Hush Functions Extended to Any Size Input versus Any Size Output

"Crypto Square": $Z^+ \rightarrow Z^+$ with Infinite Solutions to the Crypto-Square Root Equation

Gideon Samid
Case Western Reserve University, Electrical Engineering and Computer Science Department
gideon.samid@case.edu

Abstract: Traditional hush functions map a large number to a small number such that the reverse-hush has an infinity of solutions, and nonetheless a collision is hard to come by. This primitive is so abundantly useful that one is tempted to extend it such that any number large or small may be mapped to any number larger, or smaller while maintaining the above conditions. This extension would increase the flexibility of the commodity hush primitive, expand its current applications, and likely suggest new ones. Additional generality may be achieved by allowing the input to determine the computational burden, and involving Turing's Entscheidungsproblem. We propose an algorithm where a natural number, X, is mapped to another natural number Y, referring to the mapping as a "Crypto Square", and to the reverse as "Crypto Square Root": $Y = X^2|_c$ and $X = \sqrt{Y}|_c$. While the crypto-square mapping is a proper function, the square root equation has infinite solutions. There exists a deterministic solution algorithm to find any desired number of solutions to a square-root equation. This asymmetry proves itself useful, since the mapping is $Z^+ \rightarrow Z+$, and hence the chance of collision for any finite size set is negligible. Unlike standard one-way functions, crypto-square shields the identity of the input (X), not by the intractability of the reverse function, but by Vernam-like equivocation per the infinity of X candidates. This prospect suggests further examination of this "square" algorithm for possible useful roles in various crypto protocols, especially protocols concerned with privacy, authentication and deniability.

0.1 Introduction

Traditional hush functions map a large number of an arbitrary size to a small number of a fixed size. An infinite of input numbers share the same output (hush value), yet a collision is hard to come by. This primitive is so abundantly useful that one is tempted to extend it such that any number large or small may be mapped to any number larger, or smaller while maintaining the above conditions. Such an extension would increase the flexibility of the commodity hush primitive, expand its current applications, and likely suggest new ones.

The obscurity of size is of great interest. Given an input to the mapping, it is not a-priori known if the result will be of similar size, smaller, much smaller, larger, or much larger. And similarly for the reverse: given the output of the mapping, it is not clear whether the input was of similar size, smaller, much smaller, larger, or much larger. If one also keeps the computational burden undetermined to the point that it may not be known if the mapping is computable by polynomial time, then a very interesting and potentially very useful algorithm is at hand.

Following the idiosyncrasies of the classic Turing machine this desired algorithm should be governed by very simple set of rules, while packing the detailed instructions in the input number itself.

We propose an algorithm, called Crypto-Square that fits this objective. It maps $Z^+ \rightarrow Z^{+.}$ By stretching the domain and the co-domain to the full spectrum of natural numbers, (or say to strings of any length), one achieves two seemingly contradictory objectives: (1) given the output of the crypto-square, it is virtually impossible to determine the actual input because it does not distinguish itself mathematically from the infinity of input values that crypt-square to the same output, and (2) given any finite set for input candidates, there is negligible chance for a collision. So Alice can prove possession of a PIN to Bob by sending him the crypto-square result of her PIN concatenated with a session random number Bob has just given her. Bob will be convinced by the result that Alice holds the right PIN, and eavesdropper, Eve will not be the wiser as to the identity of the PIN, even if Alice and Bob repeat this authentication protocol, (each time with a different random number), any finite number of times.

A clear deviation from common hush functions comes to the fore: The crypto-square does not offer pre-image intractability. On the contrary, one can readily find an input X that would square to a given Y. This difference seems to wipe out the use of the crypto-square for applications where the above pre-image resistance is critical. However, crypto-square could be readily modified to offer strong pre-image resistance: given an input X, one will compute: $Y=X^2|_c$ and also compute $Y' = (X+r)^2|_c$ for some arbitrary number r. While it would be easy to solve: $X = \sqrt{Y}|_c$. and solve $X' = \sqrt{Y'}|_c$ it would be infeasible to also satisfy: $X' = X+r$

More applications suggest themselves based on the above conditions.

Crypto-Square relates to the common one-way functions, as Vernam cipher related to AES, DES, RSA, and other mainstay ciphers. The latter are based on intractability, the former on equivocation. The latter erode, the former endures. The Crypto-Square algorithm may be based on a sequence of encryptions and decryptions, where the former increases the size of the input string, the latter reduces it. Both the encryption and the decryptions are performed with a cipher that may take any size key, and where the key and the cipher-input string are both derived from the number that is being crypto-squared. That way one could take a squared result, apply any key of choice, process the squared result to get a cipher input string that would square into the same squared result. And since the chosen key is arbitrary, there are infinity of solutions to the square-root equation, proving the attribute of equivocation which keeps the cryptanalyst confused as to which of the infinity of possible values was squared to the given value.

The article first described the concept, then presents a solution to the concept based on a proper cipher, and subsequently overviews the Samid cipher, and it shows how it can fit for the purpose at hand. Some notes regarding use conclude this presentation.

1.0 Crypto-Square – Conceptual Description

Let us consider a class of algorithms to be called "Crypto Square", or "square" for short, to be denoted as:

$$1.\quad Y = X^2|_c$$

Where:

$$\textbf{2.} \quad .... X \epsilon Z^{+} \rightarrow \; Y \epsilon Z^{+}$$

And where the reverse equation:

$$3. \quad X = \sqrt{Y}|_{c}$$

Has infinite solutions. Let us further constrain the crypto-square algorithm as follows:

Let $h(x)$ be a histogram over a range: $0 \le x < \infty$ expressing a finite set $X_1, X_2, ... X_n \epsilon Z^{+}$ via some arbitrary interval I marked on the range. Let $h(y)$ be a histogram over a range: $0 \le y < \infty$ expressing the corresponding $Y_1, Y_2, ... Y_n \epsilon Z^{+}$ over the same interval, I. We shall denote an histogram as "flat" if for a sufficiently large interval I the ratio between highest histogram value, and the lowest histogram value, is below a pre-defined flatness ratio F_0:

$$4. \quad h_I(z)_{max} / h_I(z)_{min} < F_0$$

where z is either x or y. Accordingly we shall articulate 'forward flatness' condition to say that for any x-histogram h(x), the corresponding y-histogram will be flat, and add the 'backward flatness' condition to say that for any given Y value the x-histogram of its solutions will be flat.

An algorithm that satisfies the above may serve as a 'perfect one-way function' since the value of Y provides zero knowledge regarding the identity or even the likelihood of the value of the particular X that was squared into Y.

We now ask: (1) does a crypto-square function exist? And if yes (2) how to find it, or an instant of it?

Because the squared result may be bigger, or much bigger than the squared number, or it may be smaller, even much smaller than it, we are drawn to define the crypto function as a variable series of component-functions, one expanding its input, and one contracting it. If we will let the value of X to determine the sequence of expansion and contraction, we may achieve the desired variability. We may define the component-function within the crypto-square as crypto-addition:

$$5. \quad X' = 2X|_c = [X+X]|_c$$

Where X' may be larger, or smaller than X. And hence:

$$6. \quad \mathbf{Y = X^2|_c = 2^m X|_c} = 2(2(....2X))...) \text{ m times}$$

For some value of *m*, identifying how many times the crypto-addition will be applied in order to compute the crypto square result. The value of *m* may be predetermined, and in the case the computational burden is well bound. Alternatively, it may be defined per the history of the addition steps. We have two types of additions, as determined by the input string: expansion (Exp), and contraction (Con). One can define the crypto-square such that the addition steps would continue until a certain expansion-contraction pattern is registered. This will invoke the famous Entscheidungsproblem, where a given input X might require interdminate measure of computation (unknown number of addition steps). The longer the pattern, the more steps are likely before it is being registered. If Crypto-square will involve a series of addition steps until the pattern of: "Exp-Con-Exp-Exp-Exp-Con-Con-Exp" is achieved, then it will involve more steps compared to the case where m will have to register a subset thereof, for instance: "Con-Exp-Exp-Exp-Con"

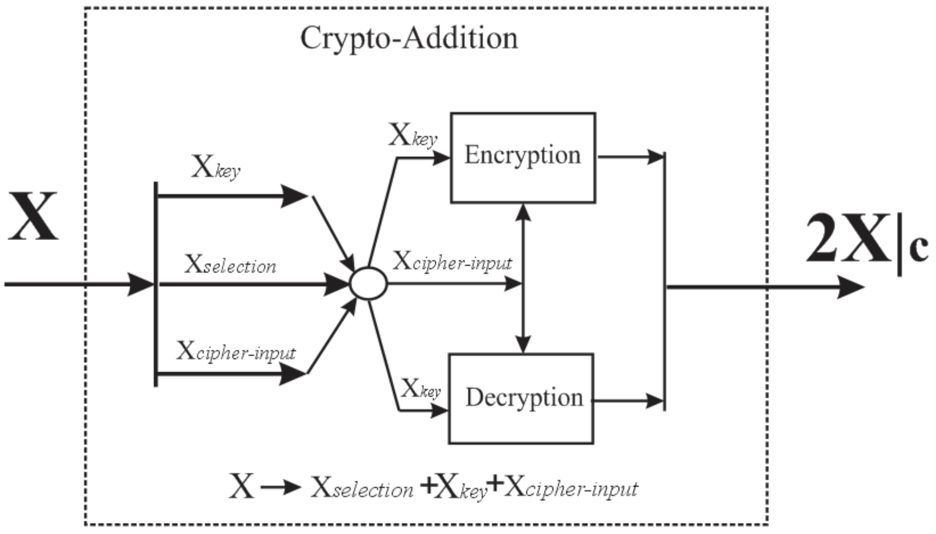
1.1 Implementing the Crypto-Square with an Expansion-Contraction Cipher

1.1.1 Concept:

Let us consider an expansion-contraction-cipher (EC-cipher), as one where the size of the ciphertext may be larger than the size of the plaintext:

$$7.\quad |C| \geq |P|$$

By subjecting an input number X to a certain pattern of sequences of encryption, and decryption, one would achieve expansion/shrinking variability as desired. By dividing an input string *X* to a part that would serve as either plaintext, or ciphertext, as the case may be, while the other part will serve as an encryption or decryption key, as the case may be, we may process any binary string through an encryption or decryption cycle, using only the information contained in X. If we further make the choice between encryption, and decryption to be extracted from the value of X, then we will have a desired crypto-addition cipher:



Input string X is mapped into X-selection, X-key, and X-input. The X-selection determines whether the X-input will be encrypted or decrypted, using (in either case) X-key as the cipher key. The result of either the encryption or the decryption (whatever was selected) is the output of the crypto-addition algorithm.

fig-1

Since the crypto-square algorithm is comprised of a series of crypto-additions, it is necessary for the addition component to also have the attribute that its reverse (crypto-subtraction) will have infinite solutions:

$$8.\quad \forall\, X' = 2X|_c \;\; \exists \,\{X_i = X'^{-2}|_c\} \;\; i=1,2,\ldots \;\; (Xk \neq Xj \text{ for } k \neq j)$$

This condition can be achieved if the selected cipher will be accepting variable size keys. Under these conditions, the above-described crypto-addition can be readily, arbitrarily reversed:

Given a string X'=2X|c, one would arbitrarily select an X'*selection* string that selects between encryption or decryption, then select an arbitrary string X'*key*, and use it to encrypt or decrypt the given X' (If encryption was arbitrarily picked for the respective addition, then the constructed subtraction will process decryption, and vice versa). The result of encrypting, or decrypting X' with X'*key*, is some output string, denoted as X'*input* – because it traces back a valid input string that can be encrypted or decrypted to X'. One now uses the reverse of the derivation algorithm that was used to derive X*selection*, X*key*, and X*input* from X, and reconstruct X" from X'*selection*, X'*key*, and X'*input*. By virtue of its construction X" will be crypto added to X':

$$9.\quad X' = 2X''|_c$$

But since X'*key* was arbitrarily selected, we can state that X' ≠ X, and hence it is another solution to the reverse-addition equation. And since we can repeat the above process as many times as we please, each time use a different key string X'*key*, combined with a different encryption/decryption selection string, it follows that one can derive an infinite number of solutions to the reverse-addition algorithm. And since the crypto-square algorithm is a succession of crypto-additions it follows that the square-root equation too has infinite solutions. If we also determine the value of *m* – the number of successive additions to be of sufficient measure, then it follows that the solution to the square root equation may be of infinite size. Say, the decryption process halves the size of the string, then a squared result Y achieved after *m* additions, would have as a solution a string of size $2^m Y$, or less, suggesting (not proving) compliance with the 'backward flatness' requirement for the perfect crypto-square process.

1.1.2 Specification:

We present here a procedure to effect a crypto addition that is specifically designed for one to be able to work back from a squared result to the squared number. There are numerous such procedures.

It works as follows: Let X be a number to be crypto-self added into $2X|_c$. And let X be written with x bits.

Step 1: XOR the rightmost bit with the leftmost bit, to determine the so called *ed-bit*. (X*selection*). If *ed=1* then the addition will be comprised of encryption; if *ed=0* then the addition will be comprised of decryption.

Step 2: remove the rightmost and leftmost bits, *(x-2)* bits remain, comprising string X'.

Step 3: Let *t* be the smallest natural number for which $2^t \geq (x\text{-}2\text{-}t)$. Let T be the numeric value (normal binary interpretation) of the left most *t* bits in X' . Such that if *t=5*, and the 5 leftmost bits are: "01001" then T=9. Let *T' = T mod (x-2-t)*. Now mark the substring of X' beginning from bit *(t+1)*, and ending with bit*: (t+T')*. This marked string will be regarded as the input string , or say the *"pc-string"*. Next mark the substring of X' beginning with bit *(t+T'+1)*, and ending with the last bit of X'. This marked string will be referred to as the *key-string*.

For illustration: Let X be 25 bits long:

X: 1111110010010110110001110

We XOR the rightmost and leftmost bits: 1 *(X)* 0 = 1, and the result indicates that this addition will be comprised of encryption, not decryption. We now remove the rightmost and leftmost bits and remain with string:

X': 11111001001011011000111

which is 23 bits long. The value of *t* is determined from the condition:

$$2^t \geq (23-t)$$

t=1,2,3,4 are too small, and t=5 is the smallest value of t that satisfies the above inequality. The 5 left most bits are: 11111, therefore T= 31, and hence: T' = 31 mod (23-5) = 13 so now the plaintext (the “pc” string) will be the substring from bit number 6=5+1 to bit: 18= 5 + 13:

plaintext (“pc” string) = 0010010110110

and the cryptographic key, will be the substring from bit 19 = t+ T'+1 = 5 + 13 + 1 to bit 23:

Key = 00111

The interesting property of this algorithm is that it lends itself to a ”work-back”. Given $Y = 2X|_c$, one could pick a random string as key, K, and apply the addition cipher to decrypt Y, using K. The result, U:

$$U = D(Y,K)$$

One would now concatenate U with K: ‘U-K’. Let U be comprised of *u* bits, and K of *k* bits. One would then find the lowest value of *t* for which:

$$2^t \geq (u+k-t)$$

And then add to ‘U-K’ on the left *t* bits such that their value T will satisfy:

$$T \bmod (u+k) = u$$

resulting in the string ‘T-U-K’ (concatenated).

Last, one would add "1" as the leftmost bit of TUK, and "0" as the rightmost bit of TUK (or vice versa), and the resultant string: V = "1-TUK-0" when added to itself will generate Y:

$$2V|_c = Y$$

which is almost certainly not the actual X that was self added to Y ($V \neq X$), but one could claim that he self-added V, while in fact he has self-added X, and since there is no mathematical difference between X and V, such claim will provide robust deniability.

Fig.-2 illustrates the above.

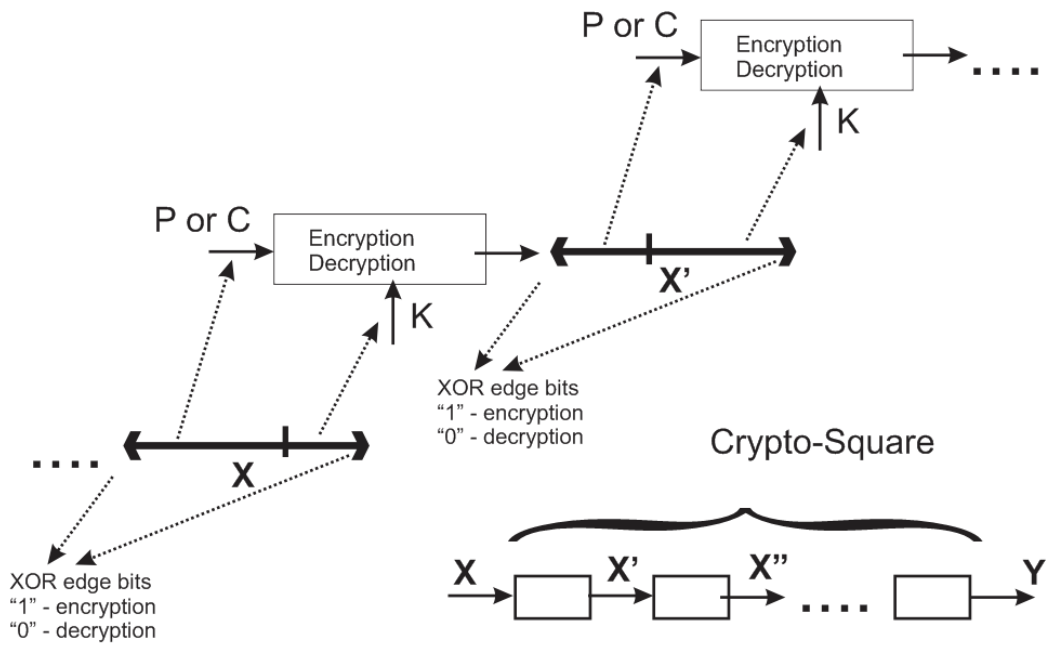


fig-2

2.0 Crypto-Square based on the Samid Cipher

The Samid cipher is described in US PTO 6,823,068, and in [Samid 04]. A limited review follows to help elucidate its use for the purpose at hand.

2.1 Review of the Samid Cipher

In a particular limited implementation of the Samid cipher one is applying a three-letter alphabet to represent any number of characters, and symbols. For example, 243 characters and symbols (almost as many as in the expanded ASCII table) can be represented by 3^5 =243 strings, where each string is comprised of five ordered letters, where each letter is either 'X', 'Y', or 'Z'. This will represent any plaintext as a 4 letters string of some length. One now eliminates all letter duplication in this plaintext string by inserting a fourth letter, 'W' between any adjacent double. Namely all 'XX', 'YY', and 'ZZ" in the plaintiff string are replaed with 'XWX, 'YWY', and "ZWZ'. The new (longer) string has no occurrence of two adjacent same letters.
By assigning: 'X'= 01, 'Y'=10, 'Z'=11, and 'W'=00, the 4-letters plaintext string may be written as a binary string.

This “no duplication” 4-letters plaintext string is now encrypted using a key in the form of an p X q matrix, comprised of pq squares adjacent to each other. Each of the pq squares is marked by one of the same four letters: ‘X’, ‘Y’,’Z’,’W’.   The markings may be arbitrary as long as the map satisfies the *universal access* condition, defined ahead. Starting with any arbitrary square on the pq matrix, one could define “allowed moves”, which are up, down, right, left, marked as: U,D,R,L. A sequence of allowed moves, defines a track or a pathway. The universal access condition says that from every square on the matrix, taken as the starting point for a pathway, there is at least one pathway that ends up with either one of the three letters (apart from the letter of the starting point), such that this pathway traverses only through squares marked by the same letter as the starting square.

The universal access condition provides assurance that whatever the identity of the 4-letters no-duplication plaintext list, it can be used as a guide to mark a pathway on the matrix. The guide works as follows: By agreement we mark the start of a plaintext string with a ‘W’, since it is clear that it is added as a start mark since ‘W’ is inserted only to break duplication, and hence it could not be the first square on the list unless it is added there as a mark for the starting point for the plaintext list. One will agree on some square on the matrix which is also marked by the letter ‘W’. Now, if the plaintext string reads, say WXY… then the universal access assures us that the matrix will have a pathway that may lead either directly from the starting square (marked ‘W’) to a square marked ‘X’, or via a series of squares marked ‘W’ which will eventually end up with a square marked ‘X’.   Whichever square marked ‘X’ that the path WW…X leads to, there will be a path from there to a square marked ‘Y’ (Of the form: XXXX…Y), and so on. Regardless of how long the plaintext list it can be regarded as a guide to mark a pathway on the matrix.   This pathway may be specified by a list comprised of identified moves: up, down, right, left (U,D,R,L). One could map: ‘U=01, ‘D’=10, ‘R’=11, ‘L’=00, and express the pathway as a binary string.

The fundamental idea of the Samid cipher is that the pathway may be regarded as a ciphertext that can readily be decrypted to the said plaintext. To decrypt the pathway string (the ciphertext), one starts at the same starting square as the encryption started with (a square marked

‘W’), and then one lists the letters that mark the squares over which the pathway passes. Because of the way the cipher string was constructed, the list of traversed, or visited squares will features quite a few duplications. For example it will look like: WWWXXYYYYYXXWWWZZ… Albeit, we recall that the plaintext string was duplication free, so we may collapse the duplication-features list of visited squares to its non-duplication equivalenet, by simply collapsing any repeat appearance of same letter into a single letter. So that the above example will look like: WXYXWZ.

We have described above how an arbitrary plaintext string may be encrypted to its ciphertext string, and decrypted back to its original plaintext – based on the identity of the matrix, which is hence regarded as the symmetric cipher key, or say cipher-map.

It is clear that any size matrix, however large or however small that satisfies the universal-access condition is a proper key for the Samid cipher, so that the Samid cipher works with variable size key. (One needs an agreed upon mapping of a key-string to the map, the matrix). In fact the smallest size key, or map, is a matrix comprised of 3x3 squares, e.g.:

X	X	Y
Z	W	Y
Z	Z	y

Which may be used to encrypt any size plaintext. Also the Samid cipher builds a ciphertext which is longer than the plaintext string. With this attributes in mind we can specify the Samid cipher as a proper cipher for the crypto-square algorithm.

2.2 Mapping a Key-String to a Samid Matrix and Vice Versa

In the crypto-square algorithm the cipher key is described as an arbitrary bit string of variable length. If one wishes to use the Samid cipher for the crypto-square application, then it is necessary to biject the Samid matrix with a corresponding bit string. To do that we further limit the Samid map as follows:

We describe here how to use the cryptographic key string comprised of *k* bits to construct a bona-fide Samid cipher cryptographic key. The procedure amounts to consecutively using the bits in the cryptographic key to guide the marking of squares in the rectangular or matrix expression of a Samid cryptographic key. To distinguish between the string and the Samid key, we will call the former the key-string, and the latter the Samid-key, or the Samid-map.

We shall use Samid cipher with the four letter alphabet X,Y,Z,W. The key is comprised of "square rings" around a starting square, all drawn on a straight Euclidean plane. The starting square (marked W) is ring-0. It is surrounded by 8 squares: 4 squares with a shared edge to the starting square, and 4 squares with only a shared vertex with the starting square. This is square-ring-1. Ring 1 is surrounded by ring 2 comprised of 16 squares, and so on: ring-*n* is comprised of *8n* squares, 4 squares of them are corner squares. All the even rings will be marked with the letter W. The odd rings will be each marked with X, Y and Z as follows. So this particular variety of the Samid map is comprised of layers or rings, which alternate between rings totally marked with W and rings marked with a combination of ‘X’, ‘Y’, and ‘Z’. So ring 0 has 1 square marked W, ring 2 has 8*2=16 squares all marked W, ring 4 has 8*4=32 squares, all marked W, etc. And ring-1 has 8 squares marked with a combination of ‘X’, ‘Y’, and ‘Z’, ring 3 has 24 squares marked with a combination of ‘X’, ‘Y’, and ‘Z’. (See Fig-2). We shall agree that the outermost ring in a Samid cipher of this variety will always be an even ring, namely a ring all marked with W.

We shall describe now how to use the bits in the cryptographic key string as a guide to fill in the current square-ring of the Samid map.

For odd ring #n, there are 8n squares to be filled-in, or say marked with either X, or Y, or Z. We determine that the four corner squares will be marked as the square which is next to them, going clockwise. So we are left to determine the marking of $u = 8n-4$ squares. We shall agree that the rings will be filled in or marked by a Fill-In-Procedure (FIP) that will assign the markings of the squares of the ring one by one, going counterclockwise, and starting with the square that defines the intersection between the rightmost column, and the middle row. Later on we shall introduce greater flexibility in marking the squares to reverse the direction, to change the starting square over which to use the FIP to mark the squares, and to use the other 5 orders: X,Z,Y; Y,X,Z; Y,Z,X; Z,XY; Z,Y,X.

We shall set: $w = u\text{-}2$, and search for the lowest value of k_1, a natural number such that:

$$2^{k_1} \geq w$$

We shall then evaluate the numeric value of the first k_1 bits from the key string, to read as the number T, and then compute:

$$x_1 = (T \bmod w) + 1$$

We shall then fill up the first x_1 squares in the ring with 'X' marks. If $x_1 = w$, then the ring has 2 squares left to be assigned. The first will be assigned with a 'Y' and the second will be assigned with a 'Z'. This will conclude the task of marking the ring with X-Y-Z symbols.

If $x_1 < w$ we shall set: $w = u - x_1 - 1$, and similarly, find the lowest value of k_2, such that $2^{k_2} \geq w$. We shall evaluate the numeric values of the substring of the key string beginning with bit k_1+1 and ending with bit k_1+k_2. Let T' be its value, from T' we shall determine:

$$y_1 = (T' \bmod w) + 1$$

and will then mark the next y_1 squares in the ring with 'Y'. If $y_1 = w$, then the ring has only one square left, and it would be marked with 'Z'.

For the other cases, where $y_1 < w$, we shall set $w = u - x_1 - y_1$, and find the lowest value of k_3, such that $2^{k_3} \geq w$. We shall then examine and evaluate the next k3 bits in the key string. Let them be evaluated to T, and then we compute:

$$z_1 = (T \bmod w) + 1$$

We shall then fill up the ring with z_1 squares marked 'Z'.

The ring has now $u' = u - x_1 - y_1 - z_1$ squares left to be filled. If $u' < 3$ then it is filled with 'Z' marks. If $u'=3$ then the first square is marked with 'X', the second with 'Y' and the third with 'Z'. If $u' > 3$ then it is treated as an empty string of squares. In other words, we treat the u' ordered squares the way we treated the u squares above. And we treat the same the remaining substring from this second round -- same as the u' substring of the first round. We can continue with this procedure, each time leaving a small substring to fill in, until the remaining substring, the new u' is comprised of three squares or less, and then it is filled in as above. The fill-in procedure (FIP) is applied iteratively then. If we fill in odd ring n, then we must determine the X,Y,Z marking for its 8n squares. The corner squares are excluded, as indicated above, so we are left with u=8n-4 squares to be filled in. Applying FIP over u (reading the bits from the key string), we then remain with u' unmarked squares (u-u` squares have been duly marked by the FIP procedure). We then re-apply FIP over the u` squares, and end up with u" un-filled squares (duly filling up, or say marking u`-u" squares). Subsequently, we apply FIP over the u" unmarked squares until we finish marking all the squares in the ring.

It is straightforward to ascertain that the Samid cipher so constructed is in compliance with the inherent attributes of such a key, mainly that from every reference square marked by any letter, there is a path that leads from it to any and each of the other three letters, and that path is comprised of squares marked by the same letter as in the reference square. (The universal access condition).

Insufficient key bits: A problem arises when there is not sufficient 'key material' to determine the marking of the ring. In that case the ring will be marked off the Samid map, and the odd ring before it will be the outmost odd ring.

The very construction of the Samid map guarantees (i) that the map will be consistent with the Samid map requirements, and (ii) that every proper allocation of squares to the letters 'X', 'Y', and 'Z' can be covered with a proper bit sequence of the Samid key string.

The smallest odd ring has 8 squares. Four of these squares are corner squares which are marked according to the preceding square – going counterclockwise. Hence, there are 4 squares left for assignment. We have k_1= 1, since $2^1 \geq 4-2$. There will be either two squares marked X or one (corner squares not counted), and so the remaining squares, if they are three will be marked X,Y,Z, (according to the rules above), and if they are two will be marked Y,Z.(according to the rules above). And hence a single bit key size is enough to construct a valid Samid-map that is ripe and ready to encrypt any size plaintext.

To further illustrate: a Samid key string comprised of the bit "0" will translate to x_1= (0 mod 2) + 1 = 1. Ring 1 has 8 squares, 4 of them are corner squares, so 4 are left to be assigned by FIP. The key bit "0" assigned one square (marked 'X'), so 3 squares are left to be marked, and according to the rules above they are marked: X, Y, Z. Ring 1 will now look like:

	X	
Y		X
	Z	

Filling in the corner-squares, per the set rules:

X	X	X
Y		X
Y	Z	Z

Now adding ring-0 and ring-2 (the outmost ring must be even), the constructed Samid map looks like:

W	W	W	W	W
W	X	X	X	W
W	Y	W	X	W
W	Y	Z	Z	W
W	W	W	W	W

And if the single bit of the Samid key is "1" then we have x_1= (1 mod 2) + 1 = 2. There remain 2 squares to be assigned, and according to the above rules they are assigned 'Y' -- the first and 'Z' the second. Accordingly, the Samid map will look like... the same as above. We conclude therefore that the first ring of any Samid map in this Crypto-Square version always looks the same, and is always determined by the first bit. We can also say: $k_{threshold}$= 1.

Let's see what happens for Ring-3. Ring-1, we have seen is comprised of 8*1-4=4 squares. Ring-2 is the 'W' ring, comprised of 8*2=16 squares. And the next ring up, Ring 3 is comprised of 8*3 =24 squares. 4 of those squares are corner squares which receive their letter marking from the preceding square, (counterclockwise), so we are left with 20 squares to be letter marked. Lets say the Samid key string looks like:

Key String = X = "10010010001011001110"

The first bit, "1" is used to mark the first ring (ring-1) as we have seen above. The next bits are used as follows: We have $w = u-2 = 20-2 = 18$. We are looking for k_1 such that:

$$2^{k_1} \geq 18 \text{ and } 2^{k_1-1} < 18$$

k_1 evaluates to 5. The 5 next bits from the key string are: "00100", which is written in decimal as: 4, so $x_1 = (4 \bmod 18) + 1 = 5$. Meaning that the first 5 squares on ring-3 will all be marked with 'X'. There are 20-5 = 15 squares left to be marked. We look for the smallest value to satisfy: $k_2 \geq \log(15-1)$, and conclude that k_2=4. The next 4 bits from the key string are: 1000, or 16 in decimal, and hence: $y_2 = (16 \bmod 14) + 1 = 3$, and so we mark the 3 next squares with 'Y'. We are left with $20\text{-}5\text{-}3=12$ squares to mark. We set $w=12$, and look for k_3, the smallest natural number such that: $2^{k_3} \geq 12$, and the answer is $k_3=4$. So we pick the next 4 bits from the key string: 1011; the decimal value is: 11, and hence: $z_1 = (11 \bmod 12) + 1 = 12$. This then assigns letters marks to all the squares in ring-3. The left over key substring is: 001110, which is too short to mark the next ring, ring-5 which is comprised of: 8*5-4 = 36 squares to assign from the key bits, so these extra bits are ignored, they don't participate in building the map. It's noteworthy that the number of 'leftover' bits depends on the number and the identities of the bits beforehand. By design the last X-Y-Z ring is surrounded by a W-ring (even ring).

The Samid map key will look like:

```
W W W W W W W W W
W Y Y Y Y X X X W
W Z W W W W W X W
W Z W X X X W X W
W Z W Y W X W X W
```

```
W Z W Y Z Z W Z W
W Z W W W W W Z W
W Z Z Z Z Z Z Z W
W W W W W W W W W
```

This particular definition of key has one noted restriction: The order of the X-Y-Z letters is always the same, and moving from the center (ring-0) to the right always reads: WXWXWX... This can be avoided in several ways, one way is as follows:

The first three bits of the key will indicate the applied order for filling the key, for example:

```
001 - XYZ
010 - XZY
100 - YXZ
011 - YZX
110 - ZXY
101 - ZYX
```

and the values "000" and "111" reserved, or used to indicate start and end of the key string respectively.

The next bits of the key will be used to indicate the starting point of the marking process. In the above detailed example the starting point was always the same, the rightmost middle square, or square #1. One could replace that with starting the very same procedure as above, but shifted from square #1 to any square #S, as follows:

Let $t = 8n\text{-}4$ be the number of squares to be filled from the key string in the particular odd ring *#n*. We shall look for the smallest natural number s* where $2^{s^*} \geq t$, and then extract the next s* from the key string, and evaluate its value, s**: We shall compute the shift :

*S = s** mod t*

and apply it. So instead of starting the letter marking at square #1 (the leftmost middle square), we shall start at square #S.

This interpretation of 3 bits for the order of X-Y-Z, and S bits for the shift can be repeated before marking any successive ring in the Samid map. Note that the decision whether to add this variance on starting square or order of the markings should be based on the application for which the crypto-square is being used for. For many instances, the simple one where the order is always X-Y-Z and the starting square is the same, will be sufficient.

For illustration: let the key string start with: "0110101" We take the first three bits '011' to indicate the order YZX per the table above. The number of squares to be filled in is 8 minus the 4 corner squares: 4. So s*=2 in this case, and the next two bits in the key string are "01", and so S = (1 mod 4) = 1. We therefore shift the starting square by one, and use the indicated order to mark the following map:

W	W	W	W	W
W	Y	Y	X	W
W	Y	W	X	W
W	Y	Z	Z	W
W	W	W	W	W

2.3 Low End Limitation

The cryto-square or the crypto-addition, as described, requires an input string of some minimum size so that it can be broken down to key, input-string, and selection string. We accommodate this limitation as follows:

We also wish to allow for a low threshold for the size of the number to be added. Since the number is the source for selecting between encryption and decryption, and for determining the key and the input string, it stands to reason that such determination may require that the number to be added, or 'squared' will be of a minimum size, let's call it $x_{threshold}$. We can then adjust the definition of addition to state that for a number X where the bit string representation is x bits long, then:

$$\forall \quad |X| = x \leq x_{threshold}: 2X|_c = X^2|_c = X$$

Namely a number smaller than $x_{\text{threshold}}$ adds to itself and squares to itself. This amounts to a 'singularity collapse' which will reduce the efficacy of the squaring operations for certain applications. For most applications such collapse may be handled by squaring off a related number. So if $X^2|_c$ ends up with collapse, it is likely that $(X+\alpha)^2|_c$ will not collapse, for some arbitrary α. Otherwise, a modular solution might help, as discussed ahead.

The Samid cipher will work with a threshold size for key and for the "*pc*" string: $k_{threshold}$, $pc_{threshold}$. These variables, in turn, will determine the size of $x_{threshold}$:

$$\boldsymbol{x_{threshold} = k_{threshold} + pc_{threshold} + 2 + 0}$$

The "+0" indicates that in the extreme case where $k=k_{threshold}$, and $pc=pc_{threshold}$, the number of bits, t, needed to indicate the separation between the *pc* string and the *k* string is $t=0$. For $x > x_{threshold}$, the value of t will be determined as follows:

For an input natural number X, where |X|=x, we will first strip the leftmost bit and the rightmost bit (that are XORed to determine whether the next step is encryption or decryption), then we separate t leftmost bits, where the value of t is the smallest natural number that satisfies:

$$2^t \geq x - 2 - k_{threshold} - pc_{threshold}$$

and where T is the numeric value of the so determined leftmost t bits, and hence:

$$T' = 1 + T \bmod (x-2-t-k_{threshold} - pc_{threshold})$$

We now divide X' by first removing the t leftmost bits that determine the cut between the "pc" string and the key string, then we divide the remaining $(x-2-t)$ bits as follows: The substring from bit t+1 on X' to bit $t+1+pc_{threshold}+T'$ will be the "pc" string, and the remaining substring from bit: $t+1+pc_{threshold}+T'+1$ to the last one on X' will be the key string. (See Fig-1).

Illustration: let: $k_{\text{threshold}}$= 10 bits, and $pc_{\text{threshold}}$= 18 bits, while |X|=x = 40 bits:

$$\mathbf{X = 1001100100011000111011011011100010110100}$$

We compute:

$$\mathbf{x_{threshold} = k_{threshold} + pc_{threshold} + 2 + 0 = 10 + 18 + 1 + 0 = 29}$$

So x >$x_{threshold}$.

We first chop off the rightmost and leftmost bits (used to determine that the next step is encryption because 1(XOR) 0 = 1.), and write X' :

$$\mathbf{X' = 00110010001100011101101101110001011010}$$

We now compute *t*:

$$\mathbf{2^t \geq x - 2 - k_{threshold} - pc_{threshold} = 40\text{-}2\text{-}10\text{-}18 = 10}$$

which evaluates to t=4. The 4 leftmost bits are "0011" and so T=3, and:

$$\mathbf{T' = T \bmod (x\text{-}2\text{-}t\text{-}k_{threshold} - pc_{threshold}) = 3 \bmod (40\text{-}2\text{-}4\text{-}10\text{-}18) = 3 \bmod 6 = 3}$$

And hence we divide X' by first removing the 4 leftmost bits, (the *t* bits) and then dividing the remaining 34 bits by allocating $pc_{threshold} + T' = 18 + 3 = 21$ to "pc" and the rest, 13 bits for the key. As we can see the "pc" string and the key string are both longer than the respective threshold figures. In conclusion, we evaluated the given X string to 21 bits to serve as a plaintext to be encrypted using a Samid cipher with a key k, comprised of 13 bits derived from the same string, X.

The Samid cipher works with a flexible size key. So we now need to establish the procedure how to translate the key string to the Samid key (the Samid map). The translation should also allow for an easy "work-back", meaning for a straight forward algorithm that will solve the square root problem.

2.4 The Special Features of the Samid Cipher:

Crypto-Square does not require the use of the Samid cipher, but that cipher packs the required attributes: *(i)* $g = |C| - |P| = f(K,P,C)$*:* the size gap between the ciphertext and the plaintext may be wide ranging and it depends on the identity of the key and the input string (plaintext or ciphertext); (ii) the cipher works with any size key above $k_{threshold}$, without any upper value limitation; (iii) the key-construction to effect decryption may tailor the opposite encryption to encrypt back to the exact string that was key-constructed for decryption.

The last point may need elaboration. The Samid cipher is a one-to-many, many-to-one cipher, where a plaintext P may encrypt into many ciphers $C_1,C_2,....C_n$, such that all of them decrypt back into the same plaintext. To prove that a string could have been a result of self-adding many possible strings, the backward designer takes a particular C_i and constructs a Samid key to decrypt it into its matching plaintext, P, such that when the same key is used, it will encrypt to exactly C_i. Alas, it is not necessary that the same key when applied in the reverse, namely to encrypt P, will yield exactly C_i, and not some C_j where $i \neq j$. Yet, the nature of the Samid key is that it allows for the key construction to effect a proper plaintext that would encrypt to exactly the ciphertext that generated it beforehand. This correspondence between the plaintext and ciphertext is essential to the equivocation claim of the crypto-cipher.

The Samid cipher by its construction and nature, provides degrees of freedom for the encryption process to choose a variety of pathways to express the same plaintext, and since every pathway is a distinct ciphertext, which decrypts back to the same plaintext, these degrees of freedom allow for the variance, and indetermination of the exact ciphertext, given the plaintext and the key.

If these degrees of freedom are left intact then it would harm the crypto square process because when one tries to solve the square root equation, and find a plaintext that would encrypt to the given ciphertext, then it is not enough to build a Samid map, or say a Samid key such that the ciphertext will indeed decrypt to a corresponding plaintext, it is necessary to insure that the key is such that when the said plaintext is fed in for encryption it will generate the very same ciphertext that created it beforehand, and not any other valid ciphertext that decrypts to it.

One can insure that the plaintext will encrypt to the desired ciphertext by setting up rules to govern these degrees of freedom, and then constructing the key accordingly. We have built a robust correspondence between the Samid key (map) structure, and a corresponding bit string, so, once the proper map is constructed, it can be expressed with a proper bit string.

Here are some of the rules:

- The Samid Map is built as successive "square rings", ("rings").

The inner "ring" is just a single square, marked W (ring-0). It is surrounded by 8 squares that comprise ring-1. It, in turn is surrounded by ring-2, comprised of 16 squares, and in general ring-n is comprised of *8n* squares (for n>0).

- All even rings: 0,2,4,6,... are all marked with 'W'.
- A Samid key outer ring is even, namely the outer ring is always occupied with W.
- All odd rings:1,3,5,... are marked with 'X, 'Y' and 'Z' such that each subset of consecutive squares marked by same letter, is terminated by one of the two other letters on one end, and the other (the third) on the other end.

Such that XXXXYYYYYXXXX is a non qualifying substring because the substring comprised of YYYYY is terminated by X marked square on both sides.

In the following we shall refer to the ‘traveler’—this is the essential feature of the Samid cipher: the entity that uses the plaintext as travel guide, and decides which steps to take on the Samid map.

- **Infinity**: if a traveler occupies a square in the rightmost column, and the travel path calls for another right step, then the traveler jumps to the leftmost square in the same row.

- **Infinity (2):** if a traveler occupies a square in the leftmost column, and the travel path calls for another left step, then the traveler jumps to the rightmost square in the same row.

- **Infinity (3):** if a traveler occupies a square in the upper row, and the travel path calls for another upward step, then the traveler jumps to the lowest square in the same column.

- **Infinity (4):** if a traveler occupies a square in the lower row, and the travel path calls for another downward step, then the traveler jumps to the highest square in the same column.

- **W direction (1):** If a W mark is called for, from an X, Y, or Z square then the outer ring of W will be chosen, not the inner one, until the highest ring is reached.

- **W direction (2):** Once the outer ring has been reached the rule for the next 'W' switches from opting to the outer ring to opting for the inner ring. When ring-0 is reached, the opting switches again, from inner to outer, and so on, the direction of the choice 'W' ring oscillates from inner to outer, indefinitely.

- **XYZ direction (1):** If an X, Y, or Z mark is called for, from a W square then the outer ring of X-Y-Z will be chosen, not the inner one, until the highest X-Y-Z ring is reached.

- **XYZ direction (2):** Once the outer XYZ ring has been reached the rule for the next 'XYZ' switches from opting to the outer ring to opting for the inner ring. When ring-1 is reached, the opting switches again, from inner to outer, and so on, the direction of the choice 'X-Y-Z' ring oscillates from inner to outer, indefinitely

- **Choice within an 'X'-'Y'-'Z' ring (1):** Let G be the letter that marks the current square in the Samid cipher process pathway on an odd ring (G may be X, Y, or Z as the case may be). Let H be the next letter indicated by the plaintext (H ≠G), and L be the third letter (L ≠ H, L≠G). Rules:

  - **Choice-1**. If the current square faces two squares marked H, then the one consistent with counterclockwise direction will be chosen.
  - **Choice-2**. If the current square faces one square marked H, that one will be moved into (chosen).
  - **Choice-3**. If the current square faces two G letters, the one consistent with counterclockwise direction will be chosen, unless that letter G was chosen before while the trip was conducted over G marked squares only since that previous choice. In the latter case, the opposite G will be selected.

  So for a case like:

```
X X Y Y Y Y Y Y Z   🡐 letter marks on an odd ring

1 2 3 4 5 6 7 8 9 🡐 numeric identification numbers
```

If square marked 5 is the current one (marked ‘Y’), and the plaintext calls for ‘Z’ as the next letter in the pathway, then, because both squares 6 and 4 are marked ‘Y’, the choice, according to the above rules is square 4 because it is counterclockwise (assuming the string shown is the upper side of the ring). This is obviously the bad choice because it moves the pathway towards square marked ‘X’, not ‘Z’, but the rules assumes only visibility of touching neighbors, so the choice-maker at square 5 has no visibility towards where the marking will change to ‘Z’, and where to ‘X’. According to the same rule the state square will move from square 5 to square 4, then to square 3. From square 3 there is no choice except to retreat to square 4 (square 2 is the wrong letter). And then according to the rules above (choice-3) the state square will move from 4 to 5, 6, 7, 8 and finally to square 9 where it abides by the dictates of the plaintext and moves from ‘Y’ to ‘Z’.

By setting up the above rules, one eliminates the degrees of freedom given by a choice of plaintext and key. The rules narrow down the choice of ciphertext to the one intended, among the many possible.

The advent of the Samid map is that one can determine the letter markings of each square where the path is stepping through, as well as the letter markings of neighboring squares, to channel the respective ciphertext to the desired pathway. So if a ciphertext is given by an arbitrary bit string, then it may be interpreted as a sequence of up, down, right, left, and mark any path of choice. Given this path it is a straightforward task to build a Samid map 'under that path' and mark the traversed squares, as well as the neighboring squares so that a respective plaintext will be marked and listed, and the neighboring squares will be marked so as to insure that the respective encryption will chart the same path as the original string that led to the back-designed decryption.

For example: if the ciphertext has a section ‘D,R,R,D (down, right, right, down), this might be effected through a Samid map portion that looks like this:

```
  W
Z X X X Z
      W
```

with a corresponding plaintext: WXW, such that on its way back (from plaintext to ciphertext) the two ‘Z’ on the key would force the corresponding pathway to be D,R,R,D.

2.5 The W-String Enhancement

The issue here pertains to design-back difficulty present in the above design. It identifies it, and fixes it.

Our design back is a mechanism to show that given any natural number Y, it is possible to compute an infinity of numbers $X_1, X_2, \ldots X_n$ such each will square-off to Y. And to do that we need to show that given any natural number, Y, one could find an infinity of numbers $X'_1, X'_2, \ldots X'_n$ such that each of them will self-add to Y*: $2X'_i|_c = Y$, for i=1,2,...* We wanted to show that such addition may involve either encryption, or decryption, because in the addition process the self-added number determines whether the process ahead involves encryption or decryption.

We clearly realize that there is no problem in design-back for decryption. We simply take any Samid map that we happen to choose, use it to encrypt Y, and then we denote the ciphertext as X", We then express the used Samid map as a key string K, concatenate K with X" (K-X"), add the appropriate t bits to allow one to properly separate K-X" to K and X", add "1" as a rightmost bit and "1" as a left most bit (or alternatively add "0" for both leftmost and rightmost bit), and so we construct X', since as designed, we clearly can write:

$$\mathbf{2('1\text{-}t\text{-}X''\text{-}K\text{-}1')|_c = Y}$$

We know for sure that X" will decrypt to Y using the Samid map constructed from K, because the same key was used to encrypt it from Y. The Samid cipher is a one-to-many for encryption, and a many-to-one for decryption, so we know for certain that the decryption will yield Y, and no other.

In principle, this would be sufficient because the above shows that we could generate an infinity of solutions to the square-root equations, even though all the generated cases terminate with a decryption, not with an encryption. We claim infinity of solutions because we could assign infinity of Samid maps, of any size.

There is a different situation when we try to design-back for addition that involves encryption. In that case we must find a string X' that would encrypt to Y. We can readily build a

Samid key that would decrypt Y to some X' (in fact any Samid map will do that). The problem arises on the reverse. How to insure that when X' is encrypted it yields Y, and not some Y' that too will decrypt to X'.

In general there are Y, Y_1, Y_2,.....Y_n ciphertexts into which X' may encrypt, and for our design-back to work we must insure that the encryption process will yield Y and not any of the other possible ciphertexts. (We may also designate the original Y as $Y=Y_0$

As discussed above, we may try to reduce the degrees of freedom innate in the encryption process by setting forth rules that would determine, how to make choices in case of ambiguity, or degrees of freedom in building the pathway (the ciphertext) from the plaintext (the 'travel guide'). By taking these rules into account it is possible to mark the squares in the Samid map such that the generated plaintext will encrypt to the desired ciphertext. Such is indeed possible in the vast majority of the cases. And if a case is encountered where no solution is found to force the generated plaintext to encrypt to the specific ciphertext, then one could try again with another Samid map.

While the above strategy will work, it may be more elegant to guarantee that for every possible Samid map, it would be possible to force the plaintext to generate the desired ciphertext.

It turns out that this is possible using the "W-string" enhancement. It works as follows:

It is easy to see that any pathway that is limited to a given odd ring may be programmed such that the corresponding plaintext will necessarily encrypt to the generating ciphertext. If necessary, this can be done by marking the squares of that odd ring as a sequence of X-Y-Z:

....X-Y-Z-X-Y-Z-X-Y-Z-....

Such basic granularity will be used, if necessary, to represent any given pathway within this ring, in a way that its encryption will lead to the generating pathway. For example, consider a 'crazy' pathway in the form (R – right, L – left):

R,L,R,L,R,R,R,L,R,L,L....

Suppose that this sequence starts at the middle Y in the above odd ring. The pathway will be faithfully described by the plaintext:

Z-Y-Z-Y-Z-X-Y-X-Y-X-Z...

and that plaintext when it is encrypted will generate the exact pathway which generated it.

It is a different problem when it comes to even rings populated by W only. When a sequence of right and left is described by the pathway (the ciphertext) over an even ring, then the corresponding plaintext collapses to a single W. And it is a challenge upon encrypting this plaintext how to generate the exact over-W pathway that generated it. Another problem arises when the pathway enters an even ring from an odd ring. Whether the pathway moves to the outer W (even) ring, or to the more internal W (even) ring, the plaintext is expressed the same: W. So upon encryption how to choose the same direction (outer or inner) as was indicated by the exact pathway that generated that plaintext. One may note that the shift from an odd ring to an even ring poses no problem because the W-square where the shift to an odd ring occurs may be marked with two different letters on the outer and inner rings options.

These two problems are to be solved, as follows:

We need to insure that a pathway comprised of some back and forth steps over an even (W) ring will be re-constructed when its corresponding plaintext is encrypted. To accomplish this one will generate a "W-guide bit string", or "W-string" for short. The string will be read from right to left, two bits at a time. When the pathway crosses to an even ring, the procedure calls for reading the 2 rightmost bits in the W-string, and interpret them as follows:

- "01" means move to the next W square clockwise.
- "10" means move to the next W square counterclockwise
- "11" means move to the outer ring upon first chance (when the current state-square becomes adjacent to the next letter in the plaintext), while moving counterclockwise along the even ('W' ring).

- "00" means move to the inner ring upon first chance (when the current state-square becomes adjacent to the next letter in the plaintext), moving clockwise along the even ('W') ring.

To illustrate: Let the W-string be:

11010110100101

Let a plaintext section read as 'XWY'. Let a slice of even ring (W) and an odd ring (X-Y-Z) be as follows:

```
X  Y  Y  Z  Z  Z  Z  X  X  Y    <= ring #11
W  W  W  W  W  W  W  W  W  W   <= ring #10
X  X  X  Y  Z  Z  X  Y  Y  Z  <= ring #9
W  W  W  W  W  W  W  W  W  W   <= ring #8
0  1  2  3  4  5  6  7  8  9    <= position markings
```

Now suppose that the X indication on the above plaintext slice corresponds to square number 6 on ring #9. Since the next letter in the plaintext specifies 'W', the encrypter will have to decide whether to select a 'W' on ring #10 or a 'W' on ring #8. So the encrypter consults the W-string. The two left most bits on that string are "01". That means that string at this point does not indicate which W-string to choose. In the absence of such guidance the standard rule will apply: the outer will be selected until the outermost ring is encountered and then the inner ring will be selected. Accordingly the encrypter determines the path as 'U' (UP) to square #6 on ring #10. The next plaintext letter is 'Y'. But the current W-square has 2 W neighbors (in the same ring), one X neighbor (the former state square on ring #9), and one Z, in ring #11. So clearly the path must move one of the possible W squares. The default choice is counterclockwise to Square #5 on ring #10. But the encrypter is bound to consult the W-string for possible preemption of the standard rule. The encrypter will consult the 2 rightmost bits on that string, which have already been examined and found to be: '01', which according to the rules means to move to the next 'W' square clockwise. Namely to square #7 on the same even ring. The next two bits are also '01' which instruct the encrypter to move to square #8 on same ring, number 10. The next two pairs of bits indicate two counterclockwise moves, so the path moves from square #8 to square #6. And then to square #7, and to square #8 again. And what is next? the last pair of bits says '11', which indicates move from now on counterclockwise, and shift to the outer ring when possible.

So despite the fact that the current square (#8 on ring #10) has a 'Y' adjacent to it (on ring #9), it will not move there because the instruction embedded in the '11', and instead it will move counterclockwise on its W-ring, and keep doing so until it reaches square #2 on its ring (#10). So instead of making a simple move down to 'Y' on ring #9, the encrypter chose to move 6 times counterclockwise, and then go up to 'Y'.

So according to the W-string, the pathway corresponds to the XWY section of the plaintext is:

Ciphertext (1): U-R-R-L-L-R-R-L-L-L-L-L-L-U

Note that the same plaintext could have been encrypted into Ciphertext(2): 'U-L-D'. And the way the Samid cipher works both ciphertexts (Ciphertexts, (1) and (2)) would decrypt to the same plaintext. This illustrates how the W-string guides the encrypter to chart a particular cipher. This language is used by one who wishes to decrypt a given string such that it would encrypt to exactly the same given string.

What is left to determine is where to find the W-string. The answer is that the W-string concatenates to the "pc" string, or in our case, to the "p" (=”pc”) string, the plaintext. P-W. As the plaintext encrypts it reads its bit from the left, while it looks for W guidance from the right. And that is why the W string is listed from left to right. Eventually the two processes meet. The plaintext, P string's last bit is the one such that the bit right to it was most recently read as W-guidance.

We can now describe how the back-design happens. A bit string marked as Y is to be reverse-added (subtracted) to find string X such that $2X|_c = Y$. It is desired that this particular addition will involve encryption. To do that the back-designer would select an arbitrary Samid map, K*, and use it to decrypt Y into X*. We know that when X* is encrypted it will follow its standard rules and produce some $Y^* \neq Y$ as its ciphertext. So the back-designer wishes to compel the encryption process that encrypts X* to produce exactly Y, and not a different ciphertext. We have seen above that any peculiarity in the pathway of Y expressed over an X-Y-Z ring can be readily accounted for by changing the square marking in the initial Samid map, K*, turning it into a modified Samid map: K*'. And just above we have seen how one could design a W-string that would compel the encryption process to follow the pathway of Y as it is expressed over W (even) rings. So the back designer will concatenate the so designed W string, W, next to P. Now when the encryption of P will take place and as the generated ciphertext pathway is about to move into W territory, it will start looking to the W string for guidance. And per its design the P string will finish at the point where the W string offered its last piece of guidance, and the result will be the exact path Y that was the starting point of the back designer.

The back-designer will set: $X = 2^{-2}Y|_c$= '1'-t-P-W-K*'-'0', and the value of the t bits will be designed to separate 'P-W' from K*'. By its construction, as above, the addition of X to itself will yield Y. Note: the term ‘back designer’ refers to the designer that builds up a string X that would self-add to the given string Y.

3.0 Properties

The main property of the crypto square is that its domain and co-domain are both the infinite series of natural numbers. The crypto square is a proper function in as much as any natural number has a well defined square, but it is not a bijection in as much as any crypto-square has infinite natural numbers that square to it. *So the full range of natural numbers is the image of the crypto square function.*

By allowing the domain to remain open ended, the full range of natural numbers, one guarantees an infinite number of solutions to the square-root equation, and thereby an infinity rated entropy measure.

Since the infinity solution theorem is proven by construction, it leads to the derived theorem that for any $Y \epsilon$ to Z^+ there are infinite values $X_1,X_2,....X_n$ that solve the equation: $\sqrt{Y} = X^2|_c$.

On the other hand, since the squaring of any number, X, has Z^+ as its co-domain, then the probability that for any given a-priori $A \epsilon\ Z^+$ we will have: $A = X^2|_c$ is infinitesimally low, and hence for any *finite* set of natural numbers, the chance that any member of the set will square to a given A is also negligible.

We address now two questions of interest:

Given $Y_A = (X+A)^2|_c$ and $Y_B = (X+B)^2|_c$, where $A \neq B \in Z^+$, what is the X entropy? Both Y_A and Y_B range over Z^+. So brute force examination will never be conclusive. The procedure itself seems to defy any analytic expression of the squaring algorithm, but if one is found then it would be possible to determine the measure of entropy, or equivocation. There may be none, one, few, or infinite solutions to the pair of equations as above. Of course if a given X, A, and B were used then there is at least one solution, but it is not clear how many more.

Proof of the Infinity Theorem: Consider the equation: $X = \sqrt{Y}|_c$. We will solve it by first 'subtracting': finding a natural number, Q, such that: $2Q|_c = Y$, or $Q = 2^{-2}Y|_c$. To do so we first make an arbitrary choice of whether $2Q|_c$ will be an encryption or a decryption. Suppose we decide it would be an encryption. Hence the $2^{-2}Y|_c$ process will be a decryption. We will then construct an arbitrary valid Samid map subscribing to the rings rules (even rings full of 'W', odd rings marked with 'X', 'Y', and 'Z') [We use the single quote to differentiate between the X, Y Z, W used in defining a Samid map, and the use of these characters elsewhere]. We now interpret Y as a bit string that lists by some order, map pathway directions: up, down, right, left. (U,D,R,L). The pathway, however long, fits into the size of the Samid map, whatever its size, and that is because of the infinity rules that dictate how to handle a pathway that seems to exceed the limits of the map. Having marked the pathway on the map, we can now use our choice letter markings on the squares of the map and deduce the corresponding plaintext, P. We have no restriction on how large the key, or the map may be. And once the size is determined we have a large variety (although not infinite) for letter-marking. The way the Samid cipher works is that not necessarily all the key structure is used for either encryption or decryption. And hence, even for back-designed decryption as we discuss now, and even for a limited size Y string to decrypt, it is possible to design an infinitely large key. Clearly so when we wish to construct a key for back-work encryption (where the crypto self-addition will be a decryption). The Samid cipher puts no restriction on how much larger the ciphertext is than the plaintext, so it can be of any desired size, using any desired large key. It is this attribute of the Samid cipher that insures that the back-work, or the solution of the crypto square root equation will have an infinity of solutions.

As we discussed, if the back-work is encryption, then there is no doubt that the addition comprised of decryption will shrink into the plaintext Y. We do have an issue in the reverse because if the back-work operation was decryption then the addition that it reverses will be encryption, and encryption in the Samid design is one-to-many. We have shown with the 'W' string that we can add a string next to the plaintext P such that it would guide the encryption process to choose exactly the string that generated it, namely Y. We have also shown how to concatenate the W-string to the plaintext, so it can properly guide the encryption process. We have thus a proof by construction that the crypto square root equation has an infinite number of solutions.

Constructing Q: We first concatenate the just produced plaintext, P, with the W bit string that was designed to guide the crypto self addition process to encrypt exactly Y and no other. Of course, if the self-addition is selected to decryption, then there is no W string. We mark this concatenation as 'P-W'. We now further concatenate with K, which is the key-string that corresponds to the Samid map that we engineered to decrypt Y to P. This creates a concatenated string P-W-K, where |P|=p is the bit count of the decrypted result, P, w=|W| is the bit count of the W string, and |K|=k is the string expression of the Samid map.

We now solve the following: identifying the number of bits *t* to be used to indicate the breakup of P-W-K to P-W and K. We are looking for the smallest *t* such that:

$$2^t \geq p + w + k - pc_{threshold} - k_{threshold}$$

and identify the bit string T for which the numeric value will be:

$$T = p + w - pc_{threshold}$$

Now we construct a bit string of length *t* bits, *T**, which evaluates to T (hence we might have to add a necessary number of zeros to the left of the number).

We finally contcatenate T* to the left of P-W-K to build T*-P-W-K. So done we now need to add 2 bits, one rightmost and the other leftmost, so that their XORed value will indicate encryption, or decryption, as the next step. The string:

Q = {1 bit}-T*-P-W-K-{1-bit}

Note: We mark the W-string for both the case where the self addition is encryption or decryption. Clearly if it is decryption the W-string vanishes, or say its length w=0.

We can now assert: $Q = 2^{-2}Y$. This is so because we have traced back the crypto addition, to insure that the result achieved, Q, will be crypto self-added to Y. We can see it clearly: to compute $2W|_c$, one will first XOR the rightmost and leftmost bits, to identify the process as encryption or decryption. Since we added these two bits, their message will fit our design. These two bits are then ignored. One will then look for the value of *t* bits to indicate the breakup of the rest of the string to P-W and K. Based on the construction of this substring, one will identify the same number *t* that we used to add the T* substring. Evaluating the value of this substring, T, one will divide the P-W-K substring exactly the way it was originally concatenated to 'P-W' and K. The next step in the crypto self-addition process would be to construct the Samid map, which one will do using in reverse the same rules that were used to build this string from the map used in the back-work. Therefore the resultant map will be the one that was used in processing of the reverse-addition equation ($Y \rightarrow Q$). If the map is the one that was used in the decryption process before, then if the rules of resolving encryption choices are applied (per the W string) then the resultant ciphertext will be exactly Y, which proves that the computed W adds up to Y.

Taking note that the back-work key, and the reverse-generated plaintext were arbitrarily selected, and that there are infinitely many choices for keys and plaintexts, one concludes that there are infinite solutions to the reverse-addition (subtraction) equation. And since the Crypto Square is essentially a finite series of additions, it also follows that the Crypto Square has infinite solutions.

It is worthwhile to remember that the key can be constructed to affect a plaintext much much smaller than the ciphertext. Especially in higher up rings. Note that ring number *n*, will be comprised of *8n* squares, and in each ring one could find long sequences of same letter. Such sequences might correspond to a plaintext much smaller than the ciphertext.

We have presented very specific procedures that can take any arbitrary natural number, Y, and back-work from it an infinite number of natural numbers X_1, X_2,... that will all Crypto-Square into Y, given the definition of crypto squaring. This cardinal fact serves to protect the identity of the particular natural number X that was used to crypto-square it to Y. This protection by equivocation is immunized against some future or unknown mathematical insight, and this very fact keeps our interest focused on this concept.

4.0 Use

Being an equivocation protected one-way function the crypto square is in a position to assume quite a few roles in important crypto protocols. The crypto-concept may be tailor specified to a wide variety of uses. One could design it for a predominance of decryption cycles, and use it as an alternative hash function, or alternatively tailor design it for encryption predominance, and satisfy expansion purposes. There would be uses derived from the fact that there is no predictable relationship between the size of the input and the size of the output. In particular if the input to the crypto-square has an unknown size, then a cryptanalyst has no final set from which to brute-force search for the input. Another use may be based on the option to tailor specify the crypto-square to a very long computation by defining a ‘stop rule’ ‘for the crypto additions that would requires a large number of cycles. One could ‘contaminate’ an input set with numbers that require such long computations, so the brute force cryptanalyst will “get lost” chasing their crypto result.

One main use of the Crypto Square involves confirmation of mutual knowledge. Suppose Alice and Bob wish to confirm that they both hold on to the same shared secret, M. Yet, other than this suspected secret they have no shared secret data. So Alice cannot use a secret key to encrypt M, for Bob to verify that it's the same M he is holding to. Alice, in general, could use any acceptable one-way function like raising a given A by M mod some number N. Bob will do the same and thereby they will confirm their shared secret. So doing keeps Alice and Bob vulnerable to Eve who might have found a mathematical shortcut to exercise the modular logarithmic computation, and derive M. Eve might use a generic computational formula, or a specific formula good for some special cases which Alice's choice happens to be one of them. This is especially important if Eve is a world class crypto organization and if the secret Alice and Bob share has a long life span.

In that case, Alice and Bob can use the crypto square as an unbreakable one-way function. As discussed in the properties section, if they agree on the same crypto result they should be quite confident that there is little chance for collision, namely that while they both agree on the same $M^2|_c$, their M values are different. And if Eve will have to list an infinite number of candidates to deduce M from, she will be confused by equivocation.

Another use involves identity verification. Alice wishes to ascertain that the one who claims to be Bob is indeed Bob. She has Bob's secret identifier, b. She would then develop some random number A and challenge Bob to square $(b+A)^2|_c$. If he computes right, he knows **b** – no doubt. In the next session Alice will challenge Bob with A' to compute: $(b+A')^2|_c$. She will then check again. Eve will have a hard time to make a list of all values that fit both squares in order to find, hopefully the one value of b that would fit both computations. Alice could apply the same trick with a new random number some k times, and although the larger the value of k, the greater the chance that there will be only one value to satisfy all the squares, because for each square there are infinite number of options, Eve will find it intractable to spot this one value. Of course Bob can ascertain Alice's identity in a symmetric fashion.

Key Management: The Crypto-Square could be used to facilitate key management in several ways: (1) identity based encryption, (2) key hierarchy.

Identity Based Encryption: Alice and Bob communicated with a center, each with his or her own key: K_a and K_b. Originally Alice and Bob give the center some personal id data, P_a, and P_b, and the center, relying on a 'deep secret' S, computes: $K_a = (S+P_a)^2|_c$, $K_b=(S+P_b)^2|_c$, and safely communicates the keys to their owners. The center does not need to maintain a database of these keys, so they cannot be stolen. Instead, when Alice or Bob communicate with the center they first identify themselves with P_a and P_b, so that the center can regenerate K_a and K_b, as before, based on S.

Key Hierarchy: A deeps secret S may generate a first echelon of high level keys:
$K_{11} = (S + P_1)^2|_c$, $K_{12} = (S + P_2)^2|_c$, ….

To be attributed to departments 1 and 2 identified by their private data P_1 and P_2 respectively. Each department could allocate crypto keys to its sections using the private identification of these sections. So department 1 will build a key for its section (a):

$$K_{11a} = (K_{11} + P_{1a})^2|_c$$

without knowing deep secret S. And so on, the sections will allocate keys down to subsections, and they to individuals, etc. Each echelon will build it down using the Crypto Square, without visibility to the generating key of the higher echelon. It's noteworthy that the keys are backward protected. Namely if a derived key is compromised, the key that generated it in a squaring process remains shielded by the equivocation principle.

Communicating Squared Results: Since the squared results may be of surprise size, it may be necessary for certain protocols to hash the results to a fixed size. This can be done by compromising some of the advantages of raw squaring, and so optimization may be in order.

5.0 Summary

The crypto-Square is a function that maps an arbitrary natural number X to a definite natural number Y, which may be larger, much larger, smaller, or much smaller than X, and which may be the mapping result of infinite count of other natural numbers, and as such, given Y it is impossible to ascertain the particular number X which squared into it. This renders the crypto square into a robust one-way function, to be used whenever a generic one-way function is used. We presented an implementation of the Crypto-Square concept, using a cipher where the ciphertext may be larger than the plaintext, and we have shown specifically how the Samid cipher may be applied for the purpose. The presentation here opens up the possibility to implement Crypto-Square anywhere a generic one-way-function is called for.

References:

- Agarwal-12: Siddharth Agarwal*, Abhinav Rungta*, R.Padmavathy*, Mayank Shankar* and Nipun Rajan, "An Improved Fast and Secure Hash Algorithm" March 2012
- Diffie-76 W. Diffie and M. Helman, New directions in cryptography, IEEE Trans. Inform. Theory, Vol. 22, 1976, pp. 644–654.
- Goldwasser-84: S. Goldwasser and S. Micali, Probabilistic Encryption, JCSS, Vol 28, No. 2, April 1984, pp. 270–299.
- Grigoryv-2011: Cryptography Without One-Way Functions, Dima Grigoriev And Vladimir Shpilrain CNRS, Math´ematiques, Universit´e de Lille, 59655, Villeneuve d'Ascq, France 2011
- Grollmann-98: J. Grollmann and A. Selman. Complexity measures for public-key cryptosystems. SIAM Journal of Computing, 17:309, 1988.
- Hemaspaandra-99 L. Hemaspaandra and J. Rothe. Creating strong, total, commutative, associative one-way functions from any one-way function in complexity theory. Journal of Computer and System Sciences, 58(3):648–659, 1999.
- Impagliazzo-89 R. Impagliazzo and M. Luby. One-way functions are essential for complexity based cryptography. In Proceedings of FOCS89 – Symposium on Foundations of Computer Science, 1989.
- Impagliazzo-90: R. Impagliazzo and L. A. Levin. No better ways to generate hard np instances than picking uniformly at random. In Proceedings of FOCS90 – Symposium on Foundations of Computer Science, 1990.
- Merkle-89: R.C. Merkle. One way hash functions and des. In G. Brassard, editor, Advances in volume 435 of Lecture Notes in Computer Science, pages 428–446. Springer, 1989.
- Naor-95: Moni Naor, Moti Yung "Universal One□Way Hash Functions and their Cryptographic Applications" Proc□ of the ACM Symposium on Theory of Computing, March 1995
- Odegard-12: Rune Steinsmo Ødegård, "Hash Functions and Gröbner Bases Cryptanalysis" Thesis for the degree of Philosophiae Doctor Trondheim, April 2012
- Ostrovsky-81: Ostrovsky One-way functions, hard on average problems and Statistical Zero Knowledge proofs. IEEE Conference on Structure in Complexity Theory, 1991
- Preneel-12: Bart Preneel "THE HASH FUNCTION CRISIS AND ITS SOLUTION" Hakin8 Extra, 2012
- Rivest-78: R. Rivest, A. Shamir and L. Adleman, A method for obtaining digital signatures and public-key cryptosystems, Comm. of the ACM, Vol. 21, 1978, pp. 120–126.
- Samid-01: Samid, G. 2001 "Re-Dividing Complexity Between Algorithms and Keys (Key Scripts)" The Second International Conference on Cryptology in India, Indian Institute of Technology, Madras, Chennai, India. December 2001.
- Samid-02: Samid, G. 2002 " At-Will Intractability Up to Plaintext Equivocation Achieved via a Cryptographic Key Made As Small, or As Large As Desired - Without Computational Penalty " 2002 International Workshop on CRYPTOLOGY AND NETWORK SECURITY San Francisco, California, USA September 26 -- 28, 2002
- Samid-03: Samid, G. 2003 "Non-Zero Entropy Ciphertexts (Stochastic Decryption): On The Possibility of One-Time-Pad Class Security With Shorter Keys" 2003 International Workshop on CRYPTOLOGY AND NETWORK SECURITY (CANS03) Miami, Florida, USA September 24 -- 26, 2003
- Samid-03B: Samid, G. 2003 "e-Identity: An Unsolved Problem" International Conference on Computer, Communication and Control Technologies: CCCT '03 July 31, August 1-2, 2003 - Orlando, Florida, USA

- Samid-04: Samid, G. US Patent #6,823,068
- Samid-07: Samid, G. 2007 "Proposing a Master One-Way Function" eprint, 3 Oct 2007, The International Association for Cryptologic Research (IACR)
- Samid-12: Samid, G. US Patent #8,229,859
- Shai-97: A. Sahai and S. P. Vadhan. A complete promise problem for Statistical Zero Knowledge. Proceedings of FOCS97 – Symposium on Foundations of Computer Science, 1997.
- Shannon-49: Shannon Claude. Communication Theory of Secrecy systems. In Bell Systems Technical J., 28:656–715, 1949. N

Table of Contents: