commonly represented by 4-byte values, we can uniquely represent $2^{32} - 1 =$ 4,294,967,295 seeds and thus words. In this case, it is sufficient to represent all 12,356,631 words of length 6 or less, but not enough to represent all 8,031,810,176 words of length 7. Similarly, using 8-byte longs as the seed we can represent all words of length 13 or less. However, 13 is a big number and a lot of words can be represented using less than 14 letters. In fact, not a single word in this paper has more than 13 letters. So there.

Additionally, there is no guarantee that a given word will actually appear in a PRNG. For sufficiently long words, a given sequence of random numbers might just not be an output of the PRNG. However, since most combinations of letters are not words, this is likely not to be a problem in actuality.

## 5 Applications

Random Seed Encryption clearly has numerous significant uses in modern life. One such use is to prevent drunk texting/messaging. Studies have found that drunk texting friends, former lovers, family members, and even bosses can have terrible terrible effects on one’s life.[4] Even worse is drunk posting on Facebook, reddit, or the reader’s social media website of choice. Imagine some kind of filter you can turn on, where for the next $k$ hours, anything you want to send to someone or post somewhere has to be encrypted using this method. Of course, no one under the influence could ever manage to correctly encode even a single word! Thus drunk messaging is prevented.

Another possible application is that of sender verification. Many people try and verify their identity over email using PGP encryption, but that is a lot of work for the receiver, and honestly, we all know no one really cares that much. However, sending someone a message under this encryption scheme has a two-fold benefit over PGP: it is much easier to decrypt, so the receiver might actually bother with it, and since RSE is so new, no one else in your social circle except you will even have heard of it, so the encryption is definitive proof that you actually sent the message!

Thus we have demonstrated that not only does RSE have real world applications, but it has numerous benefits over other, more commonly used encryption schemes. We hope this will encourage the cryptographic community to spread the good word about Random Seed Encryption.

## 6 Future Work

There are numerous ways this novel direction of research can be expanded. For example, we would love to consider:

- This algorithm using “true” random number generators. This would obviously complicate matters since seeds no longer exist, but we are sure some bright, spunky undergraduate student could figure out a solution.
- Encryption schemes where both the encryp-

tion and decryption steps are computationally expensive.

- Alternative encryption functions that are, for example, $\Omega(2^\ell)$.
- Something involving quantum computers working on big data in the cloud

We believe this paper may be the dawn of a new field of computer science, which we have preemptively named the Friederian Sciences.

## 7 Acknowledgements

We would like to thank a handful of people: Zachary Piscitelli, who came up with the original name for this paper, which I have since changed, but I feel bad taking him out of this section; Thomas Bayes, without whom none of this work would be possible, for very obvious reasons; you, the reader, for getting this far; and finally, my computer, for spending hours upon hours trying billions upon billions of different seeds.

## References

[1] Wikipedia contributors. "Cryptographic hash function." Wikipedia, The Free Encyclopedia, 14 Jan. 2016. Web.

[2] Röck, Andrea. "Pseudorandom Number Generators for Cryptographic Applications." Thesis. Paris Lodron University of Salzburg, 2005. Web.

[3] Sutner, Klaus, Teaching Professor and Associate Dean for Undergraduate Education, School of Computer Science, Carnegie Mellon University.

[4] Bartholow BD, Henry EA, Lust SA, Saults JS, Wood PK. Alcohol effects on performance monitoring and adjustment: affect modulation and impairment of evaluative cognitive control. *J Abnorm Psychol.* 2012 Feb;121(1):173-86.