Adrian Fang

2023 March 4

Hydra Cryptographic Cipher (HC-1)

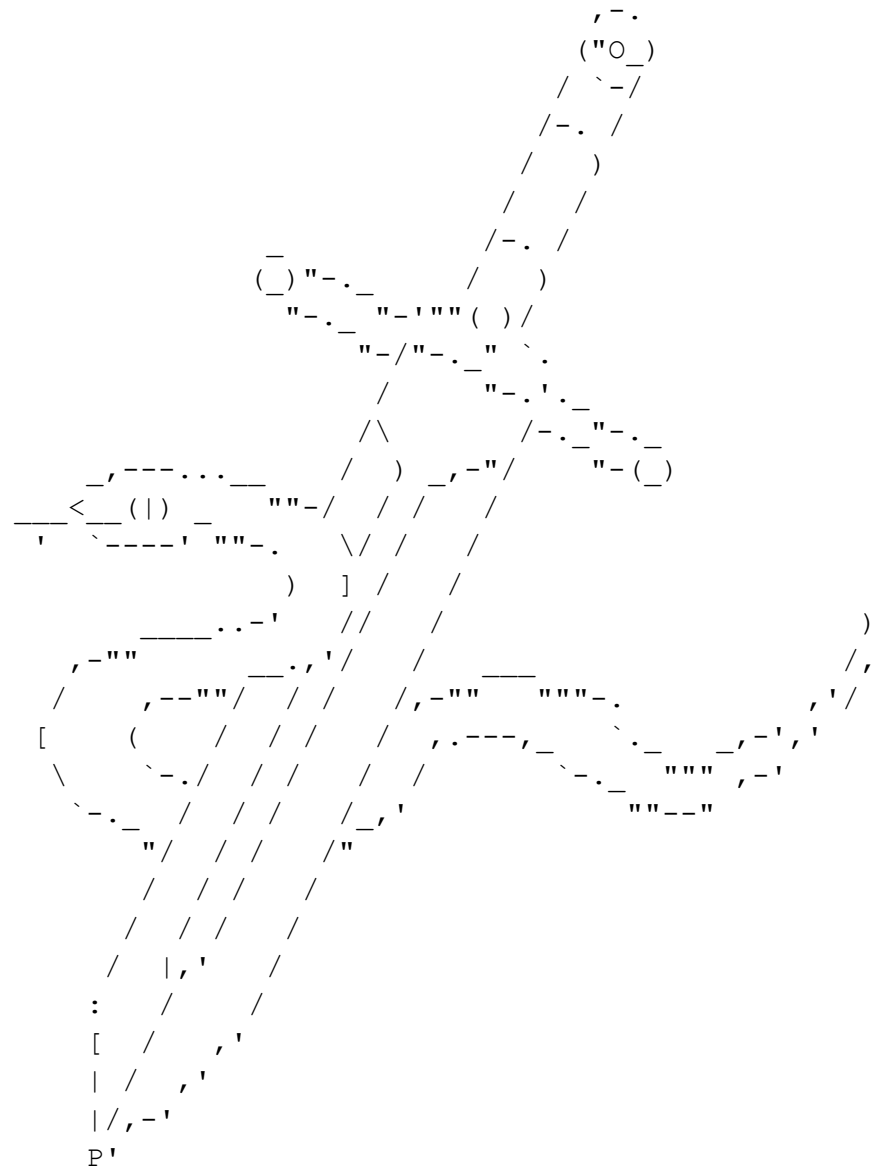

Table of Contents

1. Introduction

The Hydra Cryptographic Cipher (HC) is an algorithm aimed to implement and utilise methods and theory from other cryptographic functions with additional complexity. Similarly to other standards, HC uses mathematical operations to compute a plaintext into a cipher, where said cipher follows the inverse operation of encryption to subsequently have the cipher return to the plaintext. With the two procedures, data can be encrypted and decrypted, sent through on a network without the risk of compromising information from packet interception. HC uses varied theory from a block cipher, though using arbitrary though reversible operations for computation, producing an array of integer values for a cipher. This algorithm is to be used with the Hydra Botnet Framework.

2. Definitions

2.1 Glossary of Terms, Acronyms and Symbols

Below are numerous definitions of certain terms and acronyms for the standard. For all functions, refer to Sec. 2.2 to determine their usages:

HC, Hydra Crypt (Short for Hydra Cryptographic Cipher)

ln, The natural logarithm function equivalent to log_e

O, Big O notation, worst case time complexity

mod, The modulo operation, integer division and returning the remainder

x_l, length of set x

x_i, $x[i]$, $x_{[i]}$, ith index of set/list x (such that x_0 is the first element)

x_t, temporary variable for x_i

y_n, the count or length of y

k_{bl}, the bit length of k

o, the variable representing output for the encryption process

n, the base-10 value of the plaintext

$\oplus$, bitwise XOR operation symbol

$+$, addition symbol

$\times$, multiplication symbol

$-$, subtraction symbol

$\int$, Integral symbol

$>$, greater than symbol

$=$, equals to symbol

$\sum$, sigma, denoting a sum

$\prod$, multiplicative series symbol

$\in$, element is a part of set symbol

$\notin$, element is not a port of set symbol

$\subset$, is a subset of a set symbol

$\cup$, union between sets or appending elements to an array

2.2 Algorithm Parameters, Symbols and Functions

There are several constants used in HC, where said constants are below:

k, an arbitrary constant used as a base

r, an additional arbitrary constant, used in exponentiation

q, an arbitrary constant used with constant r

t, u, both RSA primes used for an extra layer of encryption

b_{ls}, the constant used to determine the length of the block cipher

The following constants have constraints, where the constraints are listed below:

$let\ A = \{k, r, q, t, u, b_{ls}\}$

$A \subset \mathbb{Z}^{+}$

$b_{ls} > 1$

With the constants defined, additional values are calculated based on said constants, defined as such:

$n = ut$, Product of primes u, t

$e = 65537$, RSA public exponent, not to be confused with Euler's Number

$\varphi = (u - 1)(t - 1)$, Euler's totient function, used to obtain the private exponent for RSA decryption

$d = e\ mod^{-1}\ \varphi$, the private RSA decryption exponent

Considering these constants, the following functions can be defined:

$r_e(m) = m^e\ mod\ n$

$r_d(c) = c^d\ mod\ n$

$h(x) = x + (r + q)$

$h^{-1}(x) = x - (r + q)$

$f(x) = h(x + k^{(q+r)\ mod\ 16}) \oplus r_e(|q - r|\ mod\ 2^{(tu)_{bl}-1})$

$f^{-1}(x, n) = h^{-1}(x \oplus r_e(|q - r|\ mod\ 2^{(tu)_{bl}-1})) - k^{(q+r)\ mod\ 16})$

(n will not affect calculations, as it is a bit length specifier)

$p(c) = 12c^2 + 20c + 15$

$P(c) = \int p(c)dx = 4c^3 + 10c^2 + 15c + C$

$t(c) = 5c^4 + 16c^3 + 9c^2 + 2c + 7$

$T(c) = \int t(c)dx = c^5 + 4c^4 + 3c^5 + c^2 + 7c + C$

$Q(c, i) = \int_{c-i}^{i} [p(x) + t(x)]dx$

$w(x) = \frac{3x+1}{x^2-3x+2}$

$W(x) = \int w(x)dx = 7ln(|x - 2|) - 4ln(|x - 1|) + C$

$v(x) = 9x^2 + x - 5$

$$V(x) = \int v(x)dx = 3x^3 + \frac{x^2}{2} - 5x + C$$

$$s(i) = [\sum_{n=1}^{i} (20n^2 + 50n + 100)]^{\frac{b_{ls}}{2}} = (\frac{20i^3+105i^2+385i}{3})^{\frac{b_{ls}}{2}}$$

$$X(i) = [\int_{(i \bmod k)-(i-k)}^{i+k} (w(x) + v(x))dx]^{(i \bmod 1000)+1} + q^2 + r^2 + t^2 + s(i)$$

There are other functions such as the array shift, encryption and decryption functions, however do not utilise the constants nor affect individual values in one operation. The following functions below do not follow a singular calculation (refer to Sec. 5 for further specification):

$C_e(m)$, the Encryption function

$C_d(m, c_n, r, l_n)$, the Decryption function

$S(a, x)$, the array shift function by amount by x

$S^{-1}(a, x)$, the inverse array shift function by amount x

$P_d(a, i)$, the padding function

The standard utilises every function to be effectively used. For all functions in this standard, the range is $\{y \mid y > 0\}$ (likewise with the domains, in which case arguments must be positive), where the output cannot be below 0.

3. Notation and Conventions

3.1 Inputs and Outputs

Inputs will involve a base-10 integer converted from a base-16 hexadecimal string stemming from an encoded string. Any output when converting the plaintext into the cipher results in a block, where the length of the block is dependent of constant b_{ls}. Bit lengths of the values in the blocks are dependent on the given constants, as they will change according to said values. There are several constraints for inputs, as they would be unpermitted by this standard. Let x be the value plaintext, where constraints would be:

$$x \in \{n \mid n > 0, n \in \mathbb{Z}^+\}$$

Block size can be any value, however the message must be large enough when converted into an integer to be able to contain the large portion, otherwise the algorithm will be erroneous.

3.2 Bytes

The basic unit of processing for HC is the byte, a sequence of 8 bits compiled into one singular entity. The first step in the encryption process comprises the division of bytes of the original integer, where it is converted into base-2. Said bytes are determined by b_{ls}, where the length of the bits per block is calculated through $\frac{b_l}{b_{ls}}$, where in this case b_l is the bit length of the argument. When decrypted, all bytes can be reconstructed to reform the original message.

4. Mathematical Preliminaries

4.1 Addition

Addition is an elementary concept in mathematics, and is an important element of encryption. Mathematical addition involves taking two or more separate values and putting said values together to obtain the sum of the numbers. The following example demonstrates the principle of addition:

$$let\ A = \{1\}$$
$$let\ B = \{2\}$$
$$let\ C = A \cup B = \{1, 2\}$$
$$let\ f(x) = \sum_{i=0}^{x_l - 1} x_i$$

In this case, for argument x for function f, the values are described as:

x, the set of numbers

x_l, the length of set x

x_i, the ith index of set x

As set C has been defined as the union of set A and B, function f is used to find the sum of the values in the set.

$$f(C) = \sum_{i=0}^{1} C_i$$
$$C_0 = 1, C_1 = 2$$
$$C_l = 2$$
$$f(C) = 3$$

This example is the same as the operation of $1 + 2$, which will return the value of 3, therefore demonstrating addition.

4.2 Multiplication

Multiplication is a simplistic operation, used in the various functions in HC, involving adding a number by itself by the second value in an operation. The following example demonstrates this behaviour, where two is the first value being multiplied by four, with it being equivalent to the sum of two four times, equating to 8.

$$2 \times 4 = \sum_{x=1}^{4} 2 = 2 + 2 + 2 + 2 = 8$$

4.3 Integration

In Calculus, the operation of an integral acts as the inverse of finding the derivative of a function. For definite integration, it is the process of finding the area that a function covers over a specified range of two numbers under the curve it would produce in a graph. There are numerous rules when finding a function's antiderivative. Some notable rules are specified below.

$\int [f(x) + g(x)]dx = \int f(x)dx + \int g(x)dx$, the Integral sum property

$\int af(x)dx = a\int f(x)dx$, the Integral constant property

$\int \frac{1}{x}dx = ln(|x|) + C$, the Integral rational function property

$\int x^n dx = \frac{x^{n+1}}{n+1} + C,\ n \neq -1$, the Integral power property

$\int_a^b f(x)dx = F(b) - F(a),\ F(x) = \int f(x)dx$, for definite integrals

$\int_a^b f(x)dx = \lim_{n\to\infty} \sum_{i=1}^{n} f(x_i)\Delta x$, the definition of a definite integral (Riemann Sum)

Let the following example demonstrate the obtaining the value of definite integral.

$w(x) = \frac{3x+1}{x^2-3x+2}$

$\int_5^{10} w(x)dx = v$

Function w is given, where variable v equates the definite integral of w between intervals 5 and 10. Integrating w to find function W then using the definite integral rule is used to then solve for variable v.

$W(x) = \int w(x)dx$

$W(x) = \int \frac{3x+1}{x^2-3x+2}dx$

$W(x) = \int \frac{7}{x-2}dx + \int \frac{4}{x-1}dx$

$W(x) = 7\int \frac{1}{x-2}dx + 4\int \frac{1}{x-1}dx$

$W(x) = 7ln(|x-2|) + 4ln(|x-1|) + C$

$v = W(10) - W(5)$

$v = [7ln(|10-2|) + 4ln(|10-1|) + C] - [7ln(|5-2|) + 4ln(|5-1|) + C]$

$v \approx 23.3$

In this case, the value of v is approximately 23.3.

4.4 Series

The series of a function is essentially the sum of itself for a specified amount given a certain range. Though simplistic in theory, the methodology may vary as it must be optimised algorithmically to decrease the complexity. The following example demonstrates the optimisation of a summation for computation.

$g(x) = 20n^2 + 50n + 100$

$f(x) = \sum_{n=1}^{x} g(n)$

$f(10) = g(1) + g(2) + g(3) + \ldots g(10) = 11450$

$f(x) = \sum_{n=1}^{x} 20n^2 + \sum_{n=1}^{x} 50n + \sum_{n=1}^{x} 100$

$f(x) = 20\sum_{n=1}^{x} n^2 + 50\sum_{n=1}^{x} n + 100\sum_{n=1}^{x} 1$

$$f(x) = 20\frac{x(2x+1)(x+1)}{6} + 50\frac{x(x+1)}{2} + 100x$$

$$f(x) = \frac{20x^3+30x^2+10x}{3} + 25x^2 + 125x$$

$$f(x) = \frac{20x^3+30x^2+10x}{3} + \frac{75x^2+375x}{3}$$

$$f(x) = \frac{20x^3+105x^2+385x}{3}$$

$$f(10) = \frac{20(10)^3+105(10)^2+385(10)}{3} = 11450$$

As the function's runtime is now $O(1)$ instead of $O(n)$, it can be implemented into HC without affecting the domain and range.

4.5 Exponentiation with Integer Exponents

Exponentiation is the process of multiplying a number a specified amount of times by itself. The direct behaviour can be described as such:

$$x^n = \prod_{i=1}^{n} x$$

Let the following example demonstrate the practicality of the rule specified:

$$let\ x = 2$$

$$\phi = x^3$$

$$\phi = \prod_{i=1}^{3} x$$

$$\phi = \prod_{i=1}^{3} 2 = 2 \times 2 \times 2 = 8$$

4.6 Bitwise XOR Operation

Bitwise operations are used in the encryption process, to affect numeric values on a binary level. There are four separate Bitwise operations which are the shift, OR, AND and XOR (exclusive OR) operations. The standard uses the XOR operation, which is similar to the OR operation but with differing theories. For all bitwise operations, operations are done with the base-2 variations of the original base-10 integer values. With two separate values, the bitwise operations are done comparing the bits of the numbers with the other number (in which the indices are compared), where either a 0 or a 1 is returned for each bit (denoting True or False). For the OR operation, the two values are compared, where if at least one of the corresponding indices have a 1, the index of the result will also have a 1, where if they don't, a 0 is returned. The XOR operation is similar, however a 1 is returned only if one of the indices has a 1, where the other two cases result in a 0. Let the following example demonstrate the XOR operation:

$$let\ n_1 = 25$$

$$let\ n_2 = 12$$

$b_1 = 11001$ (Base-2 Equivalent of n_1)

$b_2 = 01100$ (Base-2 Equivalent of n_2)

$p = b_1 \oplus b_2 = 10101$

$n_p = 21$

In this case, for values b_1 and b_2, executing the XOR operation results in 10101. The 1's in the value are from there being a case where the corresponding indices of the values yield 0 and 1, which are different, subsequently returning 1. All other cases yield a value of 0. With the XOR operation, the relevance is simply the applicability to encryption it has, in which this operation is reversible as opposed to OR and AND. Let the following demonstrate the inverse behaviour of the XOR operation:

$k = p \oplus b_2$

$k = 01100$

$n_k = 12$

$n_k = n_1$

As demonstrated above, the bitwise XOR operation has applications to encryption, where it has been implemented in this standard.

4.7 Modulo Operation

The modulo operation is fundamentally similar to floor (integer) division, however instead of returning the quotient, the modulo operation returns the remainder. Let the following example demonstrate the modulo operation:

$let\ q = 3\ mod\ 11 = 3$

In this case, 3 is modded by 11 in which as 3 times 3 is 9, there is a remainder of 2 as 3 cannot divide into 2, therefore the solution is 3. The modulo operation has periodic behaviour, giving it useful applications for algorithms and other purposes without the overly complicated concepts of other periodic functions such as sin, cos or tan.

5. Algorithm Specification

For any uncertainty with functions, refer to Sec. 2.2 for their definitions. Given that b_{ls} is constant, runtime for C_e and C_d is $O(1)$. If assuming otherwise, runtime is $O(n)$.

5.1 Cipher Encryption

HC encryption follows the following process to convert the plaintext into the cipher, assuming the plaintext has been converted into an integer:

$let\ n = plaintext$

$n = n + 10^{\frac{b_{ls}}{2}}$

The second step exists to account for the restricted domain of the functions in Sec. 2, where the addition negates the possibility of having an error. The following steps consist of converting n into an array of bytes, where it must first be converted into base 2:

$let\ m = \{\}$

$$c_n = \frac{b_l}{b_{ls}}$$

For a loop lasting b_{ls} times, values $n[: c_n]$ will be added to m, where $n[: c_n]$ will be removed from n to consistently have different sets of bits. The algorithm is not perfect in the sense that all bits will be inserted into m, as there is a remainder set of bits at the end which remain unused. To avoid said values from not being used, they are cast aside as a remainder value, where it will follow the same procedures as the values in m. Afterwards the set of instructions will follow:

$let\ r_d = n$ (Remainder of n)

$let\ l_n = \ length\ of\ r\ in\ bits$

$m = S(m, b_{ls} + 1)$

$for\ each\ i \in [0, m_l[,\ i \in \mathbb{Z}^+\ do$

$m_t = s(i + 1)$

$m_i = f(m_i)$

$m_i = m_i + m_t$

$m_i = m_i \oplus Q(m_i, i + 1)$

$end\ for$

The same instructions will execute for remainder:

$r_t = s(b_{ls} + 1)$

$r_d = f(r_d)$

$r_d = r_d + r_t$

$r_d = r_d \oplus Q(r, b_{ls} + 1)$

At the end of encryption, values $m,\ c_n,\ r,\ l_n$ are returned. If the user intends to add an extra security measure to their cipher, the following instruction can be executed to pad the function further:

$let\ o = \{m,\ c_n,\ r,\ l_n\}$

$o = P_d(o, x)$ (where x is an arbitrary positive integer)

Once completed, the cipher is almost unbreakable unless all constants and returned values are given.

5.2 Cipher Decryption

The process of decryption is the exact same as encryption, however following the same procedure backwards (therefore $C_d = C_e^{-1}$). Had the cipher been padded, the first operation below would occur:

$let\ o = \{m,\ c_n,\ r,\ l_n\}$

$o = P_d(o, x)$

Otherwise, decryption requires values $m,\ c_n,\ r,\ l_n$ from the encryption process to properly compute into the plaintext. Assuming the constants are correct, the plaintext should be obtainable:

$$r_t = s(b_{ls} + 1)$$
$$r_d = r \oplus Q(r_d, b_{ls} + 1)$$
$$r_d = r_d - r_t$$
$$r_d = f^{-1}(r_d, l_n)$$
$$for\ each\ i \in [0, m_l[,\ i \in \mathbb{Z}^+\ do$$
$$\quad m_t = s(i + 1)$$
$$\quad m_i = m_i \oplus Q(m_i, i + 1)$$
$$\quad m_i = m_i - m_t$$
$$\quad m_i = f^{-1}(m_i, c_n)$$
$$end\ for$$
$$m = S^{-1}(m, b_{ls} + 1)$$

All values in m will be concatenated to each other, with r being appended after iteration. The value of n is a base-2 string, and must be converted back into base-10, where afterwards the final operation occurs, completing decryption:

$$n = n - 10^{\frac{b_{ls}}{2}}$$

The user must convert the base-10 integer back into a string, to fully restore the original message.

5.3 Padding Function

If the user desires to use HC to encrypt messages without taking the bit length of the constants into account, the alternative would be to encrypt each individual character of the plaintext and return an array of cipher arrays. A problem arises from this method, considering that it would make the array vulnerable to a substitution attack, where the message can be decrypted through means of inference. To patch such a vulnerability, the padding function is used as an extra security measure to prevent ciphers from being decrypted without the constants. The parameters of the padding function P_d are simply the cipher array c_{ls} and constant i, to pad the cipher:

$$i_c = X(i + 2)$$
$$for\ each\ j\ \in [0, c_l[,\ j \in \mathbb{Z}^+\ do$$
$$\quad c_{ls[0][j]} = c_{ls[0][j]} \oplus i_c$$
$$end\ for$$
$$c_{ls[2]} = c_{ls[2]} \oplus i_c$$

Once padding has been completed, the entire cipher array has been XOR encrypted, where the key must be correct to decrypt the cipher array. In the proper application, the XOR encryption value (constant i) will be determined based on the index of iteration when encrypting the array.

5.4 Cipher Encryption Pseudocode

The following is the procedure of the encryption algorithm without specification. Refer to Sec. 5.1 for details.

$let\ n = plaintext$ (converted into long)

$n = n + 10^{\frac{b_{ls}}{2}}$ (then converted into binary)

$let\ m = \{\}$

$c_n = \frac{b_l}{b_{ls}}$

$for\ each\ j\ \in [0, b_{ls}[,\ j \in \mathbb{Z}^+\ do$

$m\ = m \cup n[: c_n]$

$n = n[c_n:]$

$end\ for$

$let\ r_d = n$

$let\ l_n = length\ of\ r\ in\ bits$

$A = m_{[0:(b_{ls}+1)]}$

$B = m_{[(b_{ls}+1):]}$

$m = B \cup A$

$for\ each\ i \in [0, m_l[,\ i \in \mathbb{Z}^+\ do$

$m_t = [\sum_{n=1}^{i+1} (20n^2 + 50n + 100)]^{\frac{b_{ls}}{2}}$

$m_i = [(m_i + k^{(q+r)\ mod\ 16}) + (r + q)] \oplus [(|q - r|\ mod\ 2^{(tu)_{bl}-1})^e mod\ tu]$

$m_i = m_i + m_t$

$m_i = m_i \oplus \int_{m_i-(i+1)}^{m_i} (5x^4 + 16x^3 + 21x^2 + 22x + 22)dx$

$end\ for$

$r_t = [\sum_{n=1}^{b_{ls}+1} (20n^2 + 50n + 100)]^{\frac{b_{ls}}{2}}$

$r_d = [(r_d + k^{(q+r)\ mod\ 16}) + (r + q)] \oplus [(|q - r|\ mod\ 2^{(tu)_{bl}-1})^e mod\ tu]$

$r_d = r_d + r_t$

$r_d = r_d \oplus \int_{r-(b_{ls}+1)}^{r} (5x^4 + 16x^3 + 21x^2 + 22x + 22)dx$

$let\ p_d = Arbitrary\ value\ for\ padding\ function$

$$let\ o = \{m,\ c_n,\ r,\ l_n\}$$

$$i_c = [\int_{((p_d+2)\ mod\ k)-(p_d+2-k)}^{p_d+k+2} [\frac{3x+1}{x^2-3x+2} + (9x^2 + x - 5)]dx]^{((p_d+2)\ mod\ 1000)+1} + q^2 + r^2 + t^2 + [\sum_{n=1}^{i+2} (20n^2 + 50n + 100)]^{\frac{b_{ls}}{2}}$$

$$for\ each\ j\ \in [0, o_l[,\ j \in \mathbb{Z}^+\ do$$

$$o_{[0][j]} = o_{[0][j]} \oplus i_c$$

$$end\ for$$

$$o_{[2]} = o_{[2]} \oplus i_c$$

5.5 Cipher Decryption Pseudocode

The following is the procedure of the decryption algorithm without specification. Refer to Sec. 5.2 for details.

$$let\ p_d = Arbitrary\ value\ for\ padding\ function$$

$$let\ o = \{m,\ c_n,\ r,\ l_n\}$$

$$i_c = ([\int_{((p_d+2)\ mod\ k)-(p_d+2-k)}^{p_d+k+2} [\frac{3x+1}{x^2-3x+2} + (9x^2 + x - 5)]dx]^{((p_d+2)\ mod\ 1000)+1} + q^2 + r^2 + t^2 + [\sum_{n=1}^{i+2} (20n^2 + 50n + 100)]^{\frac{b_{ls}}{2}})$$

$$for\ each\ j\ \in [0, o_l[,\ j \in \mathbb{Z}^+\ do$$

$$o_{[0][j]} = o_{[0][j]} \oplus i_c$$

$$end\ for$$

$$o_2 = o_2 \oplus i_c$$

$$r_t = [\sum_{n=1}^{b_{ls}+1} (20n^2 + 50n + 100)]^{\frac{b_{ls}}{2}}$$

$$r_d = r_d \oplus \int_{r-(b_{ls}+1)}^{r} (5x^4 + 16x^3 + 21x^2 + 22x + 22)dx$$

$$r_d = r_d - r_t$$

$$r_d = ((r_d \oplus ((|q - r|\ mod\ 2^{(tu)_{bl}-1})^e mod\ tu)) - (r + q)) - k^{(q+r)\ mod\ 16})$$

$$for\ each\ i \in [0, m_l[,\ i \in \mathbb{Z}^+\ do$$

$$m_t = [\sum_{n=1}^{i+1} (20n^2 + 50n + 100)]^{\frac{b_{ls}}{2}}$$

$$m_i = m_i \oplus \int_{m_i-(i+1)}^{m_i} (5x^4 + 16x^3 + 21x^2 + 22x + 22)dx$$

$$m_i = m_i - m_t$$

$$m_i = ((m_i \oplus (((|q - r| \bmod 2^{(tu)_{bl}-1})^e \bmod tu)) - (r + q)) - k^{(q+r) \bmod 16})$$

$end\ for$

$A = m_{[-(b_{ls}+1):]}$

$B = m_{[:-(b_{ls}+1)]}$

$m = A \cup B$

$for\ each\ j \in [0, b_{ls}[,\ j \in \mathbb{Z}^+\ do$

$n = n \cup m[:c_n]$

$m = m[c_n:]$

$end\ for$

$n = n - 10^{\frac{b_{ls}}{2}}$

5.6 Array Shift

The encryption and decryption algorithm require an array shift, where the specified array a will be shifted to the left x times. To avoid shifting by an unnecessary amount, x is modded by b_{ls}, where shift amounts will therefore not exceed the block size. The function has $O(1)$ complexity, given the simplicity of the function. The function has the splitting of the given array into two different pieces, A and B:

$$A = a_{[0:x]}$$

$$B = a_{[x:]}$$

As the two different sets contain the values of a where set A is equivalent to the 1st to xth values of a (or indices 0 to x), where set B is equivalent to the xth to last index of a. As the two pieces are split based on the xth index, the union of the two different sets in reverse order will be equivalent to the shifted array:

$$a_s = B \cup A$$

Though with set theory this solution would not differ from $A \cup B$, order of the array counts therefore a_s is array a shifted x times. The inverse array shift function is simply an array shift forward x times, following the following procedure:

$$A = a_{[-x:]}$$

$$B = a_{[:-x]}$$

$$a_s = A \cup B$$

A negative index refers to the xth to last element of a, simply the inverse of the original array shift function. With the array shift functions, cipher arrays can be obfuscated further though with a degree of reversibility.

6. Implementation Issues

6.1 Security Flaws

All constants must be relatively large numbers for an encrypted message to be secure. The lower the constants are, the easier it is for hackers to bruteforce the cipher and break it. Referenced in Sec 5.3, if the user is using the alternative way to encrypt a message, unless padded the message is vulnerable to a substitution attack, which allows hackers to infer what the message is based on the frequency of the characters in the message. Furthermore, another problem is that there is no known way to securely exchange constants without the risk of interception if not using an additional algorithm to exchange them. Prime constants are the most ideal for encryption, and hence are recommended for HC.

6.2 Numeric Restrictions

As the values for encryption approach larger values, the overall cipher length will significantly increase. For computers which are not as capable as others, packets will be received in fragments, which may cause complications with exchanging information.

7. Python Implementation

Below is the Python implementation of this standard:

```
import math
class RSA:
   def __init__(self, p,q,e=0x10001):
       self.p = p
       self.q = q
       self.e = e
       self.n = p * q
       self.phi = (p - 1) * (q - 1)
       self.d = pow(e,-1,self.phi)
   def encrypt(self,msg,isnum=False):
       if not isnum:
           msg = int(msg.encode().hex(),16)
       return pow(msg,self.e,self.n)
   def decrypt(self,c,isnum=False):
       if not isnum:
           c = int(c.encode().hex(),16)
           return bytes.fromhex(hex(pow(c,self.d,self.n))[2:]).decode()
       else:
           return pow(c,self.d,self.n)
class HydraCrypt:
   class ConstraintsNotMetError(Exception):
       def __init__(self,msg="""One or more of the constraints are not met! Check if
your constants fit within these contraints:\n{k, r, q, t, u, bls} ⊆ Z^+\nk > 0\nbls >
1\nk > 0\nr > 0\nq > 0\nt > 0\nu > 0"""):
           self.msg = msg
           super().__init__(self.msg)
   class NegativeBinaryValueError(Exception):
```

```
        def __init__(self,msg="The value of your plaintext is a negative value, are your
constants large enough to correctly encrypt or decrypt the message?"):
            self.msg = msg
            super().__init__(self.msg)
    def __init__(self,k,r,q,t,u,bls=10):
        self.k = k
        self.r = r
        self.q = q
        self.t = t
        self.u = u
        self.bls = bls
        self.shift = k % self.bls
        self.rsa = RSA(t,u)
        if not (k > 0 and r > 0 and q > 0 and t > 0 and u > 0 and self.bls > 1):
            raise self.ConstraintsNotMetError
    @staticmethod
    def ln(x):
        return math.log(math.e,x)
    def h(self,x):
        return x + (self.r+self.q)
    def h_(self,x):
        return x - (self.r+self.q)
    def f(self,x):
        return self.h(int(x,2) + self.k**((self.q+self.r)%2**4)) ^
self.rsa.encrypt(abs(self.q-self.r)%(2**(self.t*self.u).bit_length()-1),True)
    def f_(self,x,cn):
        return bin((self.h_(x ^
self.rsa.encrypt(abs(self.q-self.r)%(2**(self.t*self.u).bit_length()-1),True)) -
(self.k**((self.q+self.r)%2**4))))[2:].zfill(cn)
    @staticmethod
    def P(c):
        return 4*c**3 + 10*c**2 + 15*c
    @staticmethod
    def T(c):
        return c**5 + 4*c**4 + 3*c**5 + c**2 + 7*c
    def Q(self,c,i):
        return abs((self.P(c) - self.P(c - i)) + (self.T(c) - self.T(c - i)))
    def W(self,x):
        return 7*self.ln(abs(x - 2)) - 4*self.ln(abs(x - 1))
    def V(self,x):
        return 3*x**3 + (x**2)//2 - 5*x
    def s(self,i):
        return ((20*i**3+105*i**2+385*i)//3)**(int(math.sqrt(self.bls)//2))
    def X(self,i):
        return int((self.W(i + self.k) - self.W((i % self.k) - (i + self.k))) +
(self.V(i + self.k) - self.V((i % self.k) - (i + self.k))))**(((i*self.k) % 1000)+1) +
self.q**2 + self.r**2 + self.t**2 + self.s(i)
    @staticmethod
    def rshift_a(ls,s):
        ls1 = ls[0:s]
        ls2 = ls[s:]
        return ls2 + ls1
    @staticmethod
    def shift_a(ls,s):
        ls1 = ls[-s:]
        ls2 = ls[:-s]
        return ls1 + ls2
    def encrypt(self,num):
```

```
        num += 10**(self.bls//2)
        m = []
        n_ = bin(num)[2:]
        cn = len(bin(num)[2:])//self.bls
        m = []
        for i in range(self.bls):
            m.append(n_[:cn])
            n_ = n_[cn:]
        l_n = len(n_)
        try:rem = int(n_,2)
        except:rem = 0
        m = self.shift_a(m,self.shift)
        for i in range(len(m)):
            m_i = self.s(i+1)
            m[i] = self.f(m[i])
            m[i] += m_i
            m[i] ^= self.Q(m_i,i+1)
        try:
            rem = self.f(n_)
        except:
            rem = self.f("0")
        x_i = self.s(self.bls+1)
        rem += x_i
        rem ^= self.Q(x_i,self.bls+1)
        return [m,cn,rem,l_n]
    def decrypt(self,m,cn,rem,l_n):
        x_i = self.s(self.bls+1)
        rem ^= self.Q(x_i,self.bls+1)
        rem -= x_i
        for i in range(len(m)):
            m_i = self.s(i+1)
            m[i] ^= self.Q(m_i,i+1)
            m[i] -= m_i
            m[i] = self.f_(m[i],cn)
        m = self.rshift_a(m,self.shift)
        x = ""
        for i in range(self.bls):
            x += m[i]
        rem = self.f_(rem,l_n)
        x += rem
        if "b" in x:
            raise self.NegativeBinaryValueError
        return int(x,2) - 10**(self.bls//2)
    def pad(self,cls,i):
        cls = list(cls)
        ic = self.X(i+2)
        for j in range(len(cls[0])):
            cls[0][j] ^= ic
        cls[2] ^= ic
        return cls
```