Freestyle, a randomized version of ChaCha for resisting offline brute-force and dictionary attacks

P. Arun Babu[1,*], Jithin Jose Thomas[2,*]

Abstract

This paper introduces *Freestyle*, a randomized, and variable round version of the ChaCha cipher. Freestyle demonstrates the concept of *hash based halting condition*, where a decryption attempt with an incorrect key is likely to take longer time to halt. This makes it resistant to key-guessing attacks i.e. brute-force and dictionary based attacks. Freestyle uses a novel approach for ciphertext randomization by using random number of rounds for each block of message, where the exact number of rounds are unknown to the receiver in advance. Due to its inherent random behavior, Freestyle provides the possibility of generating up to 2^{256} different ciphertexts for a given key, nonce, and message; thus resisting key and nonce reuse attacks. This also makes cryptanalysis through known-plaintext, chosen-plaintext, and chosen-ciphertext attacks difficult in practice. Freestyle is highly customizable, making it suitable for both low-powered devices and security-critical applications. It is ideal for: (i) applications that favor ciphertext randomization and resistance to key-guessing and key reuse attacks; and (ii) situations where ciphertext is in full control of an adversary for carrying out an offline key-guessing attack.

Keywords: Brute-force resistant ciphers, Dictionary based attacks, Key-guessing penalty, Probabilistic encryption, Freestyle, ChaCha

2010 MSC: 00-01, 99-00

*Corresponding author

Email address: barun@iisc.ac.in (P. Arun Babu)

[1]Robert Bosch Center for Cyber-Physical Systems, Indian Institute of Science, Bengaluru

[2]Department of Electrical and Computer Engineering, Texas A&M University

1. Introduction

A randomized (*aka* probabilistic) encryption scheme involves a cipher that uses randomness to generate different ciphertexts for a given *key*, *nonce* (*a.k.a.* initial vector), and *message*. The goal of randomization is to make cryptanalysis difficult and a time-consuming process. This paper presents the design and analysis of *Freestyle*, a highly customizable, randomized, and variable-round version of ChaCha cipher [1]. ChaCha20 (i.e. ChaCha with 20 rounds) is one of the modern, popular (for TLS [2] and SSH [3, 4]), and faster symmetric stream cipher on most machines [5, 6]. Even on lightweight ciphers, realistic brute-force attacks with *key* sizes $\geq$ 128 bits is not feasible with current computational power. However, algorithms and applications that have lower key-space due to: (i) generation of keys from a poor (pseudo-)random number generator [7, 8, 9, 10, 11, 12]; (ii) weak passwords [13] and poor implementations of password based key derivation [14, 15]; and, (iii) poor protocol or cryptographic implementations [16, 17, 18] are prone to key-guessing attacks (brute-force and dictionary based attacks). Such attacks are also becoming increasingly feasible due to steady advances in the areas of GPUs [19, 20, 21], specialized hardware for cryptography [22, 23, 24, 25, 26, 27], and memories in terms of storage size and in-memory computations [28, 29, 30, 31].

Techniques such as introducing a delay between incorrect key/password attempts, multi-factor authentication, and CAPTCHAs (Completely Automated Public Turing test to tell Computers and Humans Apart) are being used to resist brute-force attacks over the network (i.e. on-line brute-force attack). However, such techniques cannot be used if the ciphertext is in full control of the adversary (i.e. offline brute-force attack); for example: encrypted data gathered from a wireless channel, or lost/stolen encrypted files/disks. To resist offline brute-force attacks, key-stretching and slower algorithms [32] are preferred. Although, such techniques are useful, they are much slower on low-powered devices, and also slow down genuine users.

1.1. Our contribution

This paper makes *three* main contributions: (i) We demonstrate the use of bounded *hash based halting condition*, which makes key-guessing attacks less effective by slowing down the adversary, but remaining relatively computationally light-weight for genuine users. To quantify the penalty an adversary has to pay in terms of computational power, we introduce a metric called the Key-guessing penalty (KGP).

Definition : Key-guessing penalty (KGP) for encryption - The ratio of expected time taken to attempt decryption of a *message* using an incorrect *key*, and the expected time taken to decrypt a *message* using the correct *key* (equation 1).

$$\text{KGP} = \frac{time(\text{attempt decryption of a } message \text{ using an incorrect } key)}{time(\text{decrypt the } message \text{ using the correct } key)} \quad (1)$$

Similarly:

Definition : Key-guessing penalty (KGP) for password hashing - The ratio of expected time taken to verify a password hash using an incorrect *password*, and the expected time taken to verify a password *hash* using the correct *password* (equation 2).

$$\text{KGP} = \frac{time(\text{to verify a password } hash \text{ using an incorrect } password)}{time(\text{to verify a password } hash \text{ using a correct } password)} \quad (2)$$

The physical significance of KGP is that the adversary would require at least KGP times computational power than a genuine user to launch an effective key-guessing attack; (ii) We demonstrate a novel approach for ciphertext randomization by using random number of rounds for each block of message; where the exact number of rounds are unknown to the receiver in advance; (iii) We introduce the concept of *non-deterministic CTR mode* of operation and demonstrate the possibility of using the random round numbers to generate up to 2^{256} different ciphertexts - even though the *key*, *nonce*, and *message* are the same. The randomization makes the cipher resistant to key re-installation attacks such as KRACK [16] and cryptanalysis by XOR of ciphertexts in the event of the *key* and *nonce* being reused.

An interesting feature of Freestyle's decryption algorithm is: that it is designed to be computationally light-weight for a user with a *correct key*; but, for an adversary with an *incorrect key*, the decryption algorithm is likely to take longer time to halt. Thus, each key-guessing attempt is likely to be computationally expensive and time-consuming.

1.2. Related work

1.2.1. Randomized encryption schemes

Use of randomized encryption schemes have been in practice for many years, and a taxonomy of randomized ciphers is presented in [33]. Also, some approaches to randomized encryption for public-key cryptography was proposed in [34, 35, 36]. Approaches based on chaotic systems for probabilistic encryption were also proposed [37]. However, the main concern with some of the existing approaches are high bandwidth expansion factor and computational overhead [38, 33].

One of the approaches in practice is to generate random bytes and sending it in the encrypted form. The random bytes along with the *key* will be used for encryption/decryption [33]. Though such approaches are capable of generating large number of ciphertexts for a given *message* and a *key*; they do not provide the possibility of KGP > 1. Also, for stream-ciphers, if the *key* and *nonce* are reused, there is a possibility of cryptanalysis by XOR-ing ciphertexts. Also, in Freestyle, the random bytes are never sent to the receiver in plain nor in the encrypted form. The random bytes must be computed by the receiver from the initial I_h *hash*es. The initial I_h *hash*es also serve the purpose of preventing an adversary from sending arbitrary ciphertext, thus resisting CCA if the *nonce* cannot be controlled by the adversary. Also, Freestyle offers the possibility of generating up to 2^{256} different ciphertexts even if *key*, *nonce*, and other cipher parameters are reused. Also, unlike some of the existing randomized ciphers, Freestyle has a low bandwidth overhead of $\approx 1.5625\%$.

1.2.2. Approaches based on difficulty and proof of work

Several algorithms have been proposed in literature to increase the difficulty in key and password guessing using a CPU intensive key-stretching [39] or key-setup phase [40] using a cost-factor. Also, approaches that consume large amount of memory have also been proposed [41, 42]. Another related area is use of client puzzles [43] and proof-of-work (e.g. Bitcoin [44]) to delay cryptographic operations.

The *hash based halting condition* described in Section 3, on a high-level use similar principle as the Halting Key Derivation Function (HKDF) proposed in [43]. In HKDF, a sender with a password and random bytes, uses the key derivation function till n iterations (or based on certain amount of time) to generate a *key* and a publicly verifiable *hash*. On the other hand, the receiver uses the random bytes and password to generate the *key* till the verifiable *hash* matches.

The rest of the paper is structured as follows: Table 1 lists the notations used in the paper; Section 2 presents the background information on ChaCha cipher and its variants; Section 3 describes the Freestyle cipher; Section 4 presents results and cryptanalysis of Freestyle cipher; Section 1.2 presents related work; and Section 5 concludes the paper.

2. ChaCha cipher and variants

ChaCha20 [1] is a variant of Salsa20 [45, 46], a stream cipher. It uses 128-bit *constant*, 256-bit *key*, 64-bit *counter*, and 64-bit *nonce* to form an initial cipher-state denoted by $S^{(0)}$, as:

$$\begin{bmatrix} constant[0], & constant[1], & constant[2], & constant[3] \\ key[0], & key[1], & key[2], & key[3] \\ key[4], & key[5], & key[6], & key[7] \\ counter[0], & counter[1], & nonce[0], & nonce[1] \end{bmatrix}$$

Table 1: List of symbols.

Notation	Description
R_{min}	Indicates the minimum number of rounds to be used for encryption/decryption. $R_{min} \in [1, 255]$.
R_{max}	Indicates the maximum number of rounds to be used for encryption/decryption. $R_{max} \in [R_{min} + 1, 255]$.
R	Number of rounds used to encrypt the current block of message. $R = random(R_{min}, R_{max})$
R_i	Number of rounds used to encrypt i^{th} block of message. $R_i = random(R_{min}, R_{max})$ and $i \geq 0$.
r	The current round number. $r \in [1, R]$
$hf()$	Freestyle hash function which generates an 8-bit $hash$ from a 144-bit input.
H_I	Round intervals at which an 8-bit $hash$ has to be computed. H_I is set to 1 at cipher initialization; and during encryption/decryption H_I is set to $gcd(R_{min}, R_{max})$.
P_b	The number of $pepper$ bits to be used during cipher initialization. $P_b \in [8, 32]$
I_h	The number of initial hashes/round-numbers to be used for cipher initialization. $I_h \in [7, 56]$
P_r	The number of rounds to be pre-computed. $P_r \in [0, 15]$ and $P_r \leq R_{min} - 4$
C_p	The 32-bit cipher-parameter created by concatenating $R_{min}, R_{max}, P_b, I_h, P_r$
$pepper$	The number of iterations required for cipher initialization. $pepper = random(0, 2^{P_b} - 1)$.
R_i^*	The number of rounds computed using the expected $hash$ and $pepper$ for i^{th} block of $message$.
$rand$	The 256-bit value that is generated at the sender side and is to be computed at the receiver side.
$\mathbb{E}[pepper]$	The expected value of $pepper$.
$\mathbb{E}[R^+]$	The expected number of rounds executed by an user when an incorrect key or $pepper$ is used.
$\mathbb{E}[R]$	The expected number of rounds executed by a genuine user to encrypt/decrypt a block of $message$. If a uniform (P)RNG is used, then $\mathbb{E}[R] = \frac{R_{min} + R_{max}}{2}$.
$v^{(r)}$	The value of v after r rounds of Freestyle. If $v^{(0)}$ is not explicitly defined, then $v^{(0)} = 0$.
$v[n]$	n^{th} element of v.
$a \oplus b$	Bit-wise XOR of a and b.
$a \boxplus b$	Addition of a and b modulo 2^{32}.
$\|v\|$	The length of v in bits.
N_b	The number of blocks in a $message$. $N_b = \left\lceil \frac{\|message\|}{512} \right\rceil$
$Pr_n(X = 1)$	The probability of getting a valid round number at the n^{th} trial, when using an incorrect key or $pepper$.
N_c	The total number of ciphertexts possible for a given: key, $nonce$, and $message$.
N_r	The number of ways a block of message can be encrypted using random number of rounds (R). $N_r = \left(\frac{R_{max} - R_{min}}{H_I} + 1 \right)$
$time(o)$	The expected time taken to execute the operation 'o'.
S	The 512-bit cipher-state for a given block of $message$.
$counter$	The counter in the CTR mode of operation.
$null$	An empty string.
$\mathcal{R}$	A random number that is independent of the key, $nonce$, $message$, and $pepper$.

ChaCha20 uses 10 double-rounds (or 20 rounds) on $S^{(0)}$; where each of the double-rounds consists of 8 quarter rounds (QR) defined as:

Odd round	**Even round**
$QR\,(S[0], S[4], S[\ 8], S[12])$	$QR\,(S[0], S[5], S[10], S[15])$
$QR\,(S[1], S[5], S[\ 9], S[13])$	$QR\,(S[1], S[6], S[11], S[12])$
$QR\,(S[2], S[6], S[10], S[14])$	$QR\,(S[2], S[7], S[\ 8], S[13])$
$QR\,(S[3], S[7], S[11], S[15])$	$QR\,(S[3], S[4], S[\ 9], S[14])$

(3)

where the 16 elements of the cipher-state matrix are denoted in row-wise fashion, using an index in the range [0,15]. And the quarter-round $QR(a, b, c, d)$ is defined as:

$$\begin{aligned} a \leftarrow a \boxplus b; \quad d \leftarrow d \oplus a; \quad d \leftarrow d \lll 16; \\ c \leftarrow c \boxplus d; \quad b \leftarrow b \oplus c; \quad b \leftarrow b \lll 12; \\ a \leftarrow a \boxplus b; \quad d \leftarrow d \oplus a; \quad d \leftarrow d \lll \ \ 8; \\ c \leftarrow c \boxplus d; \quad b \leftarrow b \oplus c; \quad b \leftarrow b \lll \ \ 7; \end{aligned} \tag{4}$$

After 20 rounds, the initial state ($S^{(0)}$) is added to the current state ($S^{(20)}$) to generate the 512-bit *keystream* (equation 5).

$$keystream = S^{(0)} \boxplus S^{(20)} \tag{5}$$

The *keystream* is then converted to little-endian format and XOR-ed with a block (512 bits) of plaintext/ciphertext to generate a block of ciphertext/-plaintext. The above operations are performed for each block of *message* to be encrypted/decrypted.

ChaCha is a simple and efficient ARX (Add-Rotate-XOR) cipher, and is not sensitive to timing attacks. ChaCha has two main flavors with reduced number of rounds i.e. with 8 and 12 rounds. ChaCha12 is considered secure enough as there are no known attacks against it yet [47]. ChaCha20 has two main variants: (i) IETF's version of ChaCha20 [2, 48] which uses a 32-bit *counter* (instead of 64-bit) and 96-bit *nonce* (instead of 64-bit); and (ii) XChaCha20 [49], which uses 192-bit *nonce* (instead of 64-bit), where a randomly generated *nonce* is considered safe enough [50]. The larger *nonce* in XChaCha20 makes the probability of *nonce* reuse low.

3. The Freestyle cipher

Freestyle's core is similar to the IETF's version of ChaCha, but uses *hash based halting condition*. Traditionally ciphers are designed to use fixed number of rounds in the encryption and decryption process. Even in variable round ciphers, the number of rounds is well-known in advance. This makes the cipher to take nearly the same amount of time to execute the decryption function, irrespective of the *key* being correct or incorrect. This is advantageous for an adversary if the cipher is lightweight and parallelizable. To resist such attacks, Freestyle uses the concept of hash based halting condition.

It works on the following principle: a sender encrypts a block of message using a random number of round (R), which is never shared with the receiver. However, the sender along with the ciphertext shares the *hash* of the cipher state (or partial cipher-state) after executing R rounds. The *hash* is sent in cleartext; and the receiver can compute R using the correct *key*, and the received *hash*. This expected *hash* acts as a halting condition for the decryption process; i.e. the receiver has to keep executing the decryption algorithm till the computed *hash* matches the expected *hash*. For an adversary using brute-force or dictionary based attack, since the *key* is incorrect, during the decryption process, the *hash* is expected to take longer time to match. This asymmetry makes offline brute-force and dictionary based attacks less efficient.

The proposed approach makes the assumption that: (i) the hash function is secure enough, that from the *hash* it is computationally infeasible to compute the number of rounds, *key*, or any other secret information; (ii) the round number (R) is generated using a good uniform (P)RNG like hardware random number generator or cryptographically secure pseudo-random number generator (CPRNG) (e.g. `arc4random` [51]).

To achieve hash based halting condition and ciphertext randomization, Freestyle uses the following 5 parameters:

1. $R_{min} \in [1, 255]$, indicating the minimum number of rounds to be used for encryption/decryption. R_{min} is recommended to be ≥ 8; however, for

security-critical applications: $R_{min} \geq 12$ is preferred.

2. $R_{max} \in [R_{min} + 1, 255]$, indicating the maximum number of rounds to be used for encryption/decryption. Using R_{min} and R_{max}, for each block of *message*, a round number (R) is generated randomly by the sender which will be used to encrypt the current block of message.

3. $P_b \in [8, 32]$, indicating the number of *pepper* bits to be used during cipher initialization. P_b determines the number of iterations that will be needed to initialize the cipher. The *pepper* serves the same function as *salt*; however, it may not be stored along with the hash or ciphertext (i.e. can be forgotten by the sender after use) [52, 42]. $P_b \geq 16$ is recommended for security-critical applications.

4. $I_h \in [7, 56]$, indicates the number of initial random-rounds to be used for cipher initialization. I_h determines the number of possible ciphertexts that can be generated for a given *key*, *nonce*, and *message*. $I_h \geq 28$ is recommended for security-critical applications.

5. $P_r \in [0, 15]$ and $P_r \leq (R_{min} - 4)$, indicates the number of Freestyle rounds to be pre-computed.

Using the values of R_{min} and R_{max}, a hash interval denoted by $H_I \in [1, R_{min}]$ is computed. H_I indicates the round intervals at which an 8-bit *hash* of partial cipher-state must be computed. The value of H_I is set to 1 during cipher-initialization and is set to $gcd(R_{min}, R_{max})$ during encryption and decryption. For a given R_{min} and R_{max}, the $gcd(R_{min}, R_{max})$ is the optimal value of hash interval possible for performance. At the time of encryption, the sender computes the *hash* after every H_I rounds; and at the end of R rounds the *hash* is sent to the receiver. On the other hand, while decrypting a block of message, the receiver computes the *hash* after every H_I rounds; and decryption is stopped only if it matches with the received *hash*. While decrypting a block of message, if the computed *hash* does not match the expected *hash* even after executing R_{max} rounds, then either the *key*, *nonce*, or one of the parameters provided by the receiver is incorrect.

R_{min}	R_{max}	0 (unused)	P_b	I_h	P_r
(8 bits)	(8 bits)	(1 bit)	(5 bits)	(6 bits)	(4 bits)

Figure 1: The 32-bit cipher-parameter (C_p).

Remark 1 It must be noted that *hash*es are computed only after executing R_{min} rounds. This avoids accidentally terminating after executing fewer than expected number of rounds.

Remark 2 The performance of Freestyle is $\propto \frac{H_I \times P_r}{R_{min} \times R_{max} \times P_b \times I_h}$. Thus the parameters must be carefully chosen based on the required security level and performance.

3.1. The initial cipher-state ($S^{(0)}$)

The initial cipher-state of Freestyle, denoted by $S^{(0)}$ (equation 6) is a 4×4 matrix of 32-bit words consisting of: 128-bit *constant*, 256-bit *key*, 32-bit *counter*, and 96-bit *nonce*. Unlike ChaCha [1], the *counter* size has been reduced to 32-bit as in practice most of the protocols such as the SSH transport protocol [53] recommend re-keying after every 1 GB of data sent/received.

Freestyle's initial cipher-state is similar to the IETF's version of ChaCha, except that the *constant*s are modified using the cipher-parameter (C_p). C_p is formed by concatenating all the 5 parameters i.e. R_{min}, R_{max}, P_b, I_h, and P_r to generate a unique 32-bit string as shown in the Figure 1. The C_p is then XORed with the *constant*[0] (equation 6). This step makes encryption with one cipher-parameter incompatible with other cipher-parameters by design; thus, cryptanalysis data collected from a cipher with weaker cipher-parameter cannot

be reused against a cipher with stronger cipher-parameter.

$$S^{(0)} = \begin{bmatrix} \begin{pmatrix} constant[0] \\ \oplus \\ C_p \end{pmatrix}, & constant[1], & constant[2], & constant[3] \\ key[0], & key[1], & key[2], & key[3] \\ key[4], & key[5], & key[6], & key[7] \\ counter, & nonce[0], & nonce[1], & nonce[2] \end{bmatrix} \quad (6)$$

3.2. Initialization for encryption

After the initial cipher-state ($S^{(0)}$) is computed, the following temporary configuration is set irrespective of the cipher-parameter (C_p):

$$R_{min} = 8,\ R_{max} = 32,\ H_I = 1,\ \text{and}\ P_r = 4 \quad (7)$$

This is done to ensure there is enough entropy even if weaker values of R_{min} and R_{max} are set by the user. This step also helps in cases where the parameters can be downgraded in Man in the middle (MiTM) attacks such as Logjam [18].

As the number of pre-computed rounds (P_r) is now set to 4; 4 rounds of Freestyle are pre-computed using 0 as the *counter*; which results in $S^{(4)}$. After which, a random *pepper* from $[0, 2^{P_b})$ is generated by the sender and added to

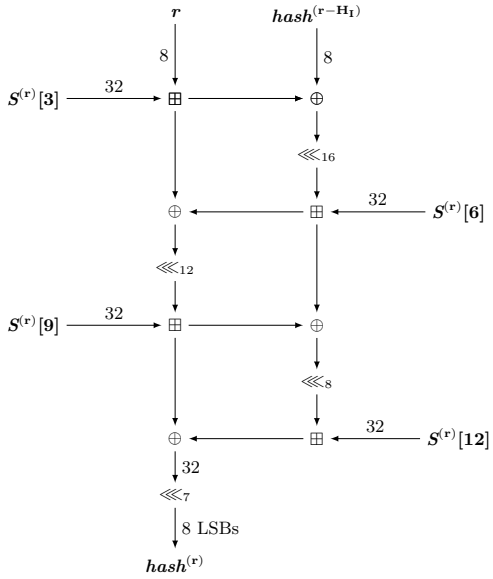

Figure 2: The Freestyle hash function - $hf()$, for the round r (the size of variables are in bits). Note that $hash^{(R_{min}-H_I)} = 0$.

$S^{(4)}[0]$ to form the intermediate cipher-state (S^*) as shown in equation 8.

$$S^* = \begin{bmatrix} \begin{pmatrix} S^{(4)}[0] \\ \boxplus \\ pepper \end{pmatrix}, & S^{(4)}[1], & S^{(4)}[2] & S^{(4)}[3] \\ \\ S^{(4)}[4], & S^{(4)}[5], & S^{(4)}[6] & S^{(4)}[7] \\ S^{(4)}[8], & S^{(4)}[9], & S^{(4)}[10] & S^{(4)}[11] \\ S^{(4)}[12], & S^{(4)}[13], & S^{(4)}[14] & S^{(4)}[15] \end{bmatrix} \tag{8}$$

After which, $S^*[12]$ is used as the *counter* in CTR mode to generate I_h number of cipher states. Here, as 4 rounds have already been pre-computed, each of the I_h blocks will now only require $(R_i - 4)$ additional rounds (where, $R_i = random(8, 32), \forall i \in [0, I_h)$). Then, from each of the I_h number of cipher states, I_h number of expected *hash*es are computed using Freestyle's hash function (Figure 2, code in Appendix - A).

The above *hash*es are used for the *hash based halting condition* described earlier in Section 3. Freestyle's hash function generates an 8-bit *hash* using: (i) the 8-bit current round number (r), (ii) the 128 bits from the anti-diagonal elements of the current cipher-state ($S^{(r)}$), and (iii) the 8-bit previous *hash* (i.e. $hash^{(r-H_I)}$).

It must be noted that at this point only I_h number of *hash*es are generated, and no encryption is performed yet. The sender then using the I_h number of random round numbers: $\{R_0, R_1, ..., R_{I_h-1}\}$, computes a 256-bit *rand* using ADD-XOR-Rotate instructions as shown in Figure 3. The number of possible values of *rand* is dependent on the I_h value, and Freestyle allows I_h to be in range $[7, 56]$. Where, each of the initial hash contributes to $\frac{32}{7}$ bits of *rand* (from Figure 3). Hence, if $I_h = 7$, then Freestyle can generate 2^{32} possible values of *rand*; whereas for $I_h = 56$, Freestyle can generate 2^{256} possible values of *rand*. Hence, by using I_h, a user may control the randomness of the cipher. Freestyle enforces $I_h \geq 7$, thus providing the possibility of generating at-least 2^{32} possible ciphertexts for a given *key*, *nonce*, and a *message*.

The *rand* value is then used to modify the current cipher-state (S^*) as shown in equation 9. This makes Freestyle resistant to chosen IV attacks [54] and cryptanalysis due to potential biases and Probabilistic Neutral Bits (PNBs) [55, 47].

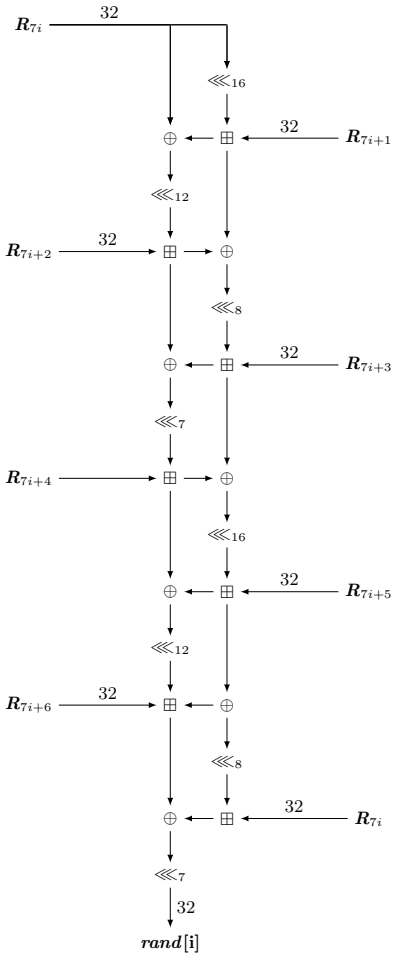

Figure 3: Generation of $rand[i]$, where $i \in [0, 7]$ (the size of variables are in bits).

$$S^* \leftarrow \begin{bmatrix} S^*[0], & \begin{pmatrix} S^*[1] \\ \oplus \\ rand[1] \end{pmatrix}, & \begin{pmatrix} S^*[2] \\ \oplus \\ rand[2] \end{pmatrix}, & \begin{pmatrix} S^*[3] \\ \oplus \\ rand[3] \end{pmatrix} \\ \begin{pmatrix} S^*[4] \\ \oplus \\ rand[4] \end{pmatrix}, & \begin{pmatrix} S^*[5] \\ \oplus \\ rand[5] \end{pmatrix}, & \begin{pmatrix} S^*[6] \\ \oplus \\ rand[6] \end{pmatrix}, & \begin{pmatrix} S^*[7] \\ \oplus \\ rand[7] \end{pmatrix} \\ S^*[8], & S^*[9], & S^*[10], & S^*[11] \\ S^*[12], & S^*[13], & S^*[14], & S^*[15] \end{bmatrix} \tag{9}$$

After which, the values of R_{min}, R_{max}, H_I, and P_r are set back to its original values (as set by the user); and the current cipher-state (S^*) is used as an input to pre-compute P_r number of rounds using $S^*[12]$ as the *counter* (equation 10).

$$S^* \leftarrow (S^*)^{(P_r)} \tag{10}$$

From now on, the new cipher-state (S^*) will be used as an input to generate the *keystream* for encryption. It is to be noted that since P_r rounds have been pre-computed, for encrypting the i^{th} block of a message, only $(R_i - P_r)$ rounds are required.

3.3. Encryption

As described earlier in Section 3.2, for initialization, Freestyle uses plain CTR mode of operation for the first I_h blocks. However, for encryption, Freestyle uses *non-deterministic CTR mode* (Figure 4). Here, we introduce the concept of *non-deterministic CTR mode* of operation: in this mode, the *counter* is XOR-ed with a secret random number ($\mathcal{R}$) that is independent of the *key*, *nonce*, *message*, and *pepper* (unlike randomized-CTR mode where the bytes used to modify the

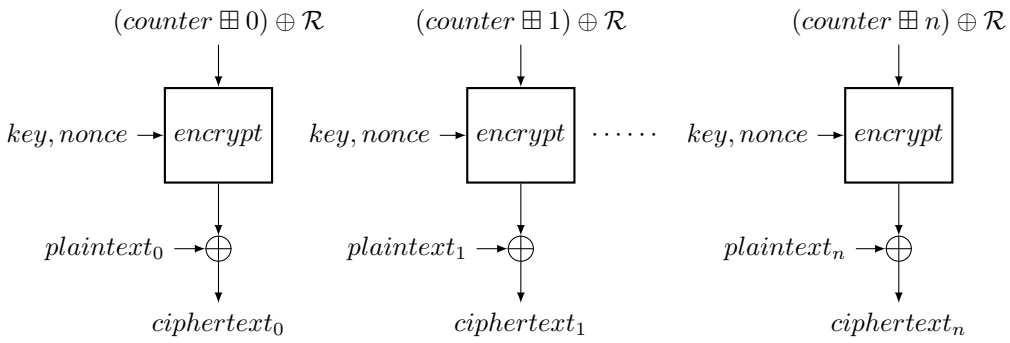

Figure 4: Non-deterministic CTR mode of operation, where the *counter* is XOR-ed with a random number ($\mathcal{R}$) that is independent of the *key*, *nonce*, *message*, and *pepper*. In case of Freestyle, *counter* is $S^{(4)}[12]$ and $\mathcal{R}$ is $rand[0]$.

counter is derived from the *key* and/or *nonce*). The non-deterministic CTR mode offers two main benefits: (i) it eliminates the need for setting the initial value for the *counter*; and (ii) the *counter*s are now chosen from a secret random permutation of the set $[0, 2^{N_b})$, that is independent of the *key*, *nonce*, *message*, and *pepper*.

In Freestyle, the random number ($\mathcal{R}$) to be XOR-ed with the *counter* (i.e. $S^*[12]$) is $rand[0]$. The rationale behind choosing $rand[0]$ to be XOR-ed with the counter is: the minimum possible value of I_h is 7, which can only generate a 32-bit random number i.e. $rand[0]$. The values of $rand[i], \forall i \in [1, 7]$ will be 0 in this case. Hence, in the worst-case scenario, the *counter*s used for encryption/decryption are random and unknown to an adversary.

Remark 3 The non-deterministic CTR mode may appear similar to key-whitening technique. However, in the non-deterministic CTR mode, the *counter* is XOR-ed with a random number independent of the *key*, *nonce*, *pepper*, or the *message*. And the random number is likely to change for each initialization even if *key*, *nonce*, *pepper*, and *message* are reused. Also, unlike key-whitening schemes, non-deterministic CTR mode in Freestyle does not require extra *key* bits to resist key-guessing attacks. ■

Using the S^* from the equation 10, the *keystream* for a block of plaintext is computed by running random number of Freestyle rounds $(R_i - P_r)$ on S^*. The S_i^* is then added to S^* to generate the keystream (equation 12). The keystream is then XOR-ed with a block of plaintext to generate a block of ciphertext. For a given i^{th} block of a message, $\forall i \in [0, N_b)$ the ciphertext is generated as shown in equations 11, 12, and 13.

$$S_i^* = (S^*)^{(R_i - P_r)} \tag{11}$$

$$keystream_i = S_i^* \boxplus S^* \tag{12}$$

$$ciphertext_i = plaintext_i \oplus keystream_i \tag{13}$$

Where $R_i = random(R_{min}, R_{max}), \forall i \in [0, N_b)$; and N_b is the total number of blocks in a message.

3.4. Initialization for decryption

For initializing the cipher for decryption, the receiver first computes cipher-parameter (C_p) and sets the following temporary configuration (same as equation 7):

$$R_{min} = 8,\ R_{max} = 32,\ H_I = 1,\ \text{and}\ P_r = 4 \tag{14}$$

and computes $S^{(4)}$ using the *key* and *nonce*. Then the receiver iterates *pepper* from 0 to $(2^{P_b} - 1)$ using equation 8, until I_h number of valid round numbers are found, corresponding to each of the received I_h number of *hash*es. Once successful, the receiver computes the 256-bit *rand* value using the valid round numbers: $\{R_0, R_1, ..., R_{I_h - 1}\}$, as shown in Figure 3.

Using the *rand*, the new cipher-state (S^*) is computed as shown in equation 9. The original values of R_{min}, R_{max}, H_I, and P_r provided by the user are restored; and P_r number of rounds are pre-computed (equation 10).

3.5. Decryption

Similar to the encryption (Section 3.3), non-deterministic CTR mode is used to decrypt all the blocks of message. And, the *keystream* is generated using S^* (equations 11 and 12). The plaintext is generated by XOR-ing the ciphertext with *keystream* (equation 15).

$$plaintext_i = ciphertext_i \oplus keystream_i \tag{15}$$

Similar to encryption, since P_r rounds have been pre-computed, for decrypting the i^{th} block of a message, only $(R_i - P_r)$ rounds are required.

4. Results and discussions

4.1. Number of possible ciphertexts

For a given *message* of length $|message|$ bits, the *message* is divided into $N_b = \left\lceil \frac{|message|}{512} \right\rceil$ blocks. Since, each block can be encrypted with a random number (R) of rounds in the range $[R_{min}, R_{max}]$; the total number of ways a given block of *message* can be encrypted using random number of rounds is denoted by N_r, given as:

$$N_r = \frac{R_{max} - R_{min}}{gcd(R_{min}, R_{max})} + 1 \tag{16}$$

And since all the N_b blocks of the *message* use the P_b bits of *pepper*, and $\frac{32 \times I_h}{7}$-bit *rand* as inputs; the total number of possible ciphertexts are:

$$N_c = 2^{P_b} \times 2^{\left(\frac{32}{7} \times I_h\right)} \times (N_r)^{N_b} \tag{17}$$

From equation 17, as the number of *pepper* bits, number of initial *hash*es, or the number of blocks in a *message* increases, the number of possible ciphertexts for a given *key*, *message*, and *nonce* increases exponentially.

4.2. Resistance to cryptanalysis

This section presents some of the results on Freestyle's resistance to cryptanalysis. Here we restrict our analysis to the additional benefits offered by Freestyle; as ChaCha with 12 rounds is known to be secure [47]. And the detailed cryptanalysis of ChaCha can be found in [55, 54, 47, 56, 57, 58].

4.2.1. Cryptanalysis using the hash

Unlike a typical cryptographic hash function, Freestyle does not require high collision-resistant hash function; the probability of 2^{-8} for collision is sufficient for its purpose. The hash function handles collisions by incrementing the *hash* till there is no collision (equation 18 and appendix E - line:47).

$$hash \leftarrow (hash + 1) \pmod{256} \tag{18}$$

Freestyle's hash function uses 128-bits of the cipher-state ($S^{(R)}$), at least 6-bits of current round number (r), and 8 bits of previous hash ($hash^{(r-H_I)}$). Hence, to generate all possible partial cipher-states that may collide with a given *hash* (Figure 2) will require 2^{142} operations. Also, assuming the 8-bit *hash*es are equally spread over 256 buckets, there are likely to be 2^{134} collisions. Hence, for an attacker it is not feasible to use the *hash* information to compute *key* bits.

The hash function uses Add-Rotate-XOR (ARX) operations, the same set of operations used by ChaCha/Freestyle's quarter-round (QR) (equation 3); hence, are not sensitive to timing attacks by design.

4.2.2. Known-plaintext attacks (KPA), Chosen-plaintext attacks (CPA), and differential cryptanalysis

For a known or chosen plaintext, due to the random behavior of Freestyle, even if the *nonce* is controlled by the adversary, there are N_c possible ciphertexts. Hence, the effort required in cryptanalysis using known plaintext, chosen plaintext, differential analysis increases N_c times.

4.2.3. Chosen-ciphertext attacks (CCA)

In chosen-ciphertext attacks we consider two cases based on the adversary's ability to control the *nonce*.

(i) If nonce cannot be controlled by the adversary. To generate an arbitrary ciphertext, an adversary while initializing the cipher (Section 3.4) has to provide I_h valid *hash*es, and at least one valid *hash* for sending block(s) of ciphertext. As a random round is chosen between [8,32] to initialize the *rand* (equation 16), there are only 25 valid *hash* values possible for a given block. Hence, at the time of decryption, the total possible *hash*es that can be accepted by the receiver for a block of ciphertext is $N_r = \left(\frac{R_{max}-R_{min}}{gcd(R_{max},R_{min})} + 1\right)$. And as there are 256 possible values for *hash*, to send a valid ciphertext, the adversary has to send $(I_h + N_b)$ valid *hash*es. By brute-force approach, the probability of such an event occurring is:

$$= \left(\frac{25}{256}\right)^{I_h} \times \left(\frac{N_r}{256}\right)^{N_b} < \begin{cases} 2^{-23} & \text{for } I_h = 7, \\ 2^{-26} & \text{for } I_h = 8, \\ \vdots & \\ 2^{-67} & \text{for } I_h = 20, \\ \vdots & \\ 2^{-187} & \text{for } I_h = 56 \end{cases} \tag{19}$$

Assuming a constant time cryptographic implementation to check the validity of $(I_h + N_b)$ *hash*es, for $I_h \geq 20$, it is hard in practice to generate an arbitrary ciphertext (CCA) that can be accepted by a receiver if *nonce* cannot be controlled by the adversary.

(ii) If nonce can be controlled by the adversary. In this case, the adversary can launch CPA which can reveal $(I_h + N_b)$ valid *hash*es. And, the adversary can replay them to make the receiver accept arbitrary ciphertext of N_b blocks.

In either of the two cases, after successfully sending a valid ciphertext, the adversary still has to guess the 128-bit *rand* (in case of $I_h = 28$). It is com-

putationally infeasible to compute which combination of *key* and *rand* the I_h *hash*es map to.

Remark 4 It must be noted that Freestyle's hash function does not use *plaintext/ciphertext* as an input. Hence, cannot prevent ciphertext tampering. In practice, Freestyle like ChaCha must be used with a secure message authentication code (MAC) such as Poly1305 [59].

4.2.4. XOR of ciphertexts when key and nonce are reused

Let us consider two *messages* M_1 and M_2 which when encrypted, produce ciphertexts C_1 and C_2. In the event of *key* and *nonce* being reused, in a deterministic stream cipher, $C_1 \oplus C_2 = M_1 \oplus M_2$. Whereas in Freestyle, for $|M_1|$ and $|M_2| \geq log_2(N_c)$:

$$Pr(C_1 \oplus C_2 = M_1 \oplus M_2) = \frac{1}{N_c} \tag{20}$$

The equation 20 indicates that Freestyle is resistant to key re-installation attacks like KRACK [16]. Also, in existing approaches of ciphertext randomization, in case of *key* and *nonce* being reused, the random bytes that are shared with the receiver are prone to XOR attacks. However, such attacks are not possible in Freestyle, as only *hash*es are sent to the receiver; which are not prone to XOR-based attacks.

4.3. Resisting brute-force and dictionary attacks

Freestyle by design resists brute-force and dictionary attacks by: (i) Restricting pre-computation of stream, and (ii) Wasting adversary's time and computational power.

4.3.1. Restricting pre-computation of keystream

In ChaCha, the *keystream* can be pre-computed for various *key*s if *nonce* is known. Pre-computation of stream is advantageous for a genuine receiver,

as there is no need to wait for the message. However, for an adversary, pre-
355 computation of streams with various *key*s is ideal to perform brute-force and dictionary attacks.

In Freestyle, since the *keystream* depends on the *rand* and *hash*, the exact *keystream* cannot be pre-computed unless the sender sends the entire expected *hash*es. This however, also restricts pre-computation of *keystream* even for a
360 genuine receiver.

4.3.2. Wasting adversary's time and computational resources

For an adversary attempting key-guessing attack, during the cipher initialization, for a given attempt, after executing the R_{min} rounds the attacker checks if the *hash* meets the expected *hash* after every H_I rounds. If the *hash* does not
365 match, the attacker does not know if: (i) the *key* is wrong, or (ii) the *pepper* is wrong, or (iii) the number of rounds is wrong. The only way to confirm that the *key* or *pepper* is wrong is to execute until R_{max} rounds and find that the computed *hash* does not match with the expected *hash*. In case the *hash* matches for a round number in range $[R_{min}, R_{max}]$, the attacker will execute
370 more number of rounds to compute the round number for the next initialization block. This has to be performed until all the I_h number of valid round numbers are found. Thus, paying penalty for each brute-force attempt.

To quantify the penalty paid by an adversary, we had introduced the Key-Guessing Penalty (KGP) in Section 1.1 (equation 1)

375 KGP is the measure of a cipher's resistance to brute-force and dictionary attacks. Based on KGP, a cipher can be classified in to two categories (i) Ciphers with KGP ≤ 1, which are not resistant to brute-force and dictionary attacks; and (ii) KGP > 1, ciphers that are brute-force and dictionary attack resistant. Ciphers with KGP > 1 are useful in scenarios where an adversary
380 has higher computational power (e.g. a powerful multi-core laptop) than the victim's system (e.g. a low powered RFID/IoT device). Such ciphers force the adversary to use a machine that is at least KGP times faster than the victim's system to launch an effective attack.

Remark 5 KGP > 1 is a bare minimum criteria necessary for an algorithm to be brute-force resistant. In the later sections of the paper we will show that in Freestyle, theoretically the KGP can be as high as 10^9. ■

Remark 6 KGP > 1 may also be achieved using delays and CAPTCHAs for each incorrect *key* attempt. However, this is not due to the property of the cipher itself. Also, such techniques are not useful in resisting offline brute-force and dictionary attacks. ■

As mentioned in Section 3.2, Freestyle uses *pepper* to achieve $KGP > 1$. If the sender uses a uniform (P)RNG to generate the *pepper* value, the $\mathbb{E}[pepper]$ will be $2^{(P_b-1)}$; however, for an adversary, since the $hash$es are unlikely to match, would require 2^{P_b} attempts in the worst-case scenario. Hence, the maximum KGP one can expect using a uniform (P)RNG is 2. To improve KGP, the sender must use a right-skewed distribution which is kept secret and need not be shared with the receiver. Please note that: a right-skewed (P)RNG is the one which tends to generate smaller values for *pepper*.

Remark 7 Irrespective of the distribution used to generate the *pepper* and number of rounds for encryption/decryption, the round numbers used to generate the *rand* must be a secure (P)RNG with uniform distribution. ■

Let us consider a scenario where an attacker is attempting a brute-force attack. The probability of an 8-bit $hash$ colliding at the n^{th} trial (i.e. after executing $(R_{min} + (n-1) \times H_I)$ rounds, $\forall n \in [1, N_r]$) when using an incorrect *key* or *pepper*, denoted by $Pr_n(X = 1)$ is given as:

$$Pr_n(X = 1) = \underbrace{\left(\prod_{i=1}^{n-1} \frac{256 - i}{257 - i}\right)}_{(n-1) \text{ failures}} \times \underbrace{\left(\frac{1}{257 - n}\right)}_{1 \text{ success}} \tag{21}$$

Then, the expected number of rounds a genuine user will execute when using an incorrect *pepper* is denoted by $\mathbb{E}[R^+]$ can be computed as given in equation 22. It is to be noted that the genuine user at the time of initialization attempts to find the correct *pepper*; hence, also executes $\mathbb{E}[R^+]$ number of rounds for each incorrect guess of *pepper*.

$$\mathbb{E}[R^+] \approx \underbrace{\sum_{h=1}^{I_h}}_{\substack{\text{For each of} \\ \text{the } I_h \text{ number} \\ \text{of hashes}}} \underbrace{\left(\sum_{n=1}^{N_r} Pr_n(X=1)\right)^{h-1}}_{\substack{\text{the probability that the} \\ \text{previous hash has matched}}} \Bigg[\underbrace{\left(\sum_{n=1}^{N_r} (R_{min} + (n-1)H_I) \times Pr_n(X=1)\right)}_{\substack{\text{Number of rounds } (R_{min} \text{ to } R_{max}) \\ \times \\ \text{probability of hash matching at that round}}}$$

$$+ \underbrace{R_{max} \times \left(1 - \sum_{n=1}^{N_r} Pr_n(X=1)\right)}_{\substack{R_{max} \times \\ \text{probability that hash did not match even after } R_{max} \text{ number of rounds}}} \Bigg] \tag{22}$$

$$\mathbb{E}[R^+] \approx 34\, rounds \tag{23}$$

During the cipher initialization, for a correct *key* and *pepper*, the expected number of rounds a user will execute is $\mathbb{E}[R] = 20$ (i.e. average of 8 and 32). After initialization, R_{min}, R_{max}, and H_I are set to their original values, and while decryption, if the expected number of rounds a genuine user executes is denoted by $\mathbb{E}[R]$. Then, the adversary executes $2^{P_b} \times \mathbb{E}[R^+]$ rounds during the initialization. For an adversary, the probability of getting all I_h valid round numbers from the I_h expected *hash*es, and attempting to decrypt the first block of *message* using an incorrect *key* is (from equation 21):

$$= \left(\sum_{n=1}^{N_r} Pr_n(X=1)\right)^{I_h} < \begin{cases} 2^{-23} & \text{for } I_h = 7, \\ 2^{-26} & \text{for } I_h = 8, \\ \quad\vdots & \\ 2^{-67} & \text{for } I_h = 20, \\ \quad\vdots & \\ 2^{-187} & \text{for } I_h = 56 \end{cases} \tag{24}$$

which is very low for $I_h \geq 20$. On the other hand, a genuine user executes

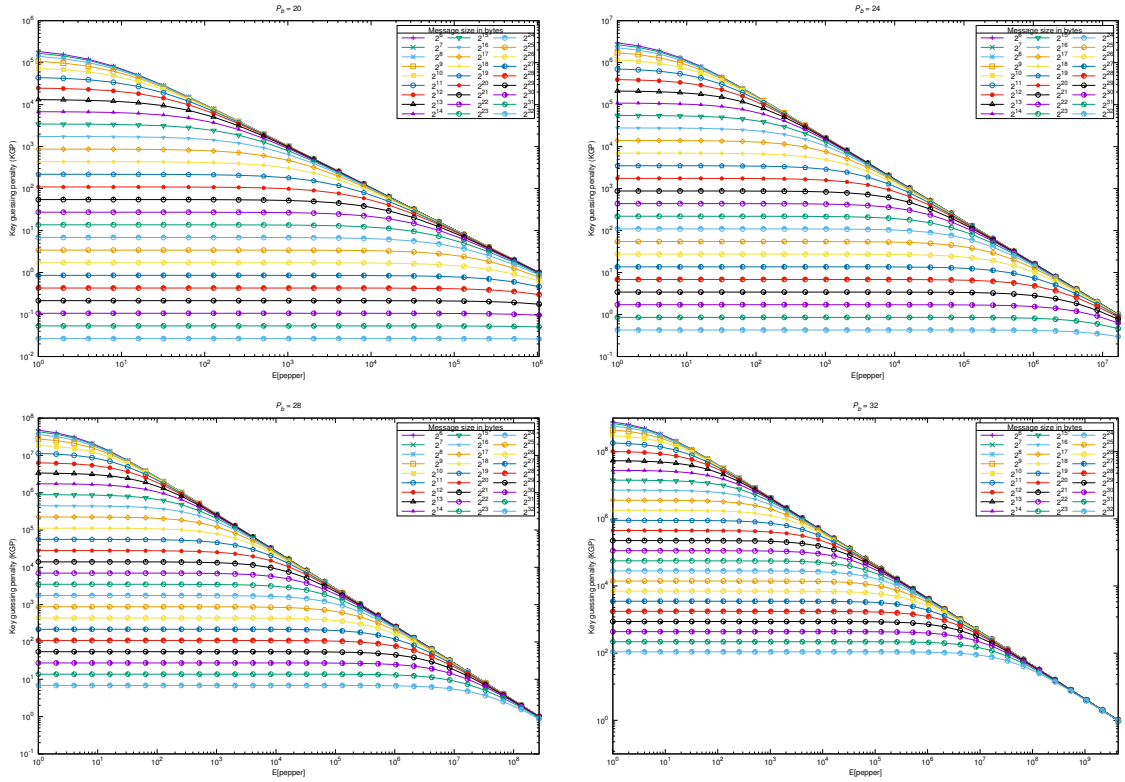

Figure 5: Key guessing penalty (KGP) vs $\mathbb{E}[pepper]$ for $R_{min} = 8, R_{max} = 32, H_I = 1$, $I_h = 7$, $\mathbb{E}[R] = 20$, $P_b \in \{20, 24, 28, 32\}$, and various message sizes (64 bytes to 4 GB).

$\mathbb{E}[pepper] \times \mathbb{E}[R^+]$ rounds during the initialization, and $I_h \times \mathbb{E}[R]$ rounds when using the correct *pepper*, and $N_b \times \mathbb{E}[R]$ rounds to decrypt a *message* of N_b blocks. Then, the KGP using equation 1 is:

$$\text{KGP} \approx \frac{2^{P_b} \times \mathbb{E}[R^+]}{\mathbb{E}[pepper] \times \mathbb{E}[R^+] + I_h \times \mathbb{E}[R] + N_b \times \mathbb{E}[R]} \tag{25}$$

The Figure 5 shows the result of KGP vs. $\mathbb{E}[pepper]$ for $R_{min} = 8, R_{max} = 32, I_h = 7$, $\mathbb{E}[R] = 20$, $P_b \in \{20, 24, 28, 32\}$, and for various message sizes 64 bytes to 4 GB. The results indicate that KGP $\propto \frac{1}{|message|}$; and can be as large as 10^9 by using a right-skewed PRNG for generating the *pepper* value.

To demonstrate the effectiveness of the KGP, we compare the number of

Cipher	Median of brute-force attempts per second
Threefish-256	222222.21
Trivium	222172.85
ChaCha20	220167.32
Speck-256-CTR	203376.04
RC4	195121.95
Simon-256-CTR	164392.56
AES-256-CTR	84509.42
Freestyle ($P_b = 8$)	8157.73
Freestyle ($P_b = 12$)	569.20
Freestyle ($P_b = 16$)	35.79
Freestyle ($P_b = 20$)	**2.24**

Table 2: Comparison of number of brute-force attempts possible for various ciphers and Freestyle ($R_{min} = 8, R_{max} = 32, I_h = 28, P_r = 4$) for various values of P_b, on a single core of i5-6440HQ (2.6 GHz) processor for a 64 byte message.

brute-force attempts per second possible on a single core of i5-6440HQ (2.6 GHz) processor. The results (Table 2) indicate that Freestyle even with the lowest possible values of pepper bits (i.e. $P_b = 8$) and initial hashes (i.e. $I_h = 7$) outperforms most of the commonly used ciphers by a wide margin (atleast by 10 times in our experiments).

Password hashing. To demonstrate the effectiveness of Freestyle for password hashing, we define a simple scheme to hash passwords without using any key-stretching. For example, for the password "*secret*", the initial cipher state is computed as:

$$\begin{bmatrix} constant[0], & constant[1], & constant[2], & constant[3] \\ \text{"secr"}, & \text{"etse"}, & \text{"cret"}, & \text{"secr"} \\ \text{"etse"}, & \text{"cret"}, & \text{"secr"}, & \text{"etse"} \\ \text{0x0}, & \text{"cret"}, & \text{"secr"} & \text{"ets"} \ \mathbf{0x6} \end{bmatrix}$$

Note that the last byte of *nonce* is the length of password (in this case 0x6). Also, in the above scheme, the passwords of length more than 43-bytes are truncated to 43 bytes. The above scheme is used only to report benchmark results for the brute-force attack resistance of plain Freestyle cipher. In real-world applications, Freestyle must be used along with a secure key-stretching algorithm such as PBKDF2 [60], bcrypt [61] based PBKDF, or scrypt [41].

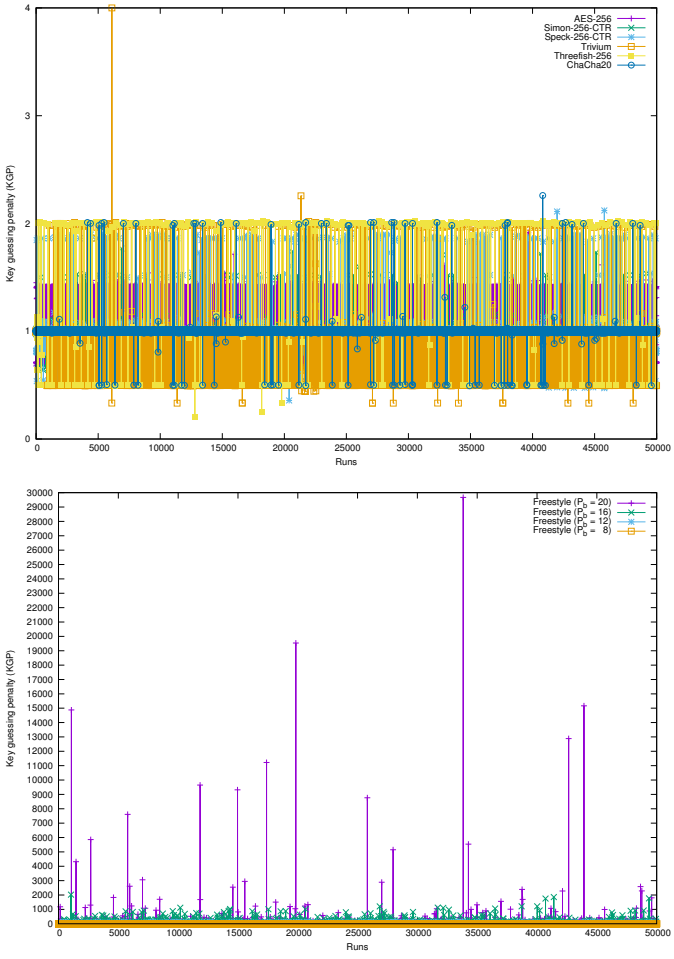

Figure 6: Comparison of KGPs of some the ciphers with Freestyle ($R_{min} = 8, R_{max} = 32, I_h = 28, P_r = 4$) for various values of P_b, on a single core of i5-6440HQ (2.6 GHz) processor.

In the benchmark tests, the password *hash* for Freestyle is computed by encrypting the *salt* with *key* and *nonce* (equation 26, appendix F).

$$password\_hash = encrypt\left(salt, key, nonce\right) \tag{26}$$

And is verified by checking if the plaintext after decrypting the password *hash* is equal to the *salt*.

$$verification = \begin{cases} verified & \text{if } decrypt\left(password\_hash, key, nonce\right) = salt \\ failed & \text{Otherwise} \end{cases} \tag{27}$$

The Table 3 summarizes the median number of brute-force attempts possible on i5-6440HQ (2.6 GHz) processor on John the Ripper's [62] password dataset. The results indicate that in-terms of CPU usage: $Freestyle(R_{min} = 8, R_{max} = 32, P_b = 20, I_h = 28, P_r = 4)$ is comparable to Blowfish-based (bcrypt - \$2b\$, rounds = 12) [61], Argon2d (t = 2, memory cost = 65536) [63], PKCS5_PBKDF2_HMAC_SHA1 (iterations = 65536), scrypt(N = 65536, r = 8, p = 1, buflen = 64) [41], and balloon (s_cost = 4096, t_cost = 1, n_threads = 1) [64].

However, the key difference between Freestyle and other hash algorithms is the KGP. From the results (Figure 7) it can be observed that the most of the hash functions have KGP ≈ 1; whereas Freestyle is able to achieve high KGP even when used with a uniform RNG.

4.3.3. Effect of Zipf's Law on Freestyle

The Zipf's law [66] states that user-generated passwords tend to have a distribution. To understand if the knowledge of the password distribution would make Freestyle less effective, we test it against real world leaked passwords [65] and compare the results with uniformly generated random passwords.

The results (Table 4 and Figure - 8) indicate that Freestyle's KGP and the number of bruteforce attempts per-second do not seem to be affected by Zipf's law.

Hash algorithm	**Number of password hashes per second**
MD5-based (crypt - 1)	7998.20
SHA256-based (crypt - 5)	393.67
SHA512-based (crypt - 6)	493.11
Freestyle-based ($R_{min} = 8$, $R_{max} = 32$, $P_b = 8$, $I_h = 28$, $P_r = 4$)	7370.98
Freestyle-based ($R_{min} = 8$, $R_{max} = 32$, $P_b = 12$, $I_h = 28$, $P_r = 4$)	512.24
Freestyle-based ($R_{min} = 8$, $R_{max} = 32$, $P_b = 16$, $I_h = 28$, $P_r = 4$)	32.24
Freestyle-based ($R_{min} = 8$, $R_{max} = 32$, $P_b = 20$, $I_h = 28$, $P_r = 4$)	2.01
Blowfish-based (bcrypt - $2b$, rounds = 10)	10.25
Blowfish-based (bcrypt - $2b$, rounds = 12)	2.56
Argon2d (t = 2, memory cost = 256)	4530.01
Argon2d (t = 2, memory cost = 4096)	119.20
Argon2d (t = 2, memory cost = 65536)	6.80
PKCS5_PBKDF2_HMAC_SHA1 (iterations = 256)	885.51
PKCS5_PBKDF2_HMAC_SHA1 (iterations = 4096)	55.80
PKCS5_PBKDF2_HMAC_SHA1 (iterations = 65536)	3.51
scrypt (N = 256, r = 8, p = 1, buflen = 64)	1247.72
scrypt (N = 4096, r = 8, p = 1, buflen = 64)	79.96
scrypt (N = 65536, r = 8, p = 1, buflen = 64)	4.92
balloon (s_cost = 256, t_cost = 1, n_threads = 1)	70.08
balloon (s_cost = 1024, t_cost = 1, n_threads = 1)	17.15
balloon (s_cost = 4096, t_cost = 1, n_threads = 1)	4.26
RiffleScrambler (depth = 2, width = 2) (Python version)	1237.62
RiffleScrambler (depth = 2, width = 4) (Python version)	193.05
RiffleScrambler (depth = 2, width = 8) (Python version)	6.28

Table 3: Number of brute-force password attempts possible for various password hash algorithms on a single core of i5-6440HQ (2.6 GHz) processor for John the Ripper's [62] password dataset.

Password dataset	No. of password hashes per second	Password dataset	No. of password hashes per second
alypaa ($P_b = 8$)	7357.53	Carders.cc ($P_b = 8$)	7367.17
alypaa ($P_b = 12$)	514.31	Carders.cc ($P_b = 12$)	514.55
alypaa ($P_b = 16$)	32.45	Carders.cc ($P_b = 16$)	32.50
alypaa ($P_b = 20$)	2.11	Carders.cc ($P_b = 20$)	2.11
Elitehacker ($P_b = 8$)	7375.15	Facebook-Pastebay ($P_b = 8$)	7446.97
Elitehacker ($P_b = 12$)	514.30	Facebook-Pastebay ($P_b = 12$)	516.78
Elitehacker ($P_b = 16$)	32.33	Facebook-Pastebay ($P_b = 16$)	32.70
Elitehacker ($P_b = 20$)	2.11	Facebook-Pastebay ($P_b = 20$)	2.11
Facebook-Phished ($P_b = 8$)	7370.41	Faithwriters ($P_b = 8$)	7383.59
Facebook-Phished ($P_b = 12$)	514.33	Faithwriters ($P_b = 12$)	514.55
Facebook-Phished ($P_b = 16$)	32.48	Faithwriters ($P_b = 16$)	32.45
Facebook-Phished ($P_b = 20$)	2.11	Faithwriters ($P_b = 20$)	2.11
Hak5 ($P_b = 8$)	7389.37	Hotmail ($P_b = 8$)	7371.54
Hak5 ($P_b = 12$)	514.31	Hotmail ($P_b = 12$)	514.29
Hak5 ($P_b = 16$)	32.46	Hotmail ($P_b = 16$)	32.46
Hak5 ($P_b = 20$)	2.11	Hotmail ($P_b = 20$)	2.11
MySpace ($P_b = 8$)	7386.29	phpbb ($P_b = 8$)	7386.21
MySpace ($P_b = 12$)	514.32	phpbb ($P_b = 12$)	514.27
MySpace ($P_b = 16$)	32.47	phpbb ($P_b = 16$)	32.47
MySpace ($P_b = 20$)	2.11	phpbb ($P_b = 20$)	2.10
Unknown-porn-site ($P_b = 8$)	7370.14	Rockyou* ($P_b = 8$)	7386.82
Unknown-porn-site ($P_b = 12$)	514.05	Rockyou* ($P_b = 12$)	514.23
Unknown-porn-site ($P_b = 16$)	32.37	Rockyou* ($P_b = 16$)	32.46
Unknown-porn-site ($P_b = 20$)	2.11	Rockyou* ($P_b = 20$)	2.11
Singles.org ($P_b = 8$)	7421.15	Ultimate-Strip-Club ($P_b = 8$)	7425.72
Singles.org ($P_b = 12$)	514.34	Ultimate-Strip-Club ($P_b = 12$)	514.86
Singles.org ($P_b = 16$)	32.37	Ultimate-Strip-Club ($P_b = 16$)	32.38
Singles.org ($P_b = 20$)	2.02	Ultimate-Strip-Club ($P_b = 20$)	2.02
John the ripper ($P_b = 8$)	7370.98	random-password (50k) ($P_b = 8$)	7382.23
John the ripper ($P_b = 12$)	512.24	random-password (50k) ($P_b = 12$)	513.86
John the ripper ($P_b = 16$)	32.24	random-password (50k) ($P_b = 16$)	32.43
John the ripper ($P_b = 20$)	2.01	random-password (50k) ($P_b = 20$)	2.11

Table 4: Number of brute-force password attempts possible for various password data sets (Leaked passwords [65], John the ripper [62], and random passwords) using Freestyle ($R_{min} = 8, R_{max} = 32, I_h = 28, P_r = 4$) on a single core of i5-6440HQ (2.6 GHz) processor. * For the Rockyou dataset, a subset of 50000 randomly selected passwords were used instead of the entire dataset.

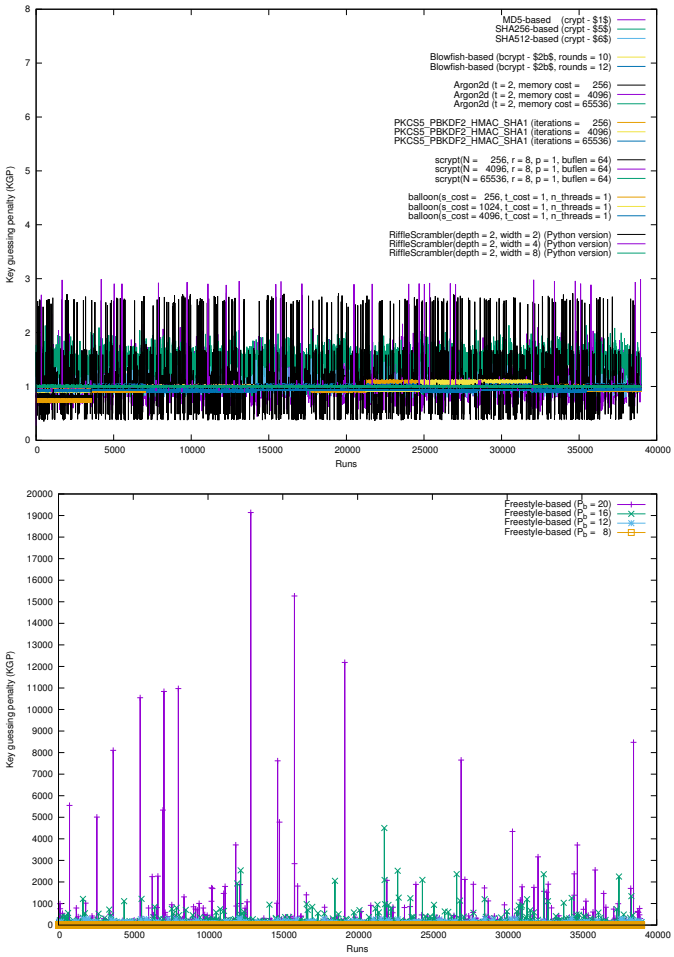

Figure 7: The observed KGP of Freestyle when used as a password hashing function on John the Ripper's [62] password dataset. Note that the PRNG used to generate *pepper* and R is a uniform RNG.

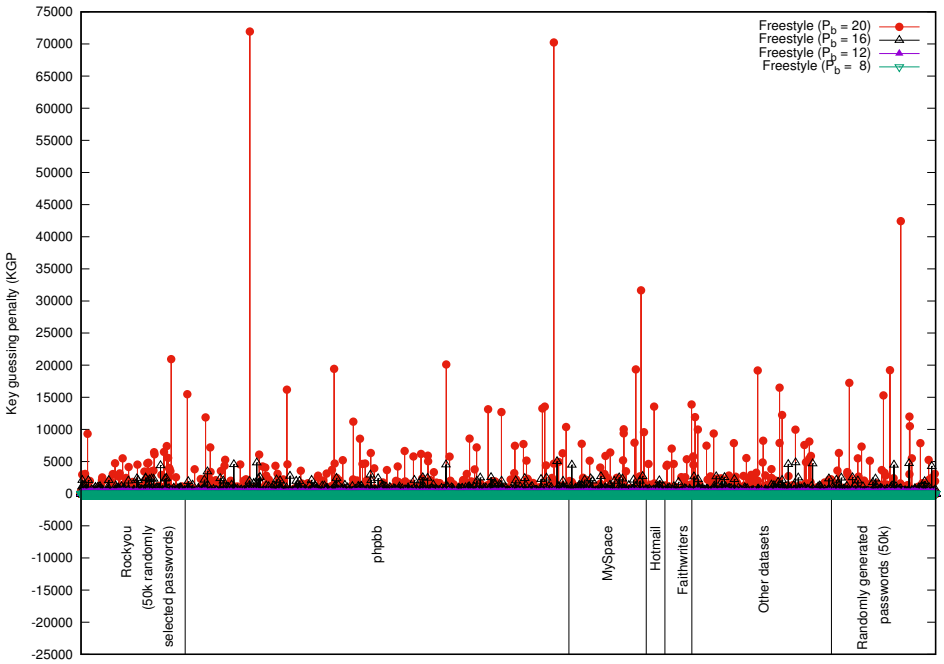

Figure 8: The observed KGP of Freestyle ($R_{min} = 8, R_{max} = 32, I_h = 28, P_r = 4$) when used as a password hashing function on leaked password datasets [65], John the ripper [62], and Random passwords. Note that the PRNG used to generate *pepper* and R is a uniform RNG.

Analytically, it is due to the fact that ChaCha is not known to be vulnerable to known-plaintext attack; i.e. it is not feasible to compute the *key* (or *password*) even if *message* (*salt* in the case of password hashing) or the *keystream* is known to an adversary (equation 5). Furthermore, since Freestyle adds at-least 32-bits of randomness through the *rand* variable (Equation 9 and Figure 4) and performs pre-computation (equations 10, 12, 13), it makes such attacks even harder.

Also, Freestyle and ChaCha use 256-bit *key*s; even though the *key*s may not be uniformly distributed over the 2^{256} state space, the 256-bit *key* is dispersed into a 512-bit *keystream* after running R rounds. Assuming that $R \geq 8$, and the *keystream* is leaked to an adversary, there are no known feasible attacks yet that can take advantage of this fact by just using the *keystream*.

4.4. Better security for smaller keys

Though, not recommended, ChaCha supports 128-bit *key*s by concatenating the *key* with itself to form a 256-bit *key*. In Freestyle, *rand* is used to modify the initial state of cipher to provide an additional $\frac{32 \times I_h}{7}$-bits of secret (in case of $I_h = 28$, 128-bit secret). The *rand* is statistically independent of the *key*, *nonce*, *pepper*, and *message* (equation 6); hence, for applications where 128-bit *key*s have to be used, Freestyle offers better security than ChaCha against brute-force or dictionary attacks.

Also, some applications may have small key-space due to poor (P)RNG. In such cases, Freestyle can resist up to $log_2(KGP)$ bits of *key* being leaked. It is to be noted that the source of (P)RNG to generate *key* may be different from the (P)RNG available at the sender. Here we assume that the (P)RNG at the sender for generating *pepper* and initial I_h round numbers are not leaked.

4.5. Overheads

4.5.1. Computational overhead

Freestyle has two main overheads when compared to ChaCha: (i) Overhead in generating a random number for each block of *message*; (ii) Computation of

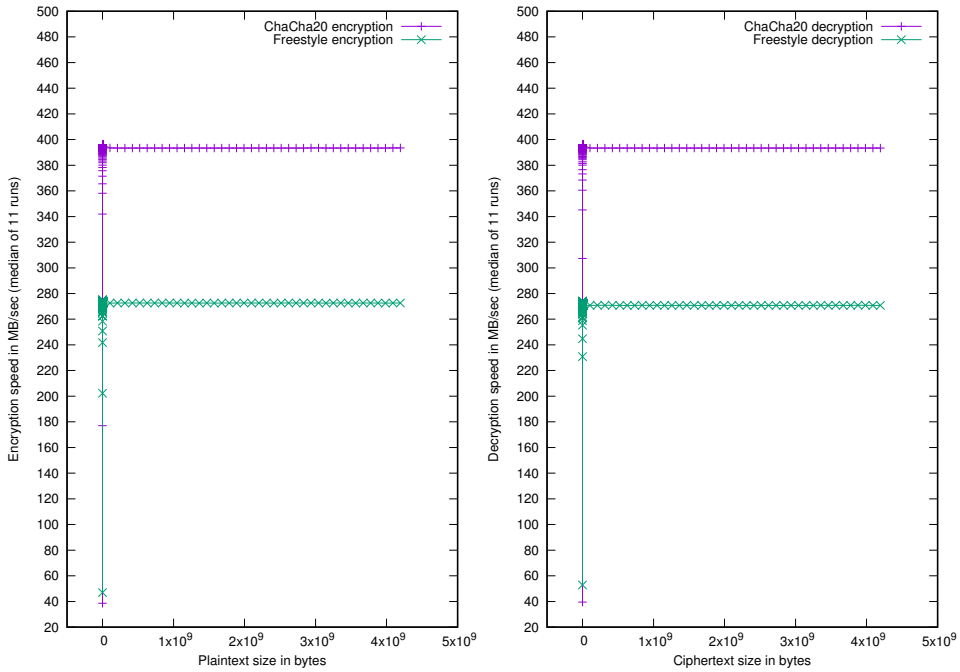

Figure 9: Performance comparison of Freestyle vs. ChaCha20 on a single core of Intel i5-6440HQ (2.6 GHz) processor using `randen` [67] as the PRNG. Note that the result does not account for the time taken for cipher initialization.

a *hash* after every H_I rounds, which uses 1 quarter rounds (QR) of Freestyle. Hence, for encryption the computational overhead is:

$$\begin{aligned} &= \quad time(\text{to generate } N_b \text{ random numbers}) \\ &\quad + \sum_{i=1}^{N_b} \left(\frac{R_i - R_{min}}{gcd(R_{min}, R_{max})} + 1 \right) \times time(1 \text{ QR}) \end{aligned} \tag{28}$$

Note that the equation 28 does not include the time taken for cipher initialization. And the worst case performance overhead is when $R_i = R_{max}, \forall i \in [0, N_b)$. The Figure 9 shows the comparison of performance between optimized versions of Freestyle and ChaCha20[3] and Freestyle[4] with various configurations,

[3] `http://cvsweb.openbsd.org/cgi-bin/cvsweb/src/usr.bin/ssh/chacha.c?rev=1.1`

[4] `https://github.com/arun-babu/freestyle/tree/master/optimized/8-32`

without accounting for the time taken for initialization. The results were obtained by running the benchmarks on a single core of Intel i5-6440HQ (2.6 GHz) processor using `arc4random` [51] (during cipher initialization) and `randen` [67][5] (during encryption) and as the CPRNG. For the performance comparison test, $R_{min} = 8$, $R_{max} = 32$ has been used to make the cipher performance comparable to ChaCha20, as a uniformly distributed random number generator is used. The results indicate that Freestyle could be 1.48 times slower than ChaCha20 (Figure 9). It is to be noted that Freestyle's encryption function is non-deterministic and its performance is dependent on the RNGs performance; whereas the decryption function is deterministic.

4.5.2. Bandwidth overhead

Freestyle algorithm requires a sender to send the final round's 8-bit *hash*; i.e. require sending extra 8 bits for each block of *message* to be sent. Also, for initialization of *rand*, it requires extra $8 \times I_h$ bits. Hence, the total bandwidth overhead in bits is $(8I_h + 8N_b)$, i.e.

$$\text{Bandwidth overhead (in \%)} = \frac{800 \times (I_h + N_b)}{|message|} \approx 1.5625\% \tag{29}$$

Freestyle's bandwidth overhead is low compared to some of the classical randomized encryption techniques [38, 33] and commonly used encoding standards such as Base64.

4.6. Freestyle vs *ChaCha*

When compared to ChaCha, Freestyle offers better security for 128-bit keys (Section 4.4). It also provides the possibility of generating 2^{256} ciphertexts for a given *message* even if *nonce* and *key* is reused (Section 4.1). This makes Freestyle resistant to XOR of ciphertext attacks if *key* and *nonce* is reused. Randomization also makes Freestyle more resistant to KPA, CPA, and CCA (Section 4.2.2). Freestyle offers the possibility of KGP > 1, which makes it

[5] `https://github.com/jedisct1/randen-rng`

resistant to brute-force and dictionary based attacks (Section 4.3). Also, due to the KGP, Freestyle can resist against attacks which can leak upto $log_2(KGP)$ *key* bits. In ChaCha20, $S^{(0)}$ is added with $S^{(20)}$ to generate keystream; which would leak the values of $S^{(20)}[i]$, for $i \in \{0, 1, 2, 3, 12, 13, 14, 15\}$ in case of CPA or CCA attacks. Although, there are no known attacks yet to extract any key bits from the above leaked values; Freestyle is not prone to such leaks as none of the elements of S^* in equation 11 is known to an adversary.

On the other hand, Freestyle could be 1.48 times slower than ChaCha20 (Section 4.5), and also has a higher cost of initialization (sections 3.2, 3.4). In terms of bandwidth overhead, Freestyle generates $\approx 1.5625\%$ larger ciphertext; hence, Freestyle may not be suitable for applications where such an increase in ciphertext size is not acceptable. In implementation overhead, Freestyle's encryption and decryption logic differ slightly. ChaCha is a simple constant time algorithm, whereas Freestyle is a randomized algorithm and adds complexity to make cryptanalysis difficult in practice. Finally, Freestyle assumes that the sender has a cryptographically secure PRNG (CPRNG). A CPRNG may not be available or could be very slow on low powered devices; for such devices, if performance and bandwidth is critical, then Freestyle may not be the right cipher.

4.7. Freestyle vs *Boyen's halting password puzzles [43]*

Freestyle differs from Boyen's work on halting password puzzles [43] in the following ways: (i) The minimum and maximum number of iterations/rounds in Freestyle is explicitly defined and is expected to be public. This step is crucial as it ensures a minimum level of security for genuine user during encryption/decryption. It also ensures that an adversary executes at least the minimum number of iterations. The maximum iterations ensure that a genuine user cannot run more than specified iterations; thus preventing the possibility of DoS attacks or getting stuck in an infinite loop due to human errors; (ii) Freestyle does not require a complex collision resistant hash function, as *hash* collisions are handled simply incrementing the *hash* if a collision occurs. Also,

the hash function uses ARX instructions to resist timing-based cryptanalysis; (iii) In Freestyle, the security of the cipher is not dependent on amount of time taken or number of iterations for cipher initialization, but on the length of *pepper* bits; (iv) Freestyle uses I_h number of 8-bit *hash*es for initialization and an 8-bit *hash* for every block of message being sent, thus the total size of *hash* is not fixed and is $\propto |message|$; (v) Freestyle does not require *hash* computation at every iteration, instead a hash interval (H_I) is used to determine the optimal round intervals at which the *hash* must be computed, thus offering the best performance; and (vi) Freestyle always initializes the cipher with $R_{min} = 8$ and $R_{max} = 32$, thus ensures enough randomness even in cases where user provides insecure parameters for cipher initialization; and (vii) Freestyle offers the possibility of much higher KGP by allowing the sender to choose a right-skewed distribution to generate *pepper* and R_i.

5. Conclusion

In this paper we have introduced Freestyle, a novel randomized cipher capable of generating up to 2^{256} different ciphertexts for a given *key*, *nonce*, and *message*; making key re-installation attacks difficult in practice. Due to its random nature, Freestyle is more resistant to known-plaintext (KPA), chosen-plaintext(CPA) and chosen-ciphertext (CCA) attacks than ChaCha. We have introduced the concepts of bounded *hash based halting condition* and *key-guessing penalty* (KGP), which are helpful in development and analysis of key-guessing attack resistant ciphers and hash algorithms. Freestyle has demonstrated KGP > 1 which makes it run faster on a low-powered machine having the correct *key*, and runs KGP times slower on an adversary's machine with an incorrect *key*. Freestyle is ideal for applications where the ciphertext is assumed to be in full control of the adversary i.e. where an offline brute-force or dictionary attack can be carried out. Example applications include disk encryption, encrypted databases, password managers, sensitive data in public facing IoT devices, etc.

Major limitations of Freestyle include: expectation of a cryptographically secure PRNG with the sender, and a $\approx 1.5625\%$ increase in ciphertext size. Hence, Freestyle may not be the right cipher for applications where performance and bandwidth are more critical than protection against key-guessing attacks.

This paper has introduced a new class of ciphers having the property of KGP > 1; there is further scope for research on other possible and simpler ways to achieve KGP > 1, and study the properties of such ciphers. The possibility of forcing an adversary to solve a NP-hard problem for as many brute-force attempts as possible could be an attractive area of research. However, the key challenge will be to make the time taken to attempt decryption with an incorrect *key*, greater than the time taken to detect if the problem is NP-hard.

Acknowledgments

This work was supported in part by the Bosch Research and Technology Centre - India under the project titled "E-sense - Sensing and Analytics for Energy Aware Smart Campus", and in part by the Robert Bosch Centre for Cyber-Physical Systems, Indian Institute of Science, Bengaluru. The authors thank Sagar Gubbi, Rajesh Sundaresan, Navin Kashyap, and Sanjit Chatterjee for helpful discussions.

References

[1] D. J. Bernstein, ChaCha, a variant of Salsa20, in: Workshop Record of SASC, Vol. 8, 2008, pp. 3–5.

[2] A. Langley, W. Chang, N. Mavrogiannopoulos, J. Strombergson, S. Josefsson, ChaCha20-Poly1305 cipher suites for transport layer security (TLS), Tech. rep. (2016).

[3] D. Miller, S. Josefsson, The chacha20-poly1305@openssh.com authenticated encryption cipher draft-josefsson-ssh-chacha20-poly1305-openssh-00,

network Working Group Internet-Draft, https://tools.ietf.org/html/draft-josefsson-ssh-chacha20-poly1305-openssh-00, Last accessed 1.12.2018 (2018).

[4] D. Miller, chacha20poly1305 protocol, https://cvsweb.openbsd.org/cgi-bin/cvsweb/src/usr.bin/ssh/PROTOCOL.chacha20poly1305?annotate=HEAD, Last accessed 1.12.2018 (2018).

[5] eBACS: ECRYPT Benchmarking of Cryptographic Systems, https://bench.cr.yp.to/results-stream.html Last accessed 10.5.2018 (2017).

[6] E. Bursztein, Speeding up and strengthening HTTPS connections for Chrome on Android, google security blog, https://security.googleblog.com/2014/04/speeding-up-and-strengthening-https.html (Apr. 2014).

[7] Vulnerability Note VU#307015, Infineon RSA library does not properly generate RSA key pairs, cVE-2017-15361, https://www.kb.cert.org/vuls/id/307015 (Oct 2017).

[8] S. H. Kim, D. Han, D. H. Lee, Predictability of Android OpenSSL's pseudo random number generator, in: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, ACM, 2013, pp. 659–668.

[9] L. Bello, M. Bertacchini, B. Hat, Predictable PRNG in the vulnerable Debian OpenSSL package: the what and the how, in: the 2nd DEF CON Hacking Conference, 2008.

[10] S. Yilek, E. Rescorla, H. Shacham, B. Enright, S. Savage, When private keys are public: Results from the 2008 Debian OpenSSL vulnerability, in: Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference, ACM, 2009, pp. 15–27.

[11] A. Lenstra, J. P. Hughes, M. Augier, J. W. Bos, T. Kleinjung, C. Wachter, Ron was wrong, Whit is right, Tech. rep., IACR (2012).

[12] N. Heninger, Z. Durumeric, E. Wustrow, J. A. Halderman, Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices., in: USENIX Security Symposium, Vol. 8, 2012.

[13] E. N. Lorente, C. Meijer, R. Verdult, Scrutinizing WPA2 Password Generating Algorithms in Wireless Routers., in: WOOT, 2015.

[14] A. Ruddick, J. Yan, Acceleration attacks on PBKDF2: or, what is inside the black-box of oclHashcat?, in: WOOT, 2016.

[15] A. Visconti, S. Bossi, H. Ragab, A. Calò, On the weaknesses of PBKDF2, in: International Conference on Cryptology and Network Security, Springer, 2015, pp. 119–126.

[16] M. Vanhoef, F. Piessens, Key Reinstallation Attacks: Forcing Nonce Reuse in WPA2, in: Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS), ACM, 2017.

[17] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, J. K. Zinzindohoue, A messy state of the union: Taming the composite state machines of TLS, in: Security and Privacy (SP), 2015 IEEE Symposium on, IEEE, 2015, pp. 535–552.

[18] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, et al., Imperfect forward secrecy: How Diffie-Hellman fails in practice, in: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, ACM, 2015, pp. 5–17.

[19] W. Gu, Y. Huang, R. Qian, Z. Liu, R. Gu, Attacking Crypto-1 Cipher Based on Parallel Computing Using GPU, in: International Conference on Applications and Techniques in Cyber Security and Intelligence, Springer, 2017, pp. 293–303.

[20] G. Agosta, A. Barenghi, G. Pelosi, High speed cipher cracking: the case of Keeloq on CUDA.

[21] V. Chiriaco, A. Franzen, R. Thayil, X. Zhang, Finding partial hash collisions by brute force parallel programming, in: Systems, Applications and Technology Conference (LISAT), 2017 IEEE Long Island, IEEE, 2017, pp. 1–6.

[22] F. Wiemer, R. Zimmermann, High-speed implementation of bcrypt password search using special-purpose hardware, in: ReConFigurable Computing and FPGAs (ReConFig), 2014 International Conference on, IEEE, 2014, pp. 1–6.

[23] K. Malvoni, D. Solar, J. Knezović, Are your passwords safe: Energy-efficient bcrypt cracking with low-cost parallel hardware, in: WOOT'14 8th Usenix Workshop on Offensive Technologies Proceedings 23rd USENIX Security Symposium, 2014.

[24] Z. Liu, J. Großschädl, Z. Hu, K. Järvinen, H. Wang, I. Verbauwhede, Elliptic curve cryptography with efficiently computable endomorphisms and its hardware implementations for the internet of things, IEEE Transactions on Computers 66 (5) (2017) 773–785.

[25] K. Javeed, X. Wang, M. Scott, High performance hardware support for elliptic curve cryptography over general prime field, Microprocessors and Microsystems.

[26] A. Khalid, G. Paul, A. Chattopadhyay, RC4-AccSuite: A Hardware Acceleration Suite for RC4-Like Stream Ciphers, IEEE Transactions on Very Large Scale Integration (VLSI) Systems 25 (3) (2017) 1072–1084.

[27] F. K. Gürkaynak, R. Schilling, M. Muehlberghuber, F. Conti, S. Mangard, L. Benini, Multi-core data analytics SoC with a flexible 1.76 Gbit/s AES-XTS cryptographic accelerator in 65 nm CMOS, in: Proceedings of the

Fourth Workshop on Cryptography and Security in Computing Systems, ACM, 2017, pp. 19–24.

[28] D. Reis, M. Niemier, X. S. Hu, Computing in memory with fefets, in: Proceedings of the International Symposium on Low Power Electronics and Design, ACM, 2018, p. 24.

[29] W. Kim, A. Chattopadhyay, A. Siemon, E. Linn, R. Waser, V. Rana, Multistate memristive tantalum oxide devices for ternary arithmetic, Scientific reports 6.

[30] S. Jain, A. Ranjan, K. Roy, A. Raghunathan, Computing in Memory with Spin-Transfer Torque Magnetic RAM, arXiv preprint arXiv:1703.02118.

[31] A. Sebastian, T. Tuma, N. Papandreou, M. L. Gallo, L. Kull, T. Parnell, E. Eleftheriou, Temporal correlation detection using computational phase-change memory, Nature Communications`doi:10.1038/s41467-017-01481-9`.

[32] W. Buchanan, When Slow Is Good - The Great Slowcoach: Bcrypt, https://www.linkedin.com/pulse/when-slow-good-great-slowcoach-bcrypt-william-buchanan (Jul. 2015).

[33] R. L. Rivest, A. T. Sherman, Randomized encryption techniques, in: Advances in Cryptology, Springer, 1983, pp. 145–163.

[34] S. Goldwasser, S. Micali, Probabilistic encryption, Journal of computer and system sciences 28 (2) (1984) 270–299.

[35] T. ElGamal, A public key cryptosystem and a signature scheme based on discrete logarithms, IEEE transactions on information theory 31 (4) (1985) 469–472.

[36] R. Cramer, V. Shoup, A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack, in: Annual International Cryptology Conference, Springer, 1998, pp. 13–25.

[37] S. Papadimitriou, T. Bountis, S. Mavroudi, A. Bezerianos, A probabilistic symmetric encryption scheme for very fast secure communication based on chaotic systems of difference equations, International Journal of Bifurcation and Chaos 11 (12) (2001) 3107–3115.

[38] S. Li, X. Mou, B. L. Yang, Z. Ji, J. Zhang, Problems with a probabilistic encryption scheme based on chaotic systems, International Journal of Bifurcation and Chaos 13 (10) (2003) 3063–3077.

[39] J. Kelsey, B. Schneier, C. Hall, D. Wagner, Secure applications of low-entropy keys, in: International Workshop on Information Security, Springer, 1997, pp. 121–134.

[40] N. Provos, D. Mazieres, Bcrypt algorithm, USENIX, 1999.

[41] C. Percival, S. Josefsson, The scrypt password-based key derivation function, Tech. rep. (2016).

[42] C. Forler, S. Lucks, J. Wenzel, Catena: A memory-consuming password-scrambling framework, Tech. rep., Citeseer (2013).

[43] X. Boyen, Halting password puzzles, in: Proc. Usenix Security, 2007.

[44] S. Nakamoto, Bitcoin: A peer-to-peer electronic cash system.

[45] D. J. Bernstein, Salsa20 specification, eSTREAM Project algorithm description, http://www. ecrypt. eu. org/stream/salsa20pf. html.

[46] D. J. Bernstein, The Salsa20 family of stream ciphers, Lecture Notes in Computer Science 4986 (2008) 84–97.

[47] A. R. Choudhuri, S. Maitra, Differential Cryptanalysis of Salsa and ChaCha-An Evaluation with a Hybrid Model., IACR Cryptology ePrint Archive 2016 (2016) 377.

[48] Y. Nir, A. Langley, ChaCha20 and Poly1305 for IETF Protocols, Tech. rep. (2015).

[49] F. Denis, The XChaCha20-Poly1305 construction, https://download.libsodium.org/doc/secret-key_cryptography/xchacha20-poly1305_construction.html (2018).

[50] Libsodium v1.0.12 and v1.0.13 Security Assessment, Tech. rep., https://www.privateinternetaccess.com/blog/wp-content/uploads/2017/08/libsodium.pdf (2017).

[51] T. De Raadt, arc4random - randomization for all occasions (2014). URL `http://www.openbsd.org/papers/hackfest2014-arc4random/index.html`

[52] G. Kedem, Y. Ishihara, Brute force attack on UNIX passwords with SIMD computer.

[53] T. Ylonen, C. Lonvick, The secure shell (ssh) transport layer protocol, rfc 4253 (2006).

[54] S. Maitra, Chosen IV cryptanalysis on reduced round ChaCha and Salsa, Discrete Applied Mathematics 208 (2016) 88–97.

[55] J.-P. Aumasson, S. Fischer, S. Khazaei, W. Meier, C. Rechberger, New features of Latin dances: analysis of Salsa, ChaCha, and Rumba, in: International Workshop on Fast Software Encryption, Springer, 2008, pp. 470–488.

[56] Security Analysis of ChaCha20-Poly1305 AEAD, Tech. rep., KDDI Research, Inc., CRYPTREC-EX-2601-2016, http://www.cryptrec.go.jp/estimation/cryptrec-ex-2601-2016.pdf (Feb 2017).

[57] G. Procter, A Security Analysis of the Composition of ChaCha20 and Poly1305., IACR Cryptology ePrint Archive 2014 (2014) 613.

[58] T. Ishiguro, Modified version of" Latin Dances Revisited: New Analytic Results of Salsa20 and ChaCha"., IACR Cryptology ePrint Archive 2012 (2012) 65.

[59] D. J. Bernstein, The Poly1305-AES Message-Authentication Code., in: FSE, Vol. 3557, Springer, 2005, pp. 32–49.

[60] K. Moriarty, B. Kaliski, A. Rusch, Pkcs# 5: password-based cryptography specification version 2.1, Tech. rep. (2017).

[61] N. Provos, D. Mazieres, A future-adaptable password scheme., in: USENIX Annual Technical Conference, FREENIX Track, 1999, pp. 81–91.

[62] S. Designer, John the ripper wordlists (2011).
URL `https://www.openwall.com/passwords/wordlists/password-2011.lst`

[63] A. Biryukov, D. Dinu, D. Khovratovich, Argon2: new generation of memory-hard functions for password hashing and other applications, in: 2016 IEEE European Symposium on Security and Privacy (EuroS&P), IEEE, 2016, pp. 292–302.

[64] D. Boneh, H. Corrigan-Gibbs, S. Schechter, Balloon hashing: A memory-hard function providing provable protection against sequential attacks, in: International Conference on the Theory and Application of Cryptology and Information Security, Springer, 2016, pp. 220–248.

[65] R. Bowes, Skullsecurity blog, passwords page (2012).
URL `https://wiki.skullsecurity.org/Passwords`

[66] D. Wang, H. Cheng, P. Wang, X. Huang, G. Jian, Zipf's law in passwords, IEEE Transactions on Information Forensics and Security 12 (11) (2017) 2776–2791.

[67] J. Wassenberg, R. Obryk, J. Alakuijala, E. Mogenet, Randen-fast backtracking-resistant random generator with aes+ feistel+ reverie, arXiv preprint arXiv:1810.02227.

Appendix A Reference code - Freestyle Hash function

```
#define AXR(a,b,c,r) {a = PLUS(a,b); c = ROTATE(XOR(c,a),r);}

static u8 freestyle_hash (
        const   u32     cipher_state[16],
        const   u8      previous_hash,
        const   u8      rounds)
{
        u8  hash;

        u32 temp1 = rounds;
        u32 temp2 = previous_hash;

        AXR (temp1, cipher_state[ 3], temp2, 16);
        AXR (temp2, cipher_state[ 6], temp1, 12);
        AXR (temp1, cipher_state[ 9], temp2,  8);
        AXR (temp2, cipher_state[12], temp1,  7);

        hash = temp1 & 0xFF;

        return hash;
}
```

Appendix B Reference code - Initialization for Encryption

```
#define MAX_INIT_HASHES (56)

static void freestyle_randomsetup_encrypt (freestyle_ctx *x)
{
        u32     i;

        u8      R [MAX_INIT_HASHES]; /* actual random rounds */
        u8      CR[MAX_INIT_HASHES]; /* collided random rounds */
```

```
805         u32     temp1;
            u32     temp2;

            const u8 saved_min_rounds               = x->min_rounds;
            const u8 saved_max_rounds               = x->max_rounds;
810         const u8 saved_hash_interval            = x->hash_interval;
            const u8 saved_num_precomputed_rounds   = x->num_precomputed_rounds;

            u32 p;

815         if (! x->is_pepper_set)
            {
                    if (x->pepper_bits == 32)
                            x->pepper = arc4random_uniform (UINT32_MAX);
                    else
820                         x->pepper = arc4random_uniform (
                                    1 << x->pepper_bits
                            );
            }

825         /* set sane values for initalization */
            x->min_rounds                   = 8;
            x->max_rounds                   = 32;
            x->hash_interval                = 1;
            x->num_precomputed_rounds       = 4;
830
            for (i = 0; i < MAX_INIT_HASHES; ++i) {
                    R [i] = CR[i] = 0;
            }

835         /* initial pre-computed rounds */
            freestyle_precompute_rounds(x);
```

```
        /* add a random/user-set pepper to constant[0] */
        x->input[CONSTANT0] = PLUS(x->input[CONSTANT0], x->pepper);

        for (i = 0; i < x->num_init_hashes; ++i)
        {
                R[i] = freestyle_encrypt_block (
                        x,
                        NULL,
                        NULL,
                        0,
                        &x->init_hash [i]
                );

                freestyle_increment_counter(x);
        }

        if (! x->is_pepper_set)
        {
                /* set constant[0] back to its previous value */
                x->input[CONSTANT0] = MINUS(x->input[CONSTANT0], x->pepper);

                /* check for any collisions between 0 and pepper */
                for (p = 0; p < x->pepper; ++p)
                {
                        x->input[COUNTER] = x->initial_counter;

                        for (i = 0; i < x->num_init_hashes; ++i)
                        {
                                CR[i] = freestyle_decrypt_block (
                                        x,
                                        NULL,
                                        NULL,
```

```
                                        0,
                                        &x->init_hash [i]
                                );

                                if (CR[i] == 0) {
                                        goto retry;
                                }

                                freestyle_increment_counter(x);
                        }

                        /* found a collision. use the collided rounds */
                        memcpy(R, CR, sizeof(R));
                        break;

retry:
                        x->input[CONSTANT0] = PLUSONE(x->input[CONSTANT0]);
                }
        }

        for (i = 0; i < 8; ++i)
        {
                temp1 = 0;
                temp2 = 0;

                AXR (temp1, R[7*i + 0], temp2, 16);
                AXR (temp2, R[7*i + 1], temp1, 12);
                AXR (temp1, R[7*i + 2], temp2,  8);
                AXR (temp2, R[7*i + 3], temp1,  7);

                AXR (temp1, R[7*i + 4], temp2, 16);
                AXR (temp2, R[7*i + 5], temp1, 12);
                AXR (temp1, R[7*i + 6], temp2,  8);
```

```
                AXR (temp2, R[7*i + 0], temp1,  7);

                x->rand[i] = temp1;
        }

        /* set user parameters back */
        x->min_rounds                   = saved_min_rounds;
        x->max_rounds                   = saved_max_rounds;
        x->hash_interval                = saved_hash_interval;
        x->num_precomputed_rounds       = saved_num_precomputed_rounds;

        /* set counter to the value that was after pre-computed rounds */
        x->input[COUNTER] = x->initial_counter;

        /* modify constant[1], constant[2], and constant[3] */
        x->input[CONSTANT1] ^= x->rand[1];
        x->input[CONSTANT2] ^= x->rand[2];
        x->input[CONSTANT3] ^= x->rand[3];

        /* modify key[0], key[1], key[2], and key[3] */
        x->input[KEY0] ^= x->rand[4];
        x->input[KEY1] ^= x->rand[5];
        x->input[KEY2] ^= x->rand[6];
        x->input[KEY3] ^= x->rand[7];

        /* Do pre-computation as specified by the user */
        freestyle_precompute_rounds(x);
}
```

Appendix C Reference code - Initialization for Decryption

```
static bool freestyle_randomsetup_decrypt (freestyle_ctx *x)
{
        u32     i;
```

```
935
        u8      R [MAX_INIT_HASHES]; /* random rounds */

        u32     temp1;
        u32     temp2;
940
        const u8 saved_min_rounds               = x->min_rounds;
        const u8 saved_max_rounds               = x->max_rounds;
        const u8 saved_hash_interval            = x->hash_interval;
        const u8 saved_num_precomputed_rounds   = x->num_precomputed_rounds;
945
        u32 pepper;
        u32 max_pepper = x->pepper_bits == 32 ?
                                UINT32_MAX : (u32) ((1 << x->pepper_bits) - 1);

950     bool found_pepper = false;

        /* set sane values for initalization */
        x->min_rounds                   = 8;
        x->max_rounds                   = 32;
955     x->hash_interval                = 1;
        x->num_precomputed_rounds       = 4;

        for (i = 0; i < MAX_INIT_HASHES; ++i) {
                R[i] = 0;
960     }

        /* initial pre-computed rounds */
        freestyle_precompute_rounds(x);

965     /* if initial pepper is set, then add it to constant[0] */
        x->input [CONSTANT0] = PLUS(x->input[CONSTANT0], x->pepper);
```

```
        for (pepper = x->pepper; pepper <= max_pepper; ++pepper)
        {
970             x->input[COUNTER] = x->initial_counter;

                for (i = 0; i < x->num_init_hashes; ++i)
                {
                        R[i] = freestyle_decrypt_block (
975                             x,
                                NULL,
                                NULL,
                                0,
                                &x->init_hash [i]
980                     );

                        if (R[i] == 0) {
                                goto retry;
                        }
985
                        freestyle_increment_counter(x);
                }

                /* found all valid R[i]s */
990             found_pepper = true;
                break;

retry:
                x->input[CONSTANT0] = PLUSONE(x->input[CONSTANT0]);
995     }

        if (! found_pepper)
                return false;

1000    for (i = 0; i < 8; ++i)
```

```
{
        temp1 = 0;
        temp2 = 0;

        AXR (temp1, R[7*i + 0], temp2, 16);
        AXR (temp2, R[7*i + 1], temp1, 12);
        AXR (temp1, R[7*i + 2], temp2,  8);
        AXR (temp2, R[7*i + 3], temp1,  7);

        AXR (temp1, R[7*i + 4], temp2, 16);
        AXR (temp2, R[7*i + 5], temp1, 12);
        AXR (temp1, R[7*i + 6], temp2,  8);
        AXR (temp2, R[7*i + 0], temp1,  7);

        x->rand[i] = temp1;
}

/* set user parameters back */
x->min_rounds                   = saved_min_rounds;
x->max_rounds                   = saved_max_rounds;
x->hash_interval                = saved_hash_interval;
x->num_precomputed_rounds       = saved_num_precomputed_rounds;

/* set counter to the value that was after pre-computed rounds */
x->input[COUNTER] = x->initial_counter;

/* modify constant[1], constant[2], and constant[3] */
x->input[CONSTANT1] ^= x->rand[1];
x->input[CONSTANT2] ^= x->rand[2];
x->input[CONSTANT3] ^= x->rand[3];

/* modify key[0], key[1], key[2], and key[3] */
x->input[KEY0] ^= x->rand[4];
```

```
        x->input[KEY1] ^= x->rand[5];
        x->input[KEY2] ^= x->rand[6];
        x->input[KEY3] ^= x->rand[7];

        /* Do pre-computation as specified by the user */
        freestyle_precompute_rounds(x);

        return true;
}
```

Appendix D Reference code - Encryption and Decryption

```
#define freestyle_encrypt(...) freestyle_xcrypt(__VA_ARGS__,true)
#define freestyle_decrypt(...) freestyle_xcrypt(__VA_ARGS__,false)

int freestyle_xcrypt (
                freestyle_ctx   *x,
        const   u8              *plaintext,
                u8              *ciphertext,
                u32             bytes,
                u8              *hash,
        const   bool            do_encryption)
{
        u32     i       = 0;
        u32     block   = 0;

        while (bytes > 0)
        {
            u8 bytes_to_process = bytes >= 64 ? 64 : bytes;

            u8 num_rounds = freestyle_xcrypt_block (
                x,
                plaintext  + i,
                ciphertext + i,
```

```
                bytes_to_process,
                &hash[block],
                do_encryption
            );

            if (num_rounds < x->min_rounds) {
                return -1;
            }

            i     += bytes_to_process;
            bytes -= bytes_to_process;

            ++block;

            freestyle_increment_counter(x);
        }

        return 0;
}
```

Appendix E Reference code - Encrypt/Decrypt a block of message

```
#define MAX_HASH_VALUES (256)

static u8 freestyle_xcrypt_block (
                freestyle_ctx   *x,
        const   u8              *plaintext,
                u8              *ciphertext,
                u8              bytes,
                u8              *expected_hash,
        const   bool            do_encryption)
{
        u32     i;
```

```
        u8      r;
        u8      hash = 0;

        u32     output[16];

        bool init = (plaintext == NULL) || (ciphertext == NULL);

        u8 rounds = do_encryption ?
                        freestyle_random_round_number (x): x->max_rounds;

        bool do_decryption = ! do_encryption;

        bool hash_collided [MAX_HASH_VALUES];

        memset (hash_collided, false, sizeof(hash_collided));

        for (i = 0; i < 16; ++i) {
                output [i] = x->input [i];
        }

        /* modify counter */
        output[COUNTER] ^= x->rand[0];

        for (r = x->num_precomputed_rounds + 1; r <= rounds; ++r)
        {
                if (r & 1)
                        freestyle_column_round   (output);
                else
                        freestyle_diagonal_round (output);

                if (r >= x->min_rounds && r % x->hash_interval == 0)
                {
                        hash = freestyle_hash (output,hash,r);
```

```
                while (hash_collided [hash]) {
                                ++hash;
                        }

                        hash_collided [hash] = true;

                        if (do_decryption && hash == *expected_hash) {
                                break;
                        }
                }
        }

        if (do_encryption)
                *expected_hash = hash;
        else
                if (r > x->max_rounds)
                        return 0;

        if (! init)
        {
                u8 keystream [64];

                for (i = 0; i < 16; ++i)
                {
                        output [i] = PLUS(output[i], x->input[i]);
                        U32TO8_LITTLE (keystream + 4 * i, output[i]);
                }

                for (i = 0; i < bytes; ++i) {
                        ciphertext [i] = plaintext[i] ^ keystream[i];
                }
        }
```

```
1165            return do_encryption ? rounds : r;
    }
```

Appendix F Reference code - Password hashing

```
    void freestyle_hash_password (
            const   char            *password,
1170            const   u8              *salt,
                    u8              *hash,
            const   size_t          hash_len,
            const   u8              min_rounds,
            const   u8              max_rounds,
1175            const   u8              num_precomputed_rounds,
            const   u8              pepper_bits,
            const   u8              num_init_hashes)
    {
            int i,j;
1180
            freestyle_ctx   x;

            /* salt is 'hash_len' bytes long */
            const u8        *plaintext      = salt;
1185            u8              *ciphertext     = NULL;

            u8 key_and_iv [44];

            u8 expected_hash;
1190
            int password_len = strlen (password);

            assert (password_len    >=  1);
            assert (password_len    <= 43);
1195            assert (hash_len        <= 64);
```

```
if (! (ciphertext = malloc(hash_len)))
{
        perror("malloc failed ");
        exit(-1);
}

/* Fill the key (32 bytes)
   and IV (first 11 bytes) with password */
for (i = 0; i < 43; )
{
        for (j = 0; i < 43 && j < password_len; ++j)
        {
                key_and_iv [i++] = (u8) password[j];
        }
}

// last byte of IV is the password length
key_and_iv [43] = password_len;

u8 *key = key_and_iv;
u8 *iv  = key_and_iv + 32;

freestyle_init_encrypt (
        &x,
        key,
        256,
        iv,
        min_rounds,
        max_rounds,
        num_precomputed_rounds,
        pepper_bits,
        num_init_hashes
```

```
        );

        freestyle_encrypt (
                &x,
                plaintext,
                ciphertext,
                hash_len,
                &expected_hash
        );

        // 'hash' is (num_init_hashes + 1 + hash_len) long

        memcpy (
                hash,
                x.init_hash,
                num_init_hashes
        );

        memcpy (
                hash + num_init_hashes,
                &expected_hash,
                1
        );

        memcpy (
                hash + num_init_hashes + 1,
                ciphertext,
                hash_len
        );
}

bool freestyle_verify_password_hash (
        const   char            *password,
```

```
                const   u8              *salt,
                        u8              *hash,
                const   size_t          hash_len,
1265            const   u8              min_rounds,
                const   u8              max_rounds,
                const   u8              num_precomputed_rounds,
                const   u8              pepper_bits,
                const   u8              num_init_hashes)
1270    {
                int i,j;

                freestyle_ctx   x;

1275            const u8        *ciphertext     = hash + num_init_hashes + 1;
                u8              *plaintext      = NULL;

                u8 key_and_iv [44];

1280            u8 expected_hash = hash [num_init_hashes];

                int password_len = strlen (password);

                assert (password_len    <= 43);
1285            assert (hash_len        <= 64);

                if (! (plaintext = malloc(hash_len)))
                {
                        perror("malloc failed ");
1290                    exit(-1);
                }

                /* Fill the key (32 bytes)
                   and IV (first 11 bytes) with password */
```

```
1295        for (i = 0; i < 43; )
            {
                    for (j = 0; i < 43 && j < password_len; ++j)
                    {
                            key_and_iv [i++] = (u8) password[j];
1300                }
            }

            // last byte of IV is the password length
            key_and_iv [43] = password_len;
1305
            u8 *key = key_and_iv;
            u8 *iv  = key_and_iv + 32;

            if (! freestyle_init_decrypt (
1310                    &x,
                        key,
                        256,
                        iv,
                        min_rounds,
1315                    max_rounds,
                        num_precomputed_rounds,
                        pepper_bits,
                        num_init_hashes,
                        hash
1320        ))
            {
                    return false;
            }

1325        freestyle_decrypt (
                    &x,
                    ciphertext,
```

```
                    plaintext,
                    hash_len,
1330                &expected_hash
            );

            return (0 == memcmp(plaintext,salt,hash_len));
    }
```