# Pretty Frecking Strong Encryption

Louis Cordier <lcordier@gmail.com>
6 September 2009

*Abstract*— **In this paper I'll show that we can address the key distribution problem of one-time pads in a viable and practical manner. Instead of distributing a key, we will (publically) distribute a recipe to make a key. The two parties communicating will have shared prior knowledge of a secret ingredient used in the construction of the key. The attacker, armed with only knowledge of the cipher-text and the recipe, will have to guess the secret ingredient. I will then show that the secret-ingredient-space is with all likelihood in the order of aleph-null.**

## I. Introduction

In 1917 Gilbert Vernam invented the one-time pad [1], later Claud Shannon showed it had a property of *perfect secrecy*. For this to work, it is vital that the keystream values (a) be truly random and (b) never be reused [2]. Additionally, the keystream must be as long as the data to be encrypted and has to be shared *securely* with the recipient. Sharing the key stream securely is known as the Key Distribution problem [3].

## II. Assumptions

We will assume all clear text messages will always be compressed at the maximum compression ratio to increase message entropy.

## III. Key Recipes

A key recipe will be a string that references public resources (ingredients) and specify the size of the key to be created. Recipies can be encoded in any language, obfuscated in any format. However, we will assume the attacker will always know the format and language in which recipies are encoded, and always have access to all recipes.

### A. URL Recipe

For my first recipe encoding scheme, I decided to stick to plain URLs. In general we will use the fragment of an URI. In RFC3986 section 3.5 [4] — a fragment identifier component is indicated by the presence of a number sign ("#") character and terminated by the end of the URI. Thus we can embed URLs (without fragment sections) into the fragment section of an URI. These URLs can be delimited with a number sign ("#"). For example, an URL encoded recipe could look like:

- `http://www.google.com/#http://example.com/ingredient1#http://example.com/ingredient2`

This recipe can then be obfuscated using URL-shorteners, for example tinyurl and bit.ly. Thus the example recipe would now look like:

- `http://tinyurl.com/nhzyue`
- `http://bit.ly/NKsoX`

If we always use the same URL-shortener (even privately controlled ones), we simply have to send the last 5 characters. We could SMS it, tweet it, hide it [5], [6], [7], morse code it or even publish it as a coupon code in a newspaper ad.

### B. Email Recipe

Email recipes can be written in natural languages wich will make automated techniques to parse them difficult. For example:

```
From: Bob <bob@example.com>
To: Alice <alice@example.com>

Here is a nice article you might like.
http://www.example.com/recipe.html
Use the first two images on the page as
your ingredients, and that hot image
of you at summer camp as the secret
ingredient ;).

Regards, Bob.
```

## IV. Key Ingredients

Ingredients can be thought of as data, sampled in an endless loop. The loop is not really infinite, it just loops until we gathered enought values to construct the required length key. Ingredients can also be code or algorithms that generate values, PRNGs [8] for example.

> Progression of a Lisp programmer — the newbie realizes that the difference between code and data is trivial. The expert realizes that all code is data. And the true master realizes that all data is code.

### A. Conditional Access Ingredients

Since ingredients will most likely be stored on public webservers, we could implement features like access control. Think of an AppEngine application that returns a file of requested size, $N$. The file is dynamically created as the first $N$ outcomes of an PRNG, optionally modulated by some algorithm/state-machine, seeded with the requesting machine's IP address. If Alice is constructing a key and requesting her ingredients over SSL the attacker will have to request his own set of ingredients. If the public webserver shows stats of how many times resources have been accessed, Alice can tell if an attacker is trying to construct her key. An example of this would be to see how many people accessed our recipe.

- `http://bit.ly/info/NKsoX`

### B. The Secret Ingredient

Ingredients can be both data and code. The secret ingredient, the one only Alice and Bob knows, should ideally be some algorithm. This algorithm takes a keystream $K_i$, constructed from the ingredients listed in the recipe and mutates, transforms or transcode it into our final keystream $K'_i$. $K'_i$ is the keystream we use to encode and decode our message. The attacker will have to determine a secret algorithm of arbitrary complexity. To make things even more difficult, within each message we can include a new secret ingredient (algorithm) to be used in the response message. These algorithms can be general functions with sets of parameters that control their behaviour. The controling parameters can then be selected at random within bounds. Here is a few example secret ingredients:

*1) Simple Function:* Our secret ingredient is the function $F$ such that $K'_i = F(K_i)$. Where: $K'_i = (K_i + 3)\ MOD\ 255$ when $i$ even and $K'_i = (K_i + 5)\ MOD\ 255$ when $i$ is odd. The function $F$ is secret, but it is also in a general form with the parameters (3,5) that we could have chosen at random. So instead of sending a new secret function $F$ in each message, we could simply only send new contol parameters.

*2) PRNG:* In this case our secret ingredient is a function that interacts with a PRNG, lets say Mersenne Twister. We seed it with an secret parameter $s$ and we ignore the first $n$ outcomes. Then $F$ is the function such that $K'_i = K_i \oplus MT(s)_{i+n}$ where $MT$ is the Mersenne Twister with outcomes bound to the bit-width of $K_i$.

## V. Key Construction

OTPs require that the key be of the same length as the original message. For a 1TB message we will need a 1TB key. Thus a length-limited recipe must be turned into an arbitrary length key. Think of this as an inverse-one-way-hash function, given a hash (recipe) it will return the minimum-sized data that will produce that hash (desired key). Constructing a key is the process of mixing ingredients in their various quantities, and then altering the mixture in some way (baking it) to produce the final result.

### A. Mixing Ingredients

The easiest way to mix ingredients would be to cyclicly multiplex them. For fixed length data, put it in a circular buffer of the same size as the data. Then to generate the intermediate keystream $K_i$ simply select, in a cyclic ordered fasion, values from each ingredient buffer. Thus we have a length-unlimited keystream.

If the ingredient is an algorithm, simply pass it an unlimited time-series as input. For example the sequence $1, 2, 3, \ldots$

If our secret ingredient is data instead of an algorithm, say an image taken on a digital camera when Bob and Alice are together, we could multiplex it with our other ingredients. We then wouldn't need to alter the keystream further, $K'_i = K_i$.

### B. Baking The Mixture

Since the attacker has our recipe, he can with all likelihood always construct the intermediate keystream $K_i$. It is therefore vital that we alter $K_i$ in some secret way to produce our key $K'_i$. We require that it MUST be computationally unfeasible for the attacker to find our secret. To that end we choose an arbitrary algorithm to mutate $K_i$ into $K'_i$. I would venture to guess that there are at least as many possible algorithms as there are integers and way more than Prime numbers. Thus my secret ingredient search space is at least of the order $Aleph_0$.

We can write it as equations:

$$K_i = Mux(I_1, I_2, \ldots, I_n, i) \quad (1)$$
$$K'_i = F(K_i, i, \ldots) \quad (2)$$

Which states the intermediate keystream $K_i$ is the multiplexed ingredients $I_1, I_2, \ldots, I_n$, and the final keystream $K'_i$ is the result of a mutation function $F$, our secret ingredient, acting on the intermediate keystream.

## VI. Conclusion

I have shown that the key distribution problem could trivially be addressed with the (public) distribution of key generation recipes and a shared secret. The shared secret could be data or an algorithm and its size extremely small compared to the message to be encryptred. I also showed a protocol where new shared secrets get distributed with each message, to be used in the response of that message. Thus to establish and maintain *perfect secrecy* you only need to distribute one *small* secret. That secret could be obfuscated [5] and hidden [9] in physical "unbreakable" objects, couriered around the world.

This leaves us with a moral dilemma. Should this genie be left out of the bottle? I could patent it in such legal speak that no one could implement it, and then sell various hardware solutions to governments to be used by their agents in the field. If I don't release this, someone else is bound to come up with a similar solution. Should the public, terrorists and pedophiles have perfect secrecy?

## References

[1] G. Vernam. (1917) One-time pad. [Online]. Available: http://en.wikipedia.org/wiki/One-time_pad

[2] B. Schneier. (2009, September) The History of One-Time Pads and the Origins of SIGABA. [Online]. Available: http://www.schneier.com/blog/archives/2009/09/the_history_of.html

[3] Key Distribution. [Online]. Available: http://en.wikipedia.org/wiki/Key_distribution

[4] T. Berners-Lee, et al. (2005, January) Uniform Resource Identifier (URI): Generic Syntax. [Online]. Available: http://www.ietf.org/rfc/rfc3986.txt

[5] 42 ways to distribute DeCSS. [Online]. Available: http://decss.zoy.org/

[6] Steganography. [Online]. Available: http://en.wikipedia.org/wiki/Steganography

[7] Spread Spectrum. [Online]. Available: http://en.wikipedia.org/wiki/Spread_spectrum

[8] Pseudorandom Number Generator. [Online]. Available: http://en.wikipedia.org/wiki/Pseudorandom_number_generator

[9] The Hunt for the Kill Switch. [Online]. Available: http://www.spectrum.ieee.org/semiconductors/design/the-hunt-for-the-kill-switch/0