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

P. ARUN BABU, Robert Bosch Center for Cyber-physical Systems, Indian Institute of Science, India
JITHIN JOSE THOMAS, Department of Electrical and Computer Engineering, Texas A&M University, USA

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, which makes it suitable for both low-powered devices as well as 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.

CCS Concepts: • Security and privacy → Block and stream ciphers; • Theory of computation → Cryptographic primitives;

Additional Key Words and Phrases: Brute-force resistant ciphers, Dictionary based attacks, Key-guessing penalty, Probabilistic encryption, Freestyle, ChaCha

P. Arun Babu and Jithin Jose Thomas. 2018. Freestyle, a randomized version of ChaCha for resisting offline brute-force and dictionary attacks. 9, 4, Article 39 (March 2018), 30 pages. https://doi.org/0000001.0000001

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[Bernstein 2008a]. ChaCha20 (i.e. ChaCha with 20 rounds) is one of the modern, popular (for TLS [Langley et al. 2016] and SSH [Miller 2018; Miller and Josefsson 2018]), and faster symmetric stream cipher on most machines[cha 2017; Bursztein 2014]. 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 [cve 2017; Bello et al. 2008; Heninger et al. 2012; Kim et al. 2013; Lenstra et al. 2012; Yilek et al. 2009]; (ii) weak passwords [Lorente et al. 2015] and poor implementations of password based key derivation [Ruddick and Yan 2016; Visconti et al. 2015]; and, (iii) poor protocol or cryptographic implementations [Adrian et al. 2015; Beurdouche et al. 2015; Vanhoef...
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 [Agosta et al. 2013; Chiriaco et al. 2017; Gu et al. 2017], specialized hardware for cryptography [Gürkaynak et al. 2017; Javeed et al. 2016; Khalid et al. 2017; Liu et al. 2017; Malvoni et al. 2014; Wiemer and Zimmermann 2014], and memories in terms of storage size and in-memory computations [Jain et al. 2017; Kim et al. 2016; Reis et al. 2018; Sebastian et al. 2017].

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 [Buchanan 2015] 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. We introduce the key guessing penalty, which acts as a measure for a cipher’s resistance to key-guessing attacks. 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 [Vanhoef and Piessens 2017] and cryptanalysis by XOR of ciphertexts in the event of the key and nonce being reused.

The 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.

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 5 presents related work; and section 6 concludes the paper.

2 CHACHA CIPHER AND VARIANTS

ChaCha20 [Bernstein 2008a] is a variant of Salsa20 [Bernstein 2005a, 2008b], 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:

$$
\begin{bmatrix}
\text{constant}[0], & \text{constant}[1], & \text{constant}[2], & \text{constant}[3] \\
\text{key}[0], & \text{key}[1], & \text{key}[2], & \text{key}[3] \\
\text{key}[4], & \text{key}[5], & \text{key}[6], & \text{key}[7] \\
\text{counter}[0], & \text{counter}[1], & \text{nonce}[0], & \text{nonce}[1]
\end{bmatrix}
$$
Freestyle, a randomized version of ChaCha for resisting offline brute-force and dictionary attacks

Table 1. List of symbols

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\min$</td>
<td>Indicates the minimum number of rounds to be used for encryption/decryption. $\min \in [1, 255]$</td>
</tr>
<tr>
<td>$\max$</td>
<td>Indicates the maximum number of rounds to be used for encryption/decryption. $\max \in [\min + 1, 255]$</td>
</tr>
<tr>
<td>$R$</td>
<td>Number of rounds used to encrypt the current block of message. $R = \operatorname{random}(\min, \max)$</td>
</tr>
<tr>
<td>$R_i$</td>
<td>Number of rounds used to encrypt i-th block of message. $R_i = \operatorname{random}(\min, \max)$ and $i \geq 0$.</td>
</tr>
<tr>
<td>$f$</td>
<td>The current round number. $f \in [1, R]$</td>
</tr>
<tr>
<td>$h_f(i)$</td>
<td>Freestyle hash function which generates an 8-bit hash from a 144-bit input.</td>
</tr>
<tr>
<td>$H_f$</td>
<td>Round intervals at which an 8-bit hash has to be computed. $H_f$ is set to 1 at cipher initialization; and during encryption/decryption $H_f$ is set to $\gcd(R_{\max}, R_{\min})$.</td>
</tr>
<tr>
<td>$P_b$</td>
<td>The number of pepper bits to be used during cipher initialization. $P_b \in [5, 32]$</td>
</tr>
<tr>
<td>$I_b$</td>
<td>The number of initial hashes/round numbers to be used for cipher initialization. $I_b \in [7, 56]$</td>
</tr>
<tr>
<td>$P_r$</td>
<td>The number of rounds to be pre-computed</td>
</tr>
<tr>
<td>$\operatorname{pepper}$</td>
<td>The number of iterations required for cipher initialization. pepper = random(0, 2(^{k_b} - 1))</td>
</tr>
<tr>
<td>$R_f$</td>
<td>The number of rounds computed using the expected hash and pepper for i-th block of message</td>
</tr>
<tr>
<td>(|)</td>
<td>The expected value of pepper</td>
</tr>
<tr>
<td>$[R]$</td>
<td>The expected number of rounds executed by an user when an incorrect key or pepper is used</td>
</tr>
<tr>
<td>$[\operatorname{e}(R)]$</td>
<td>The expected number of rounds executed by a genuine user to encrypt/decrypt a block of message. $\operatorname{e}(R) = \frac{\max + \min}{2}$</td>
</tr>
<tr>
<td>$e^{(r)}$</td>
<td>The value of $e$ after $r$ rounds of Freestyle. If $e^{(0)}$ is not explicitly defined, then $e^{(0)} = 0$.</td>
</tr>
<tr>
<td>$\lfloor n \rfloor$</td>
<td>$n$-th element of $v$.</td>
</tr>
<tr>
<td>$a \oplus b$</td>
<td>Bit-wise XOR of $a$ and $b$.</td>
</tr>
<tr>
<td>$a | b$</td>
<td>Addition of $a$ and $b$ modulo $2^b$.</td>
</tr>
<tr>
<td>$</td>
<td>v</td>
</tr>
<tr>
<td>$N_b$</td>
<td>The number of blocks in a message. $N_b = \lfloor \frac{</td>
</tr>
<tr>
<td>$\Pr_{\text{RT}}(X = 1)$</td>
<td>The probability of getting a valid round number at the $n$-th trial, when using an incorrect key or pepper.</td>
</tr>
<tr>
<td>$N_f$</td>
<td>The total number of ciphertexts possible for a given key, nonce, and message.</td>
</tr>
<tr>
<td>$N_t$</td>
<td>The number of ways a block of message can be encrypted using random number of rounds ($R$). $N_t = \frac{\max - \min + 1}{R_f}$</td>
</tr>
<tr>
<td>$\operatorname{time}(o)$</td>
<td>The expected time taken to execute the operation $o$.</td>
</tr>
<tr>
<td>$\operatorname{counter}$</td>
<td>The counter in the CTR mode of operation.</td>
</tr>
<tr>
<td>null</td>
<td>An empty string.</td>
</tr>
<tr>
<td>$N$</td>
<td>A random number that is independent of the key, nonce, message, and pepper.</td>
</tr>
</tbody>
</table>

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

\[
\begin{align*}
\text{Odd round} & : QR \{S[0], S[4], S[8], S[12]\} & QR \{S[0], S[5], S[10], S[15]\} \\
\text{Even round} & : QR \{S[1], S[5], S[9], S[13]\} & QR \{S[1], S[6], S[11], S[12]\} \\
\text{QR} \{S[2], S[6], S[10], S[14]\} & : QR \{S[2], S[7], S[8], S[13]\} \\
\text{QR} \{S[3], S[7], S[11], S[15]\} & : QR \{S[3], S[4], S[9], S[14]\}
\end{align*}
\]

(1)

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:

\[
\begin{align*}
a & \leftarrow a \oplus b; \quad d \leftarrow d \oplus a; \quad d \leftarrow d \lll 16; \\
c & \leftarrow c \oplus d; \quad b \leftarrow b \oplus c; \quad b \leftarrow b \lll 12; \\
a & \leftarrow a \oplus b; \quad d \leftarrow d \oplus a; \quad d \leftarrow d \lll 8; \\
c & \leftarrow c \oplus d; \quad b \leftarrow b \oplus c; \quad b \leftarrow b \lll 7.
\end{align*}
\]

(2)
After 20 rounds, the initial state \(S(0)\) is added to the current state \(S(20)\) to generate the final state. The final state is then serialized in the little-endian format to form the 512-bit key-stream, which is then 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 [Choudhuri and Maitra 2016]. ChaCha20 has two main variants: (i) IETF’s version of ChaCha20 [Langley et al. 2016; Nir and Langley 2015] which uses a 32-bit counter (instead of 64-bit) and 96-bit nonce (instead of 64-bit); and (ii) XChaCha20 [Denis 2018], which uses 192-bit nonce (instead of 64-bit), where a randomly generated nonce is considered safe enough [lib 2017]. The larger nonce in XChaCha20 makes the probability of nonce reuse low.

3 THE FREESTYLE CIPHER

The 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 [De Raadt 2014]).

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 \(\geq 8\); 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,
Freestyle, a randomized version of ChaCha for resisting offline brute-force and dictionary attacks

<table>
<thead>
<tr>
<th>$R_{\text{min}}$ (8 bits)</th>
<th>$R_{\text{max}}$ (8 bits)</th>
<th>$P_b$ (6 bits)</th>
<th>$I_h$ (6 bits)</th>
<th>$P_r$ (4 bits)</th>
</tr>
</thead>
</table>

Fig. 1. The 32-bit cipher-parameter ($C_p$)

however it may not be stored along with the hash or ciphertext (i.e. can be forgotten by the sender after use) [Forler et al. 2013; Kedem and Ishihara 1999]. $P_b \geq 16$ 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 $P_r \leq (R_{\text{min}} - 4)$, indicates the number of Freestyle rounds to be pre-computed.

Using the values of $R_{\text{min}}$ and $R_{\text{max}}$, a hash interval denoted by $H_I \in [1, R_{\text{min}}]$ 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 $gcd(R_{\text{min}}, R_{\text{max}})$ during encryption and decryption. For a given $R_{\text{min}}$ and $R_{\text{max}}$, the $gcd(R_{\text{min}}, R_{\text{max}})$ 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 $R_{\text{max}}$ rounds, then either the key, nonce, or one of the parameters provided by the receiver is incorrect.

Remark 1 It must be noted that hashes are computed only after executing $R_{\text{min}}$ rounds. This avoids accidentally terminating after executing fewer than expected number of rounds.

Remark 2 The performance of Freestyle is $\propto \frac{H_I \times P_r}{R_{\text{min}} \times R_{\text{max}} \times P_b \times I_h}$. 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 3) is a 4x4 matrix of 32-bit words consisting of: 128-bit constant, 256-bit key, 32-bit counter, and 96-bit nonce. Unlike ChaCha[Bernstein 2008a], the counter size has been reduced to 32-bit as in practice most of the protocols such as the SSH transport protocol [Ylonen and Lonvick 2006] recommend re-keying after 1GB of data sent/received.

Freestyle’s initial cipher-state is similar to the IETF’s version of ChaCha, except that the constants are modified using the cipher-parameter ($C_p$). $C_p$ is formed by concatenating all the 5 parameters i.e. $R_{\text{min}}, R_{\text{max}}, P_b, I_h, \text{and} P_r$ to generate a unique 32-bit string as shown in the figure 1. The $C_p$ is then XOR-ed with the constant[0] (equation 3). 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.
counter; which results in ADD-XOR-Rotate instructions as shown in figure 3. The number of possible values of

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

(3)

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 \( C_p \):

\[ R_{\text{min}} = 8, \quad R_{\text{max}} = 32, \quad H_I = 1, \quad \text{and} \quad P_r = 4 \]  

(4)

This is done to ensure there is enough entropy even if weaker values of \( R_{\text{min}} \) and \( R_{\text{max}} \) 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 [Adrian et al. 2015].

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 \( [0, 2^{P_r}] \) is generated by the sender and added to \( S^{(4)}[0] \) to form the intermediate cipher-state \( S^* \) as shown in equation 5.

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

(5)

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

The above hashes 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. \( \text{hash}^{(r−1,H_I)} \)).

It must be noted that at this point only \( l_k \) number of hashes are generated, and no encryption is performed yet. The sender then using the \( l_k \) number of random round numbers: \( \{R_0, R_1, ..., R_{l_k-1}\} \), a 256-bit rand is computed using ADD-XOR-Rotate instructions as shown in figure 3. The number of possible values of rand is dependent on the \( l_k \) value.
Freestyle, a randomized version of ChaCha for resisting offline brute-force and dictionary attacks

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

After which, the values of \( R_{min}, R_{max}, H_I \), and \( P_r \) are set back to its original values; 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 7).

\[ S^* \leftarrow (S^*)^{(P_r)} \]
From now on, the new cipher-state ($S^*$) will be used as an input to generate the key-stream 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 ($R$) that is independent of the key, nonce, message, and pepper (unlike randomized-CTR mode where the bytes used to modify the 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 counters 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 (\(R\)) to be XOR-ed with the counter (i.e. \(S^*[12]\)) is \(\text{rand}[0]\). The rationale behind choosing \(\text{rand}[0]\) 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. \(\text{rand}[0]\). The values of \(\text{rand}[i], \forall i \in [1, 7]\) will be 0 in this case. Hence, in the worst-case scenario, the counters are always 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 7, the key-stream for a block of plaintext is computed by running random number of round (\(R_i - P_r\)) on \(S^*\). The \(S_i^*\) is then added to \(S^*\) to generate the keystream (equation 9). 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 8, 9, and 10.

\[
S_i^* = (S^*)^{(R_i - P_r)}
\]  
\[
\text{keystream}_i = S_i^* \oplus S^*
\]  
\[
\text{ciphertext}_i = \text{plaintext}_i \oplus \text{keystream}_i
\]

Where \(R_i = \text{random}(R_{min}, R_{max}), \forall i \in [0, N_b];\) 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 like the sender first computes cipher-parameter \((C_p)\) and sets the following temporary configuration:

\[
R_{min} = 8, \ R_{max} = 32, \ H_f = 1, \text{ and } P_r = 4
\]
and computes $S^{(4)}$ using the key and nonce. Then the receiver iterates pepper from 0 to $(2^{P_b} - 1)$ using equation 5, until $I_h$ number of valid round numbers are found, corresponding to each of the received $I_h$ number of hashes. 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 6. The original values of $R_{\text{min}}, R_{\text{max}}, H_I,$ and $P_r$ provided by the user are restored; and $P_r$ number of rounds are pre-computed (equation 7).

### 3.5 Decryption

Similar to the encryption (section 3.3), non-deterministic CTR mode is used to decrypt all the blocks of message. And, key-stream is generated using $S^*$ (equations 8 and 9). The plaintext is generated by XOR-ing the ciphertext with key-stream (equation 12).

$$\text{plaintext}_i = \text{ciphertext}_i \oplus \text{keystream}_i$$

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 $|\text{message}|$ bits, the message is divided into $N_b = \left\lceil \frac{|\text{message}|}{512} \right\rceil$ blocks. Since, each block can be encrypted with a random number ($R$) of rounds in the range $[R_{\text{min}}, R_{\text{max}}]$; 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:

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

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:

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

From equation 14, as the number of pepper bits, number of initial hashes, 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 [Choudhuri and Maitra 2016]. And the detailed cryptanalysis of ChaCha can be found in [ch-2017; Aumasson et al. 2008; Choudhuri and Maitra 2016; Ishiguro 2012; Maitra 2016; Procter 2014].

#### 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 (appendix E, line 46).
Freestyle, a randomized version of ChaCha for resisting offline brute-force and dictionary attacks

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 ($\text{hash}^{(r-H)}$). 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 hashes are equally spread over 256 buckets, there are likely to be $2^{134}$ collisions.

The hash function uses Add-Rotate-XOR (ARX) operations, the same set of operations used by ChaCha/Freestyle’s quarter-round (QR) (equation 1); 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 hashes, 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 13), there are only 25 valid hash values possible for a given block. Hence, at the time of decryption, the total possible hashes that can be accepted by the receiver for a block of ciphertext is $N_r = \left\lfloor \frac{R_{\text{max}} - R_{\text{min}}}{H_r} + 1 \right\rfloor$. And as there are 256 possible values for hash, to send a valid ciphertext, the adversary has to send ($I_h + N_b$) valid hashes. By brute-force approach, the probability of such an event occurring is:

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

Assuming a constant time cryptographic implementation to check the validity of ($I_h + N_b$) hashes, 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 hashes. 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 computationally infeasible to know which combination of key and rand the $I_h$ hashes 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 message authentication code (MAC) such as Poly1305 [Bernstein 2005b].
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 $|M_2| \geq \log_2(N_c)$:

$$Pr(C_1 \oplus C_2 = M_1 \oplus M_2) = \frac{1}{N_c} \quad (16)$$

The equation 16 indicates that Freestyle is resistant to key re-installation attacks like KRACK [Vanhoef and Piessens 2017]. 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 hashes are sent to the receiver.

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 key-stream. In ChaCha, the key-stream can be pre-computed for various keys 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-computation of streams with various keys is ideal to perform brute-force and dictionary attacks.

In Freestyle, since the key-stream depends on the rand and hash, the exact key-stream cannot be pre-computed unless the sender sends the entire expected hashes. This however, also restricts pre-computation of key-stream even for a 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 $R_{\text{min}}$ rounds the attacker checks if the hash meets the expected hash after every $H_f$ rounds. If the hash does not 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 $R_{\text{max}}$ 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 $[R_{\text{min}}, R_{\text{max}}]$, the attacker will execute more number of rounds to compute the round number for the next block of message. This has to be performed until all the $I_h$ number of valid rounds are found. Thus, paying penalty for each brute-force attempt. To quantify the penalty an adversary has to pay in terms of computational power, we introduce the Key-guessing penalty (KGP) metric.

**Definition : Key-guessing penalty (KGP)** - 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 17).

$$KGP = \frac{\text{time(attempt decryption of a message using an incorrect key)}}{\text{time(decrypt the message using the correct key)}} \quad (17)$$

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 $\leq 1$, 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 has higher computational power (e.g. a powerful multi-core laptop) than the victim’s system (e.g. a
Freestyle, a randomized version of ChaCha for resisting offline brute-force and dictionary attacks

low powered RFID/IoT device). Such ciphers forces 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 to for an algorithm to be brute-force resistant. In the later sections of the paper we will show that in Freestyle, KGP can be as high as $10^9$. ■

**Remark 6** KGP > 1 may also be achieved by using delays and CAPTCHAs for each incorrect key attempt. However, this this 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 $E[pepper]$ will be $2^{(P_R - 1)}$; however, for an adversary, since the hashes are unlikely to match, would require $2^{P_R}$ 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. Not 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, to generate the *rand*, a secure (P)RNG with uniform distribution must be used. ■

The probability of an 8-bit hash colliding at the $n^{th}$ trial when an incorrect key or *pepper* is used (denoted by $Pr_n(X = 1)$) is given as:

$$Pr_n(X = 1) = \prod_{i=1}^{n-1} \frac{256 - i}{257 - i} \times \frac{1}{257 - n}$$

(18)

Then, the expected number of rounds a user with an incorrect key or *pepper* will execute is denoted by $E[R^+]$ can be computed as given in equation 19.

$$E[R^+] = \sum_{h=1}^{I_h} \sum_{n=1}^{N_r} Pr_n(X = 1) \left[ \sum_{n=1}^{N_r} (R_{min} + nH_I) \times Pr_n(X = 1) \right] + R_{max} \times \left[ 1 - \sum_{n=1}^{N_r} Pr_n(X = 1) \right]$$

(19)

$$E[R^+] \approx 34.2727$$

(20)

During the cipher initialization, for a correct key and *pepper*, the expected number of rounds a user will execute is $E[R] = 20$ (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 $E[R]$. Then, the adversary executes $2^{P_R} \times E[R^+]$ rounds during the initialization. For an adversary, the probability of getting all $I_h$ valid round numbers
from the $I_h$ expected hashes, and attempting to decrypt the first block of message using an incorrect key is:

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

which is very low for $I_h \geq 20$. On the other hand, a genuine user executes $\mathbb{E}[\text{pepper}] \times \mathbb{E}[R^+]$ 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 17 is:

$$\text{KGP} = \frac{2^{P_b \times \mathbb{E}[R^+]} \times \mathbb{E}[\text{pepper}] \times \mathbb{E}[R^+] + I_h \times \mathbb{E}[R] + N_b \times \mathbb{E}[R]}{\mathbb{E}[\text{pepper}] \times \mathbb{E}[R^+] + I_h \times \mathbb{E}[R] + N_b \times \mathbb{E}[R]}$$
Freestyle, a randomized version of ChaCha for resisting offline brute-force and dictionary attacks

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 4GB. The results indicate that KGP $\propto \frac{1}{\text{message}}$; and can be as large as $10^9$ by using a right-skewed probability distribution for generating the $pepper$ value.

To demonstrate the effectiveness of the KGP, we compare the number of brute-force attempts per second possible on a single core of i5-6440HQ processor. The results (figure 6) 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.

4.4 Better security for smaller keys

Though, not recommended, ChaCha supports 128-bit keys 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 128-bit random secret (in case of $I_h = 28$). The $rand$ is statistically independent of the key, nonce, pepper, and message (equation 3); hence, for applications where 128-bit keys have to be used, Freestyle offers better security than ChaCha.

Also, some applications may have lower 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 with the sender. Here we assume that the (P)RNG at the sender for generating pepper and initial $I_h$ round numbers are not leaked.
Fig. 7. Performance comparison of Freestyle vs. ChaCha20 on a single core of Intel i5-6440HQ processor using randen and arc4random() as the PRNG. Note that the result does not account for the time taken for cipher initialization.

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

$$
\text{time (to generate } N_b \text{ random numbers)} + \sum_{i=1}^{N_b} \left( \frac{R_i - R_{\text{min}}}{H_I} + 1 \right) \times \text{time (1 QR of Freestyle)}
$$

Note that the equation 23 does not account for the time taken for cipher initialization. And the worst case performance overhead is when $R_i = R_{\text{max}}$, $\forall i$. The figure 7 shows the comparison of performance between optimized versions of Freestyle and ChaCha20\(^1\) and Freestyle\(^2\) with various configurations, without accounting for the time taken for initialization. The results were obtained by running the benchmarks on a single core of Intel i5-6440HQ processor using arc4random[De Raadt 2014] (during cipher initialization) and randen[Wassenberg et al. 2018]\(^3\) (during encryption) and as the CPRNG. For the performance comparison test, $R_{\text{min}} = 8$, $R_{\text{max}} = 32$ has been used to make the cipher performance comparable to ChaCha20, as an uniformly distributed random number generator is used. The results indicate that Freestyle could be 1.13 to 1.60 times slower than ChaCha20 (figure 7). The difference between the

\(^1\)http://cvsweb.openbsd.org/cgi-bin/cvsweb/src/usr.bin/ssh/chacha.c?rev=1.1
\(^2\)https://github.com/arun-babu/freestyle/tree/master/optimized/8-32
\(^3\)https://github.com/jedisct1/randen-rng
Freestyle, a randomized version of ChaCha for resisting offline brute-force and dictionary attacks

performance of encryption and decryption in Freestyle (figure 7) is mostly due to the delay in generating a random round number during encryption.

4.5.2 Bandwidth overhead. Freestyle algorithm requires a sender to send the final round’s 8-bit hash; i.e. requiring to send 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.

$$\text{Bandwidth overhead (in %)} = \frac{800 \times (I_h + N_b)}{|\text{message}|} \approx 1.5625\% \text{ (for a block of message)}$$ (24)

5 RELATED WORK

5.1 Randomized encryption schemes

Use of randomized encryption schemes have been in practice for many years, and a taxonomy of randomized ciphers is presented in [Rivest and Sherman 1983]. Also, some approaches to randomized encryption for public-key cryptography was proposed in [Cramer and Shoup 1998; ElGamal 1985; Goldwasser and Micali 1984]. Approaches based on chaotic systems for probabilistic encryption were also proposed [Papadimitriou et al. 2001]. However, the main concern with some of the existing approaches are high bandwidth expansion factor and computational overhead [Li et al. 2003; Rivest and Sherman 1983].

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 [Rivest and Sherman 1983]. 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 $I_h$ hashes. The initial $I_h$ hashes 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 $\approx 1.5625\%$.

5.2 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 an CPU intensive key-streaching [Kelsey et al. 1997] or key-setup phase [Provos and Mazieres 1999] using a cost-factor. Also approaches that consume large amount of memory have also been proposed [Forler et al. 2013; Percival and Josefsson 2016]. Another related area is use of client puzzles [Boyen 2007] and proof-of-work (e.g. Bitcoin [Nakamoto 2008]) to delay cryptographic operations.

The hash based halting condition described in section 3, on a high-level uses similar principle as the Halting Key Derivation Function (HKDF) proposed in [Boyen 2007]. 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.

Our approach however differs from [Boyen 2007] in the following ways: (i) The minimum and maximum number of iterations 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 ensures 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 any side-channel 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 hashes 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 \( (H_I) \) parameter is used to determine round intervals at which hash must be computed, thus offering flexibility to adjust performance and security; and (vi) Freestyle forces the cipher initialization 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 \( R_i \).

5.3 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 resistant to KPA, CPA, and CCA (section 4.2.2). Freestyle offers the possibility of KGP > 1, which makes it resistant to brute-force and dictionary based attacks (section 4.3). Also, due to the KGP, Freestyle can resist against attacks which can leak up to \( \log_2(\text{KGP}) \) key bits. In ChaCha20, \( S^{(0)} \) is added with \( S^{(26)} \) 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 8 is known to an adversary.

On the other hand, Freestyle was found to be 1.13 to 1.60 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. In implementation overhead, Freestyle’s encryption and decryption logic differ slightly. ChaCha is a simple constant time algorithm, where as 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.

6 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 known-plaintext (KPA), chosen-plaintext(CPA) and chosen-ciphertext (CCA) attacks difficult in practice. We have introduced the concepts of bounded hash based halting condition and key-guessing penalty (KGP), which are helpful in development and analysis of ciphers resistant to key-guessing attacks. Freestyle has demonstrated KGP > 1 which makes it run faster on a low-powered machine having the correct key, and is KGP times slower (with high probability) on an adversary’s machine. 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 use-cases include disk encryption, encrypted databases, password managers, sensitive data in public facing IoT devices, etc. The paper has introduced a new class of ciphers having 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 every decryption attempt with an incorrect key could be an attractive topic of research. The key challenge however is making the time taken for decryption attempt with an incorrect key, greater than the time taken to detect if the problem is NP-hard.

REFERENCES


Appendix A  REFERENCE CODE - FREESTYLE HASH FUNCTION

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

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

```c
#define MAX_INIT_HASHES (56)

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 */
    u32 temp1;
    u32 temp2;
    const u32 saved_min_rounds = x->min_rounds;
    const u32 saved_max_rounds = x->max_rounds;
    const u32 saved_hash_interval = x->hash_interval;
    const u8  saved_num_precomputed_rounds = x->num_precomputed_rounds;
```
u32 p;

if (! x->is_pepper_set)
{
    x->pepper = arc4random_uniform (x->pepper_bits == 32 ? -1 : (1 << x->pepper_bits));
}

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

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

/* 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

void freestyle_randomsetup_decrypt (freestyle_ctx *x) {
    u32 i;
    u8 R[MAX_INIT_HASHES]; /* random rounds */
Freestyle, a randomized version of ChaCha for resisting offline brute-force and dictionary attacks

```c
u32 temp1;

u32 temp2;

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;

u32 pepper;

u32 max_pepper = x->pepper_bits == 32 ? UINT32_MAX : (u32)((1 << x->pepper_bits) - 1);

/* set sane values for initialization */
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;

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

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

/* if initial pepper is set, then add it to constant[3] */
x->input[CONSTANT0] = PLUS(x->input[CONSTANT0], x->pepper);

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

    for (i = 0; i < x->num_init_hashes; ++i)
    {
        R[i] = freestyle_decrypt_block (x,
 NULL,
 NULL,
 0,
 &x->init_hash[i]
```
if (R[i] == 0) {
    goto retry;
}

freestyle_increment_counter (x);

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

```c
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 D  REFERENCE CODE - ENCRYPTION AND DECRYPTION

```c
#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;

        u32 num_rounds = freestyle_xcrypt_block(
            x,
            plaintext + i,
            ciphertext + i,
            bytes_to_process,
```

```c
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 D  REFERENCE CODE - ENCRYPTION AND DECRYPTION

```c
#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;

        u32 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 OR DECRYPT A BLOCK OF MESSAGE

#define MAX_HASH_VALUES (256)

u8 freestyle_xcrypt_block (u32 i;
    freestyle_ctx *x,
    const u8 *plaintext,
    u8 *ciphertext,
    u8 bytes,
    u8 *expected_hash,
    const bool do_encryption)
{
    u32 hash = 0;
    u32 output[16];
    bool init = (plaintext == NULL) || (ciphertext == NULL);
    u8 r;
    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(x, 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];
    }
}

return do_encryption ? rounds : r;