

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

P. Arun Babu and Jithin Jose Thomas

Abstract—This paper introduces *Freestyle*, a randomized and variable round version of the ChaCha cipher. Freestyle uses 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 Freestyle resistant to key-guessing attacks i.e. brute-force and dictionary based attacks. Freestyle demonstrates a novel approach for ciphertext randomization by using random number of rounds for each block, where the exact number of rounds are unknown to the receiver in advance. Freestyle provides the possibility of generating 2^{128} different ciphertexts for a given key, nonce, and message; thus resisting key and nonce reuse attacks. Due to its inherent random behavior, Freestyle makes cryptanalysis through known-plaintext, chosen-plaintext, and chosen-ciphertext attacks difficult in practice. On the other hand, Freestyle has costlier cipher initialization process, typically generates 3.125% larger ciphertext, and was found to be 1.6 to 3.2 times slower than ChaCha20. Freestyle is suitable for applications that favor ciphertext randomization and resistance to key-guessing and key reuse attacks over performance and ciphertext size. Freestyle is ideal for applications where ciphertext can be assumed to be in full control of an adversary, and an offline key-guessing attack can be carried out.

Index Terms—Brute-force resistant ciphers, dictionary based attacks, key-guessing, probabilistic encryption, Freestyle, ChaCha.

I. 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 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 ≥ 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]–[12]; (ii) weak passwords being used to derive keys; and, (iii) poor protocol or cryptographic implementations [13]–[15] are prone to key-guessing attacks (brute-force and dictionary based attacks). Also, steady advances are being made in the areas of GPUs [16]–[18], specialized hardware for cryptography [19]–[24], and memories in terms of storage and in-memory processing [25]–[27] to speedup key-guessing attacks.

P. Arun Babu (arun.babu@rbccps.org) is with the Robert Bosch Center for Cyber-physical Systems, Indian Institute of Science, Bengaluru.

Jithin Jose Thomas (jithint@iisc.ac.in) was with the Department of Electrical Communication Engineering, Indian Institute of Science, Bengaluru.

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 available with 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 [28] are preferred. Although, such techniques are useful, they are much slower on low-powered devices, and also slow down genuine users.

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 simpler for genuine users. We introduce the *key guessing penalty*, which is 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 2^{128} 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 [13] and cryptanalysis by XOR of ciphertexts in the event of the *key* and *nonce* being reused.

Freestyle attempts to address the following two issues: (i) reuse of a *key* and *nonce* combination is not secure in deterministic stream ciphers, as demonstrated attacks such as Key installation attack (KRACK) [13]. And maintaining a list of used *keys* and *nonces* is an overhead, especially for constrained and low-powered devices, (ii) Existing ciphers take nearly the same amount of time to decrypt a *message* irrespective of whether the *key* used is correct or not. This makes lightweight ciphers prone to key-guessing attacks. The proposed decryption algorithm in Freestyle is designed to be computationally simpler 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 brute-force or dictionary attack attempt is likely to be computationally expensive and time consuming.

The rest of the paper is structured as follows: Table I

TABLE I
LIST OF SYMBOLS

Notation	Description
R_{min}	The minimum number of rounds to be used for encryption. $R_{min} \in [1, 2^{16}]$
R_{max}	The maximum number of rounds to be used for decryption. $R_{max} \in [1, 2^{16}]$ and $R_{max} \geq R_{min}$.
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. $r \in [R_{min}, R]$
$h()$	Freestyle hash function which generates a 16-bit <i>hash</i> .
H_I	Round intervals at which a 16-bit <i>hash</i> has to be computed. $H_I \in [1, R_{min}]$, $R_{min} H_I$, $R_{max} H_I$.
H_C	The complexity of Freestyle's hash function to be used. $H_C \in \{1, 2, 3\}$
I_C	The $\log_2(\textit{iterations})$ (or number of <i>pepper</i> bits) to be used in during initialization. $I_C \in [8, 32]$
<i>pepper</i>	The <i>pepper</i> value indicating the number of iterations required during initialization. $pepper = random(0, 2^{I_C} - 1)$.
C_{R_i}	The number of rounds computed using an expected <i>hash</i> and <i>pepper</i> for i^{th} block of message.
E_{pepper}	The expected value of <i>pepper</i> .
E_{R_w}	The expected number of rounds executed by an adversary during cipher initialization.
E_R	The expected number of rounds used by a genuine user to encrypt/decrypt a block of <i>message</i> . If a uniform distribution is used, then $E_R = \frac{R_{min} + R_{max}}{2}$.
v (in red color)	An input variable.
v (in green color)	A variable derived from one or more input variables.
v (in blue color)	An output variable.
$v^{(r)}$	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 .
$v_1 v_2$	Concatenation of v_1 and v_2 .
$v_1 v_2$	v_2 is a factor of v_1 .
$v_1 \oplus v_2$	Bit-wise XOR of v_1 and v_2 .
$v_1 \boxplus v_2$	Addition of v_1 and v_2 modulo 2^{32} .
$v_1 \boxminus v_2$	Subtraction of v_1 and v_2 modulo 2^{32} .
mod	The modulo operator.
v^*	Set of values guessed by an adversary for v .
$c_f(v_1, v_2)$	A set containing common factors of integers v_1 and v_2 .
$ v $	The length of v in bits.
N_b	The number of blocks in a <i>message</i> . $N_b = \left\lceil \frac{ message }{512} \right\rceil$
$Pr_n(X = 1)$	The probability of collision of a 16-bit hash at the n^{th} trial when using an incorrect key.
N_c	The total number of ciphertexts possible for a given: <i>key</i> , <i>nonce</i> , and <i>message</i> .
N_r	The number of ways a block of message can be encrypted by using random number of rounds. $N_r = \left(\frac{R_{max} - R_{min}}{H_I} + 1 \right)$
$T(o)$	The expected time taken to execute the operation o .
S	The 512-bit cipher state for a given block of <i>message</i> .
<i>counter</i>	The counter in CTR mode of operation.
<i>null</i>	An empty string.

and Table II lists the notations and abbreviations used in the paper; section II presents the background information on ChaCha cipher and its variants; section III describes the Freestyle cipher; section IV presents results and cryptanalysis of Freestyle cipher; section V presents related work; and section VI concludes the paper.

II. CHACHA CIPHER AND VARIANTS

ChaCha20 [1] is a variant of Salsa20 [29], [30], a stream cipher. ChaCha20 uses 128-bit *constant*, 256-bit *key*, 64-

TABLE II
LIST OF ABBREVIATIONS

Abbreviation	Expansion
ARX	Add-Rotate-XOR
CAPTCHA	Completely Automated Public Turing test to tell Computers and Humans Apart
CCA	Chosen Ciphertext Attack
CPA	Chosen Plaintext Attack
CTR	Counter mode of operation
DoS	Denial of service
HKDF	Halting Key-Derivation Function
KGP	Key Guessing Penalty
KPA	Known Plaintext Attack
KRACK	Key Re-installation Attack
MAC	Message Authentication Code
MITM	Man In The Middle Attack
NONCE	Number used once
QR	Quarter Round
SSH	Secure Shell
TLS	Transport Layer Security

bit *counter*, and 64-bit *nonce* to form an initial cipher state denoted by $S^{(0)}$, as:

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

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

$$\begin{aligned} QR(S[0], S[4], S[8], S[12]) \\ QR(S[1], S[5], S[9], S[13]) \\ QR(S[2], S[6], S[10], S[14]) \\ QR(S[3], S[7], S[11], S[15]) \end{aligned} \quad (1)$$

$$\begin{aligned} QR(S[0], S[5], S[10], S[15]) \\ QR(S[1], S[6], S[11], S[12]) \\ QR(S[2], S[7], S[8], S[13]) \\ QR(S[3], S[4], S[9], S[14]) \end{aligned} \quad (2)$$

The 16 elements of the cipher-state matrix are denoted by using an index in range [0,15], and the quarter-round $QR(a, b, c, d)$ is defined as:

$$\begin{aligned} a \leftarrow a \boxplus b; & \quad d \leftarrow d \oplus a; & \quad d \leftarrow d \lll 16; \\ c \leftarrow c \boxplus d; & \quad b \leftarrow b \oplus c; & \quad b \leftarrow b \lll 12; \\ a \leftarrow a \boxplus b; & \quad d \leftarrow d \oplus a; & \quad d \leftarrow d \lll 8; \\ c \leftarrow c \boxplus d; & \quad b \leftarrow b \oplus c; & \quad b \leftarrow b \lll 7; \end{aligned} \quad (3)$$

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 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/plain-text. 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. ChaCha8 is considered secure enough as there are no known attacks against it yet. ChaCha20 has two main variants: (i) IETF's version of ChaCha20 [2], [31] which uses a 32-bit *counter* (instead of 64-bit) and 96-bit *nonce* (instead of 64-bit); and (ii) XChaCha20 [32], which uses 192-bit *nonce* (instead of 64-bit), where a randomly generated

nonce is considered safe enough [33]. The large *nonce* in XChaCha20 makes the probability of *nonce* reuse low.

III. THE *Freestyle* CIPHER

A. Hash based halting condition

Traditionally ciphers are designed to use fixed number of rounds in the encryption and decryption process. 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, we use the concept of *hash based halting condition*.

The purpose of hash based halting condition is to make decryption take longer time to halt if an incorrect *key* is used. It works on the principle that the exact number of rounds to decrypt a block of *message* is not shared with the receiver, but can be computed by the receiver using the correct *key* and one or more *hashes*. The *hashes* must be shared by the sender in cleartext along with the ciphertext. The number of rounds (R) to be used to encrypt a given block is generated randomly by the sender from the range $[R_{min}, R_{max}]$; and only an expected *hash* of the state of the cipher after running R rounds are shared. The expected *hash* acts as a stop condition for decrypting a block of message; and the receiver has to execute the decryption algorithm till the computed *hash* equals 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 (with high probability). This property makes offline brute-force and dictionary based attacks less efficient. The hash based halting condition is only applicable to ciphers having a symmetric structure (e.g. use of feistel network).

Remark 1 For better security, R must be generated using a good uniform random number generator like hardware random number generator or cryptographically secure pseudo-random number generator (e.g. `arc4random` [34]).

Remark 2 The proposed approach makes the assumption that 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.

B. Cipher parameter

The *Freestyle* cipher is formally defined as $Freestyle(R_{min}, R_{max}, H_C, H_I, I_C)$; where R_{min} , R_{max} indicate the minimum and maximum number of rounds to be used for encryption/decryption respectively. $H_C \in \{1, 2, 3\}$, indicates the level of complexity of hash function to be used; where 1 indicates the lowest complexity, the highest performance, and the lowest security; and 3 indicates the highest complexity, the lowest performance, and the highest security. H_C is also used to determine the number of quarter rounds (QRs) to be used to compute the *hash*. H_I indicates the round intervals at which a 16-bit *hash* of cipher-state must be computed. And $I_C \in [8, 32]$ indicates the number of bits used to generate a random number (*pepper*) which is chosen between $[0, 2^{I_C})$. The *pepper* value is used as number of iterations performed to initialize the cipher. The *pepper*

in general is a number which has the same function as *salt*, but is usually of fewer bits, and is not stored along with the hash or ciphertext (i.e. can be forgotten by the sender after use) [35], [36]. At initialization, *Freestyle* concatenates R_{min} , R_{max} , H_C , and H_I ; to generate a unique 64-bit *cipher_parameter* as shown in the figure 1.

R_{min} (16-bits)	R_{max} (16-bits)	H_I (16-bits)	H_C (8-bits)	I_C (8-bits)
------------------------	------------------------	--------------------	-------------------	-------------------

Fig. 1. The 64-bit *cipher_parameter*

The *cipher_parameter* is to be XOR-ed with the *key* (equation 6), which makes encryption with one *cipher_parameter* incompatible with other cipher parameters by design; thus cryptanalysis data collected for a weaker *cipher_parameter* cannot be used directly for other parameters. For a given *cipher_parameter*, the total number of ways a block of *message* can be encrypted using random number of rounds (in the range $[R_{min}, R_{max}]$) which is denoted by N_r , given as:

$$N_r = \frac{R_{max} - R_{min}}{H_I} + 1 \quad (4)$$

Remark 3 While choosing a *cipher_parameter*, it must be noted that the performance of *Freestyle* is $\propto \frac{H_I}{H_C \times I_C}$. The value of R_{min} must be chosen carefully based on the required security level, and is recommended that R_{min} be at least 8 as there are no known attacks for ChaCha8. For security-critical applications though, $R_{min} \geq 12$ is preferred. To have better randomization, it is recommend that $N_r \geq 4$; also, as there are only 2^{16} unique possible *hashes* represented by a 16-bit unsigned integer; R_{min} , R_{max} , and H_I must be chosen such that the following relationship holds (from equation 4):

$$3 \leq \frac{R_{max} - R_{min}}{H_I} \leq 65535 \quad (5)$$

Also, for better security, the recommended values for H_C is 3 or 2, and for H_I it is 1 or 2. I_C must be chosen based on performance and the security level required, and $I_C \geq 20$ is recommended for security-critical applications. ■

C. The initial cipher state

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, the *counter* size has been reduced to 32-bit as in practice most of the protocols such as the SSH transport protocol [37] recommend re-keying after 1GB of data sent/received.

The initial cipher state acts the input for generating a key-stream for a block of *message*. The cipher state of *Freestyle* is similar to the IETF's version of ChaCha, except that the *constant*, *key*, and *counter* is modified as shown in equation 6. The initial-state has been modified in such a way that: either a publicly known value is XOR-ed with a secret element of the matrix, or a secret value is XOR-ed with a publicly known element of the matrix.

Here, we introduce the *non-deterministic CTR mode* of operation where, the *counter* is XOR-ed with a random value that is independent of the *key* or *nonce* (unlike randomized-CTR mode where the random number is derived from *key* and/or *nonce*). Hence, the property of CTR mode of operation: that the difference between the *counters* of $(n + 1)^{th}$ block and n^{th} block is equal to 1 may no longer hold. The random number to be XOR-ed with *counter* in Freestyle is denoted by *random_word*[3], and its value is initialized during cipher initialization (section III-E and figure 4). The Freestyle cipher starts with the plain CTR mode of operation and shifts to *non-deterministic CTR mode* after 28 blocks (i.e. after random number initialization), thus making cryptanalysis difficult.

In equation 6, the *random_words* indicate the 128-bit random number generated by the sender, which can be computed by the receiver using the correct *key*. The *random_words* are initially set to 0 by both sender and receiver and must be computed while initializing the cipher (section III-E). Using the initial cipher state, Freestyle uses ChaCha's approach to generate the final state (equations 1, 2, 3); however unlike ChaCha, Freestyle supports both even and odd number of rounds.

D. Hash function

Freestyle's hash function is used to generate the hash based halting condition described in section III-A. The hash function (figure 3) generates a 16-bit *hash* using: (i) the current round number (r), (ii) the first $128 \times (H_C + 1)$ bits of current cipher state ($S^{(r)}$), (iii) the 128-bit *random_words*, and (iv) the previous *hash* (i.e. $hash^{(r-H_I)}$).

To resist timing-based or side-channel attacks, the hash function uses Add-Rotate-XOR (ARX) operations, the same set of operations used by Freestyle quarter-round (QR). Also, unlike a typical cryptographic hash function, Freestyle does not require high collision-resistant hash function. The probability of $\frac{1}{2^{16}}$ for collision is enough for its purpose.

E. Random number initialization

As mentioned earlier in section III-A, Freestyle uses random number of rounds to encrypt a *message* (equation 4). To randomize ciphertext even further, Freestyle requires the sender to generate a 128-bit random number denoted by *random_words*; that will act as one of the inputs for encryption and decryption. Freestyle enables a sender to securely send *random_words* to the receiver even though the *key* and *nonce* may be reused.

After the *cipher_parameter* is computed (section III-B), the following temporary configuration is set irrespective of the *cipher_parameter*:

$$R_{min} = 12, R_{max} = 36, H_C = 3, H_I = 1 \quad (7)$$

This is done to ensure there is enough entropy even if weaker values of R_{min} and R_{max} are provided by the user; and also in cases where the parameters can be downgraded in Man in the middle (MITM) attacks such as Logjam [15].

The sender then sets *random_words* to 0 and generates a random *pepper* (p) in the range $[0, 2^{1c})$, which is added to

the initial cipher-state. The sender then generates 28 random numbers (R_0 to R_{27}) in the range $[12, 36]$ using a uniform distribution. Each of the 28 random numbers is then used as number of rounds (equations 1, 2, and 3) in Freestyle cipher to generate 28 *hashes* after executing R_i rounds, where $i \in [0, 27]$ (figure 3). It must be noted that for each of the 28 round numbers, no encryption is performed, only expected *hashes* are generated. The sender also ensures that hash collisions are handled correctly, which is a crucial step for correct decryption by the receiver. The sender then sends the 28 *hashes* to the receiver, and computes *random_words* from R_i values as shown in figure 4.

On the other hand, the receiver first sets the *random_words* to 0; and increments $S^{(0)}$ [3] (i.e. the *constant*[3]) and for each increment, computes 28 *hashes*, until the computed *hashes* equals with the received 28 *hashes*. Receiver then computes the R_0 to R_{27} from: *key*, *nonce*, and 28 *hashes*. Using which, *random_words* are computed as shown in figure 4.

Finally, both sender and receiver will: reset the *counter* to 0, set R_{min} , R_{max} , H_C , and H_I to their original values, and the new initial cipher state is computed using equation 6. From now on, the new initial cipher state will be used for encryption/decryption (section III-F). The steps to initialize the Freestyle cipher are described in Algorithm 1 and 2.

Remark 4 The rationale behind using 28 random number of rounds to generate a 128-bit *random_words* is: as the total possible random numbers that a sender can choose between $[12, 36]$ using $H_I = 1$ is 25 (equation 4). Thus, if the sender has to generate n random numbers, and as the *random_words* is a 128-bit value, $25^n \geq 2^{128}$ to be a good random number generator. Thus, n must be at least 28. The *random_words* provide the possibility of generating 2^{128} different ciphertexts for a given *key*, *nonce*, and *message*.

Remark 5 The proposed approach is different from generating a 128-bit random number (\mathcal{R}) and sending it in encrypted form. For example:

$$encrypt(\mathcal{R}, key) \parallel encrypt'(message, key, \mathcal{R}) \quad (8)$$

In the latter case, for stream-ciphers, if the *key* and *nonce* are reused, there is a possibility of cryptanalysis by XOR-ing ciphertexts.

F. Encryption and decryption

After the computation of *random_words* and the new initial cipher state ($S^{(0)}$); to encrypt a block of *message*, the sender generates a random number (R) in the range $[R_{min}, R_{max}]$, using which a key-stream and a *hash* are generated after R rounds of Freestyle. The plaintext is XOR-ed with the key-stream to generate the ciphertext. The ciphertext along with the expected *hash* is sent to the receiver.

On the other hand, to decrypt a block of *message*, the receiver computes the $S^{(r)}$ using n number of H_I rounds of Freestyle until R_{max} rounds or until the computed *hash* at the end of each H_I rounds equals with the received *hash*. After which, a key-stream is generated which is then XOR-ed with the ciphertext to generate the plaintext. The steps to

$$S^{(0)} = \left[\begin{array}{cccc} \left(\begin{array}{c} \text{constant}[0] \\ \oplus \\ \text{random_word}[0] \end{array} \right), & \left(\begin{array}{c} \text{constant}[1] \\ \oplus \\ \text{random_word}[1] \end{array} \right), & \left(\begin{array}{c} \text{constant}[2] \\ \oplus \\ \text{random_word}[2] \end{array} \right), & \text{constant}[3] \\ \left(\begin{array}{c} \text{key}[0] \\ \oplus \\ \text{cipher_parameter}[0] \end{array} \right), & \left(\begin{array}{c} \text{key}[1] \\ \oplus \\ \text{cipher_parameter}[1] \end{array} \right), & \text{key}[2], & \text{key}[3] \\ \text{key}[4], & \text{key}[5], & \text{key}[6], & \text{key}[7] \\ \left(\begin{array}{c} \text{counter} \\ \oplus \\ \text{random_word}[3] \end{array} \right), & \text{nonce}[0], & \text{nonce}[1], & \text{nonce}[2] \end{array} \right] \quad (6)$$

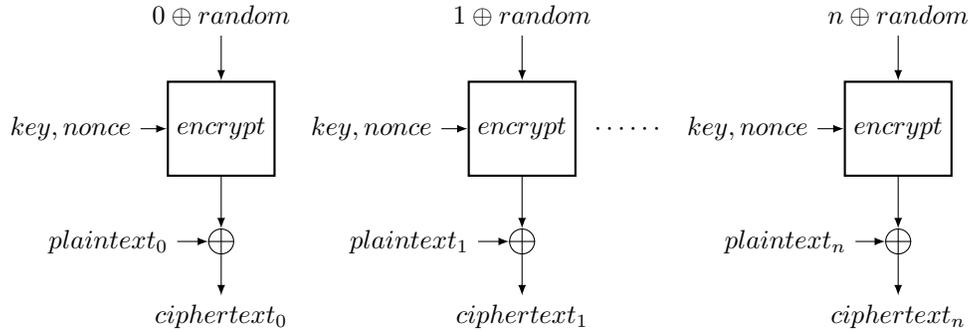


Fig. 2. Non-deterministic CTR mode of operation, where the *counter* is XOR-ed with a random number that is independent of the *key* and *nonce*

encrypt/decrypt a block (512 bits) of *message* is described in Algorithm 3 and 4, and are to be performed for each block of *message* to be exchanged.

Remark 6 If both sender and receiver have used the same *cipher_parameter*, *random_words*, *key*, and *nonce*; then the sender and receiver would have taken same number of steps and operations (i.e. R rounds) to generate the key-stream for a given block of *message*.

Remark 7 During initialization, Freestyle's hash function uses 512 bits of $S^{(R)}$ and 16 bit value of current round number r as inputs to generate a 16-bit *hash*. Whereas during encryption/decryption the hash function uses at least 256-bits of $S^{(R)}$ and 128-bit *random_words*. It is computationally infeasible to compute the *key* or key-stream using brute-force approach, as it would require at least 2^{320} operations (i.e. 256 bits of $S^{(R)}$ and at least 64 bits of Add-Rotate-XOR result of $r, \text{hash}^{(r-H_1)}$, and *random_words*) to generate all possible cipher states (or partial cipher-states in case of encryption/decryption) that may collide with a given *hash* (figure 3). Also, assuming the 16-bit *hashes* are equally spread over 2^{16} buckets, there are likely to be 2^{304} collisions.

IV. RESULTS AND DISCUSSIONS

A. 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 $[R_{min}, R_{max}]$. And since all the blocks

of the *message* use the 128-bit *random_words* as input, the total number of possible ciphertexts are:

$$N_c = 2^{128} \times (N_r)^{N_b} \quad (9)$$

From equation 9, as the number of blocks in a *message* increases, the number of possible ciphertext increases exponentially.

B. Resisting cryptanalysis

1) *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.

2) *Chosen-ciphertext attacks (CCA)*: In chosen-ciphertext attacks we consider two cases based on the adversary's ability to control the *nonce*.

a) *If nonce cannot be controlled by the adversary*: To generate a ciphertext, an adversary while initializing the cipher (section III-E) has to provide 28 valid *hashes*, and at least one valid *hash* for sending block(s) of ciphertext. As a random round is chosen between $[12,36]$ to initialize the *random_words* (equation 4), there are only 25 valid values for *hash*. While performing decryption, the total possible *hashes* that can be accepted by the receiver for a block of ciphertext is $N_r = \left(\frac{R_{max}-R_{min}}{H_I} + 1 \right)$. And as there are 2^{16} possible values for *hash*, to send a valid ciphertext, the

Algorithm 1 Freestyle initialization for the sender

```

1: procedure FREESTYLE_INIT_SENDER
   Inputs:  $S^{(0)}, R_{min}, R_{max}, H_I, H_C$ 

2:   Save the values of  $R_{min}, R_{max}, H_C$ , and  $H_I$ 
3:   Set  $R_{min} \leftarrow 12, R_{max} \leftarrow 36, H_C \leftarrow 3, H_I \leftarrow 1$ 
4:   Set  $random\_word[i] \leftarrow 0, \forall i \in [0, 3]$ 
5:    $pepper \leftarrow random(0, 2^{I_C} - 1)$ 
6:    $S^{(0)}[3] \leftarrow S^{(0)}[3] \boxplus pepper$ 

   ▷ Generate 28 hashes using 28 random number of rounds
7:   for  $i \leftarrow 0$  to 27 do
8:      $\{R_i, hash[i]\} \leftarrow freestyle\_encrypt\_block ($ 
            $S^{(0)},$ 
            $null,$ 
            $random\_word,$ 
            $R_{min},$ 
            $R_{max},$ 
            $H_I,$ 
            $H_C,$ 
            $i$            ▷ the counter
       )

9:   end for

   ▷ Check if the receiver will find a hash collision between
   0 and  $(pepper - 1)$ . If yes, update  $R_i, \forall i \in [0, 27]$ 
10:   $S^{(0)}[3] \leftarrow S^{(0)}[3] \boxplus pepper$            ▷ Restore constant
11:  for  $p \leftarrow 0$  to  $(pepper - 1)$  do
12:    for  $i \leftarrow 0$  to 27 do
13:       $C_{R_i} \leftarrow freestyle\_decrypt\_block ($ 
            $S^{(0)},$ 
            $null,$ 
            $hash[i],$            ▷ expected hash
            $random\_word,$ 
            $R_{min},$ 
            $R_{max},$ 
            $H_I,$ 
            $H_C,$ 
            $i$            ▷ the counter
       )

14:      if  $C_{R_i} = 0$  then
15:        goto step 20 ▷ Increment  $pepper$  and retry
16:      end if
17:    end for

18:     $R_i \leftarrow C_{R_i}, \forall i \in [0, 27]$            ▷ Found a collision
19:    break

20:     $S^{(0)}[3] \leftarrow S^{(0)}[3] \boxplus 1$            ▷ Retry
21:  end for

22:  Compute  $random\_words$  (as given in figure 4)
23:  Restore the original values of  $R_{min}, R_{max}, H_C, H_I$ 
24:   $S^{(0)}[12] \leftarrow 0$            ▷ Reset counter

25:  return  $hash[i], \forall i \in [0, 27]$ 
26: end procedure
    
```

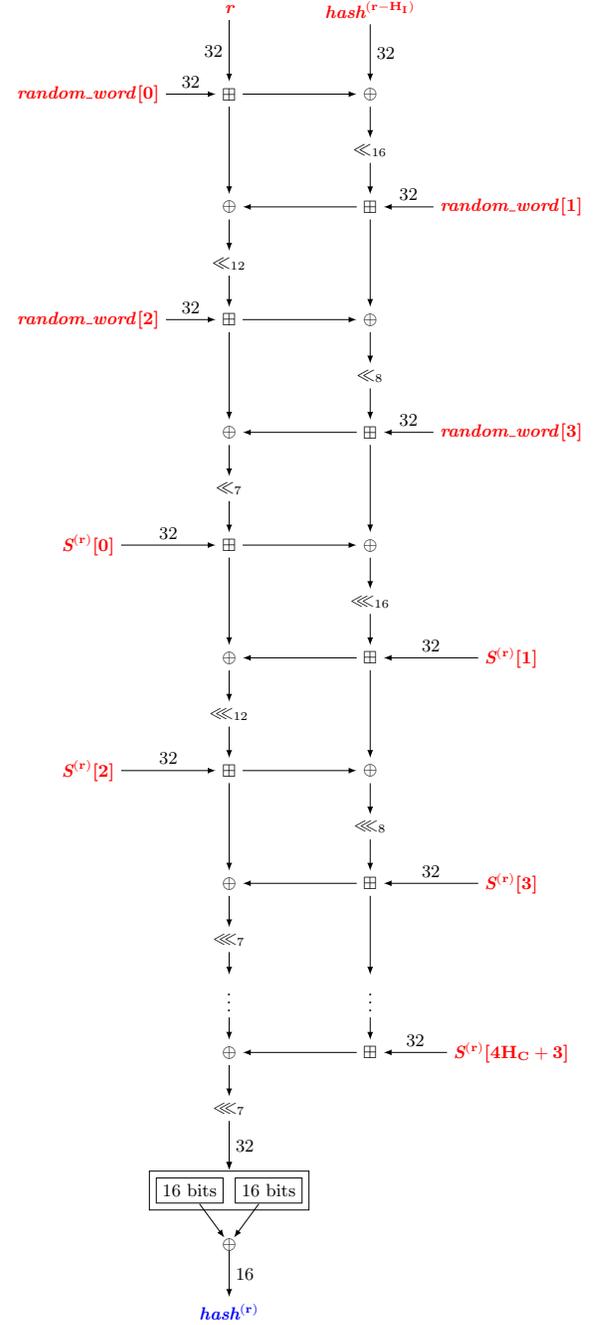


Fig. 3. The Freestyle hash function - $h(\cdot)$, for the round r (the size of variables are in bits). Note that the value of $hash^{(R_{min}-H_I)}$ is always 0.

adversary has to send $(28 + N_b)$ valid *hashes*. By brute-force approach, the probability of such an event occurring is:

$$\left(\frac{25}{2^{16}}\right)^{28} \times \left(\frac{N_r}{2^{16}}\right)^{N_b} \quad (10)$$

$$< \frac{1}{2^{317}} \quad (11)$$

Assuming a constant time cryptographic implementation to check the validity of $(28 + N_b)$ *hashes*, it is infeasible to generate a ciphertext that can be accepted by a receiver. This

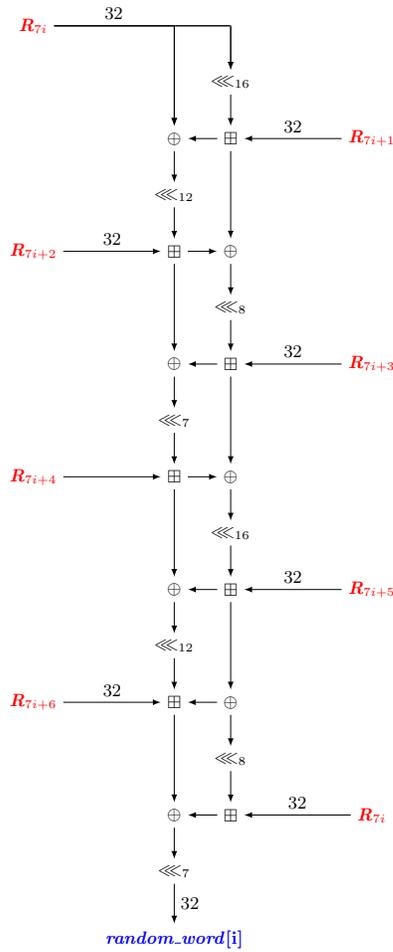


Fig. 4. Generation of $random_word[i]$, where $i \in [0, 3]$ (the size of variables are in bits)

makes chosen-ciphertext attacks difficult in practice if *nonce* cannot be controlled by the adversary.

b) *If nonce can be controlled by the adversary:* In this case, the adversary can launch CPA which can reveal $(28 + N_b)$ valid *hashes*. Thus, 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 *random_words*. It is computationally infeasible to know which combination of *key* and *random_words* the 28 *hashes* map to.

Remark 8 It must be noted that Freestyle's hash function does not use *message* 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 [38].

3) *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 $|M_2| \geq \log_2(N_c)$:

$$Pr(C_1 \oplus C_2 = M_1 \oplus M_2) = \frac{1}{N_c} \quad (12)$$

Algorithm 2 Freestyle initialization for the receiver

```

1: procedure FREESTYLE_INIT_RECEIVER
   Inputs:  $S^{(0)}$ , cipher_parameter, hash

2:   Save the values of  $R_{min}$ ,  $R_{max}$ ,  $H_C$ , and  $H_I$ 
3:   Set  $R_{min} \leftarrow 12$ ,  $R_{max} \leftarrow 36$ ,  $H_C \leftarrow 3$ ,  $H_I \leftarrow 1$ 
4:   Set  $random\_word[i] \leftarrow 0$ ,  $\forall i \in [0, 3]$ 

5:   for  $pepper \leftarrow 0$  to  $(2^{I_C} - 1)$  do
6:     for  $i \leftarrow 0$  to 27 do
7:        $C_{R_i} \leftarrow freestyle\_decrypt\_block($ 
            $S^{(0)}$ ,
           null,
            $hash[i]$ ,  $\triangleright$  expected hash
           random_word,
            $R_{min}$ ,
            $R_{max}$ ,
            $H_I$ ,
            $H_C$ ,
            $i$   $\triangleright$  the counter
       )
8:       if  $C_{R_i} = 0$  then
9:         goto step 13  $\triangleright$  Increment pepper and retry
10:      end if
11:    end for

12:    break  $\triangleright$  Found all 28 valid round numbers ( $R_i$ )

13:     $S^{(0)}[3] \leftarrow S^{(0)}[3] \oplus 1$   $\triangleright$  Retry
14:  end for

15:  Compute random_words (as given in figure 4)

16:  Restore the original values of  $R_{min}$ ,  $R_{max}$ ,  $H_C$ ,  $H_I$ 
17:   $S^{(0)}[12] \leftarrow 0$   $\triangleright$  Reset counter

18: end procedure

```

The equation 12 indicates that Freestyle is resistant to key re-installation attacks like KRACK [13]. Also, in existing approaches of ciphertext randomization, in case of *key* and *nonce* being reused, the random bytes to be shared with receiver are prone to XOR attacks. However, this is not possible with Freestyle, as only *hashes* are sent to the receiver. And the random bytes are never sent to the receiver neither in plain or encrypted form.

C. Resisting brute-force and dictionary attacks

Freestyle cipher can resist brute-force and dictionary attacks in *three* ways: (i) By keeping the *cipher_parameter* secret, (ii) Restricting pre-computation of stream, (iii) Wasting adversary's time and computational power.

1) *By keeping cipher_parameter a secret:* In Freestyle cipher, the secrecy of the plaintext depends only on the secrecy of the *key*; and the *cipher_parameter* in general need not be kept secret. The main purpose of *cipher_parameter* (figure

Algorithm 3 Encryption of a block of message

```

1: procedure FREESTYLE_ENCRYPT_BLOCK
   Inputs:  $S^{(0)}$ , plaintext, random_word,
            $R_{min}$ ,  $R_{max}$ ,  $H_I$ ,  $H_C$ , counter

2:    $hash \leftarrow 0$ 

3:    $collided[h] \leftarrow false, \forall h \in [0, 2^{16}]$ 

4:    $S^{(0)}[12] \leftarrow counter \oplus random\_word[3]$ 

5:    $R \leftarrow random(R_{min}, R_{max})$ 

6:   for  $r \leftarrow 1$  to  $R$  do

7:     Compute  $S^{(r)}$  using Freestyle (Equations 1, 2)

8:     if  $r \geq R_{min}$  and  $r|H_I$  then
9:        $hash \leftarrow h(S^{(r)}, r, random\_words, hash)$ 
10:      while  $collided[hash] = true$  do
11:         $hash \leftarrow (hash + 1) \bmod (2^{16})$ 
12:      end while
13:       $collided[hash] = true$ 
14:    end if

15:  end for

16:  if plaintext = null then      ▷ While initialization
17:    return  $\{R, hash\}$ 
18:  else
19:     $keystream \leftarrow little\_endian(S^{(R)} \boxplus S^{(0)})$ 
20:     $ciphertext \leftarrow plaintext \oplus keystream$ 

21:    return  $\{R, hash, ciphertext\}$ 
22:  end if
23: end procedure
    
```

1) is to discourage reuse of cryptanalysis data collected from weaker *cipher_parameters*. However, if kept secret, it can resist brute-force attacks. Assuming the adversary guesses that R_{min} and R_{max} values are in the range $[a, b]$, where a and b are divisible by H_I and $a \leq b$. Then, the total possible values of (R_{min}, R_{max}) adversary has to try is:

$$\left(\frac{b-a}{H_I} + 1\right) + \left(\frac{b-a}{H_I}\right) + \left(\frac{b-a}{H_I} - 1\right) + \dots + 1 \quad (13)$$

$$\text{or } \frac{\left(\frac{b-a}{H_I} + 1\right) \left(\frac{b-a}{H_I} + 2\right)}{2} \quad (14)$$

As $H_C \in \{1, 2, 3\}$ the number of possible values of (R_{min}, R_{max}, H_C) the adversary has to try is:

$$\frac{3}{2} \times \left(\frac{b-a}{H_I} + 1\right) \left(\frac{b-a}{H_I} + 2\right) \quad (15)$$

Algorithm 4 Decryption of a block of message

```

1: procedure FREESTYLE_DECRYPT_BLOCK
   Inputs:  $S^{(0)}$ , plaintext, expected_hash, random_word,
            $R_{min}$ ,  $R_{max}$ ,  $H_I$ ,  $H_C$ , counter

2:    $R \leftarrow 0$ 
3:    $hash \leftarrow 0$ 

4:    $collided[h] \leftarrow false, \forall h \in [0, 2^{16}]$ 

5:    $S^{(0)}[12] \leftarrow counter \oplus random\_word[3]$ 

6:   for  $r \leftarrow 1$  to  $R_{max}$  do

7:     Compute  $S^{(r)}$  using Freestyle (Equations 1, 2)

8:     if  $r \geq R_{min}$  and  $r|H_I$  then
9:        $hash \leftarrow h(S^{(r)}, r, random\_words, hash)$ 
10:      while  $collided[hash] = true$  do
11:         $hash \leftarrow (hash + 1) \bmod (2^{16})$ 
12:      end while

13:      if  $hash = expected\_hash$  then
14:         $R \leftarrow r$ 
15:        break
16:      end if

17:       $collided[hash] = true$ 
18:    end if

19:  end for

20:  if plaintext = null then      ▷ While initialization
21:    return  $R$ 
22:  else
23:     $keystream \leftarrow little\_endian(S^{(R)} \boxplus S^{(0)})$ 
24:     $plaintext \leftarrow ciphertext \oplus keystream$ 

25:    return  $\{R, plaintext\}$ 
26:  end if
27: end procedure
    
```

If the adversary's guesses for R_{min}, R_{max} is represented as R_{min}^* and R_{max}^* , then:

$$R_{min}^* = \{a, a + 1, \dots, b\} \quad (16)$$

$$R_{max}^* = \{a, a + 1, \dots, b\} \quad (17)$$

Such that the guessed $R_{min} \leq R_{max}$, and the value of $H_I \in c_f(R_{min}, R_{max})$, where $c_f(R_{min}, R_{max})$ is a set containing common factors of R_{min} and R_{max} . Also, as $I_C \in [12, 36]$ there are 25 possible values of I_C . Then, the total possible values of $(R_{min}, R_{max}, H_C, H_I, I_C)$ or *cipher_parameters* are:

$$\sum_{n \in R_{min}^*} \sum_{\substack{m \in R_{max}^* \\ m \geq n}} \sum_{h \in c_f(m,n)} \frac{75}{2} \left(\frac{m-n}{h} + 1 \right) \left(\frac{m-n}{h} + 2 \right) \quad (18)$$

For example, if an adversary guesses $a = 8$ and $b = 32$, then using the equation 18, the effort required for the brute-force attack increases by 42525 ($\approx 2^{15}$) times. Thus, for an effective attack, the adversary has to depend on other sources of information to guess the *cipher_parameter*.

2) *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 data. 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 *random_words* and *hash*, the exact key-stream cannot be pre-computed unless the sender sends expected *hashes*. This however, also restricts pre-computation of key-stream for a genuine receiver.

3) *Wasting adversary's time and computational resources*: Here we introduce the Key-guessing penalty (KGP) metric which indicates the penalty an adversary has to pay in terms of computational power if an incorrect *key* is used.

Definition : Key-guessing penalty (KGP) - The ratio of expected time taken for attempting to decrypt a *message* using an incorrect *key* and the expected time taken to decrypt a *message* using a correct *key* (equation 19).

$$\frac{T(\text{attempt to decrypt a message using an incorrect key})}{T(\text{decrypt a message using the correct key})} \quad (19)$$

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 high end laptop) than the attacked system (e.g. a low powered IoT device). Such ciphers forces the adversary to use a machine that is at least KGP times faster than the attacked system, to launch an effective attack.

Remark 9 While computing KGP, the probability that an adversary can detect if the guessed *key* is incorrect must be taken into account. For ciphers with $KGP > 1$, for a given length of *message*, the amount of time required by an adversary to detect if the attempted *key* is incorrect must be greater than the time taken to attempt decryption using the incorrect *key*.

Remark 10 $KGP > 1$ 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. ■

If the sender uses uniform distribution to select the *pepper* value, the E_{pepper} will be $2^{(I_C-1)}$; however, for an adversary, since the *hashes* are unlikely to match, would require 2^{I_C} attempts. Hence, the maximum KGP one can expect using uniform distribution is ≈ 2 . To improve KGP, the sender must use a right-skewed distribution which is kept secret and is not needed to be shared with the receiver. A right-skewed distribution is the one which tends to use smaller values for *pepper*.

Remark 11 Irrespective of the distribution used to generate *pepper* and the number of rounds for encryption/decryption, to generate *random_words* a good (pseudo-)random number generator with uniform distribution must be used. ■

As mentioned earlier in section III-E, during initialization a temporary configuration of $R_{min} = 12, R_{max} = 36, H_I = 1, H_C = 3$ is set. When an adversary uses an incorrect *key*, the probability of having a collision for a 16-bit *hash* changes in each trial, and not all *hashes* have equal probability of occurring. Also, it must be noted that *hashes* are picked without replacement i.e. if a collision occurs, the *hash* is incremented until there is no collision. Then, in the worst-case scenario, the maximum difference between the probability of getting two *hashes* which may occur at the 24th trial is:

$$\left(\frac{25}{2^{16} - 24} \right) - \left(\frac{1}{2^{16} - 24} \right) = 0.0003 \quad (20)$$

which is negligible value for all practical purposes. Hence, for simplicity, we present approximate results assuming that all the *hashes* at a given trial are equally likely. Then, the probability of colliding a 16-bit *hash* at the n^{th} trial when an incorrect *key* or *pepper* is used (denoted by $Pr_n(X = 1)$) is given as:

$$\begin{cases} \frac{1}{2^{16}}, & \text{if } n = 1 \\ \left(\frac{1}{2^{16} - n + 1} \right) \times \prod_{i=0}^{n-1} \left(\frac{2^{16} - i - 1}{2^{16} - i} \right), & \text{Otherwise} \end{cases} \quad (21)$$

Then, the expected number of rounds a user with an incorrect *key* or *pepper* will execute is denoted by E_{R_w} can be computed as given in equation 22, i.e. $E_{R_w} \approx 36.0095$.

During the cipher initialization, for a correct *key* and *pepper*, the expected number of rounds a user will execute is 24 (i.e. average of 12 and 36). After initialization, R_{min} , R_{max} , and H_C are set to their original values, and while decryption, if the expected number of rounds a genuine user executes is denoted by E_R . To compute KGP using equation 19, the adversary has to execute $2^{I_C} \times E_{R_w}$ rounds during initialization, and E_R rounds to decrypt a single block of *message*. Where as a genuine user has to run $E_{pepper} \times E_{R_w}$ rounds during initialization, and 28×24 rounds when using the correct *pepper*, and $N_b \times E_R$ rounds to decrypt a *message* of N_b blocks. Hence KGP is computed as:

$$E_{R_w} \approx \sum_{h=1}^{28} \left(\sum_{n=1}^{N_r} Pr_n(X=1) \right)^{h-1} \left[\left(\sum_{n=1}^{N_r} (R_{min} + nH_I) Pr_n(X=1) \right) + R_{max} \left(1 - \sum_{n=1}^{N_r} Pr_n(X=1) \right) \right] \approx 36.0095 \quad (22)$$

$$KGP = \frac{2^{I_C} \times E_{R_w} + E_R \times \left(\sum_{n=1}^{N_r} Pr_n(X=1) \right)^{28}}{E_{pepper} \times E_{R_w} + 28 \times 24 + N_b \times E_R} \quad (23)$$

The probability of getting all the 28 *hashes* correct and attempting to decrypt the first block of *message* using an incorrect *key* is:

$$\left(\sum_{n=1}^{N_r} Pr_n(X=1) \right)^{28} \approx 10^{-96} \quad (24)$$

which is negligible for all practical purposes. Hence:

$$KGP \approx \frac{2^{I_C} \times E_{R_w}}{E_{pepper} \times E_{R_w} + 28 \times 24 + N_b \times E_R} \quad (25)$$

i.e. for $KGP > 1$:

$$E_{pepper} < 2^{I_C} - \left(\frac{672 + N_b \times E_R}{36.0095} \right) \quad (26)$$

The value of E_{pepper} , E_R , and I_C can be chosen considering the performance, security level, and the required KGP. The figure 5 shows the result of KGP vs. E_{pepper} for $R_{min} = 8, R_{max} = 32, H_C = 3, H_I = 1, E_R = 20, I_C \in \{20, 24, 28, 32\}$, and various message sizes 64 bytes to 4GB.

D. Better security for 128-bit keys

Though, not recommended, ChaCha supports 128-bit *keys* by concatenating the *key* with itself to form a 256-bit *key*. In Freestyle, *cipher_parameter* and *random_words* are used to modify the initial state of cipher to provide an additional 128-bit random secret (in the form of *random_words*). The *random_words* are statistically independent of the *key* and *nonce* (equation 6); hence, for applications where 128-bit *keys* have to be used, Freestyle offers better security than ChaCha.

E. Overheads

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 $H_C + 2$ quarter rounds of Freestyle. Hence, the computational overhead for encryption is:

$$= T(\text{generate } N_b \text{ random numbers}) + \sum_{i=1}^{N_b} \left(\frac{R_i - R_{min}}{H_I} + 1 \right) (H_C + 2) \times T(1 \text{ QR of Freestyle}) \quad (27)$$

The worst case performance overhead is when $R_i = R_{max}, \forall i$. The figure 6 shows the comparison of performance between optimized versions of and ChaCha20¹ and Freestyle² with various configurations without accounting for the time taken for initialization. The results were obtained on Intel Core i5-6300HQ processor with `arc4random` [34] as the random number generator on OpenBSD. For the performance tests, $R_{min} = 8, R_{max} = 32$ 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 1.6 to 3.2 times slower than ChaCha20 (figure 6).

2) *Bandwidth overhead*: Freestyle algorithm requires a sender to send the final round 16-bit *hash*; i.e. requiring to send extra 8 bits for each block of *message* to be sent. Also, for initialization of *random_word*, it requires extra 28×16 bits. Hence, the total bandwidth overhead in bits is $16N_b + 28 \times 16$, i.e.

$$\text{Bandwidth overhead (in \%)} = \frac{16N_b + 512}{|message|} \times 100 \quad (28)$$

For a *message* of length in multiples of 512 bits, the bandwidth overhead is $\approx 3.125\%$.

F. Side-channel attacks

Freestyle uses Add-Rotate-XOR instructions to resist timing and side channel attacks. However, if a device leaks information related to randomness, Freestyle is at least as secure as ChaCha with R_{min} rounds. This is because, from *hash* at least 2^{256} operations are required to generate possible cipher states ($S^{(R)}$), and since *hash* is a 16 bit value, there are likely to be 2^{248} cipher states that collide a given *hash*. For devices that have weak or insecure (pseudo-)random number generator, we recommend a conservative configuration of $H_C = 3$; in which case at least 2^{512} operations are required to generate the cipher states colliding a given *hash*.

Another potential attack in Freestyle could be to gather intermediate *hashes*. While performing encryption, Freestyle requires maintaining the hash collision information for various *hashes*. One of the simplest implementation is to use a look-up table implementations which are prone to timing attacks. Although, by leaking intermediate *hashes* it is computationally infeasible to compute the *key* or key-stream. However, with the leaked *hash*, the adversary can detect if the attempted *key* is correct or not after R_{min} rounds; making $KGP < 1$. Also, some cryptanalysis advantage may be gained by observing consecutive *hashes*. Although, this would also require knowledge of *cipher_parameter* and 128-bit *random_words*.

¹<http://cvswb.openbsd.org/cgi-bin/cvswb/src/usr.bin/ssh/chacha.c?rev=1>

²<https://github.com/aron-babu/freestyle>

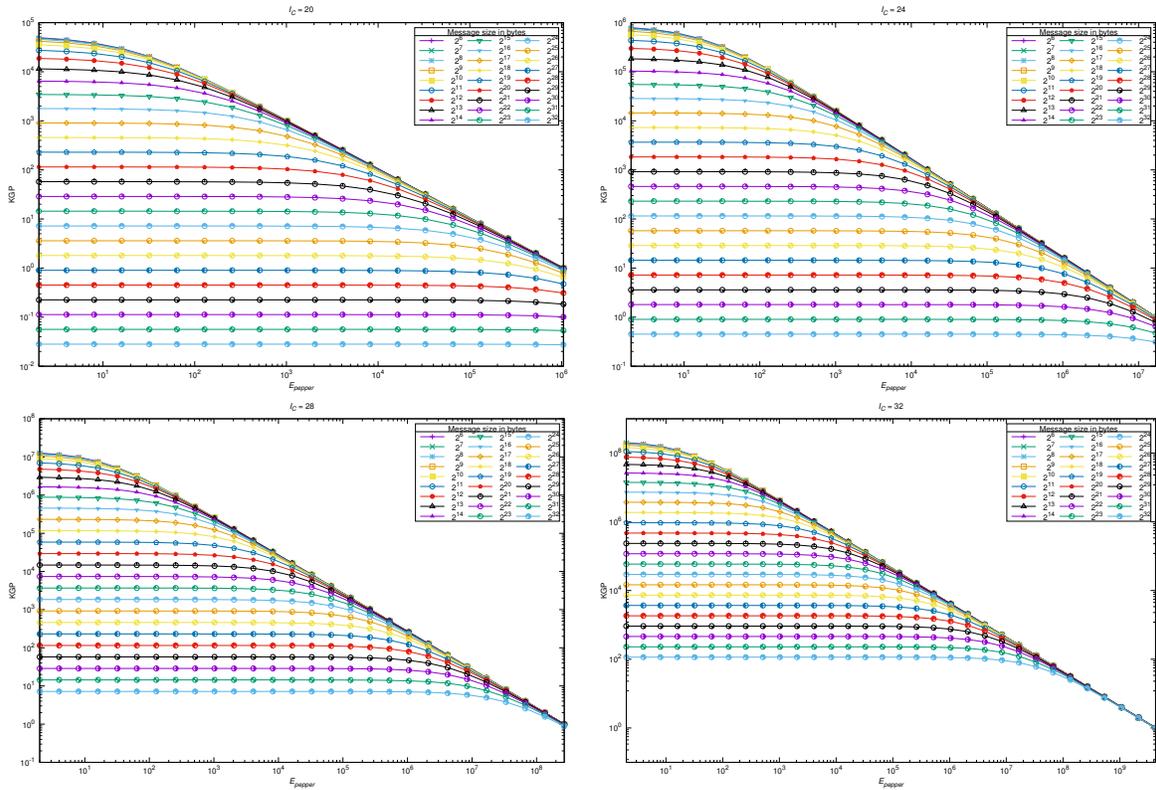


Fig. 5. KGP vs E_{pepper} for $R_{min} = 8$, $R_{max} = 32$, $H_C = 3$, $H_I = 1$, $E_R = 20$, $I_C \in \{20, 24, 28, 32\}$, and various message sizes (64 bytes to 4GB)

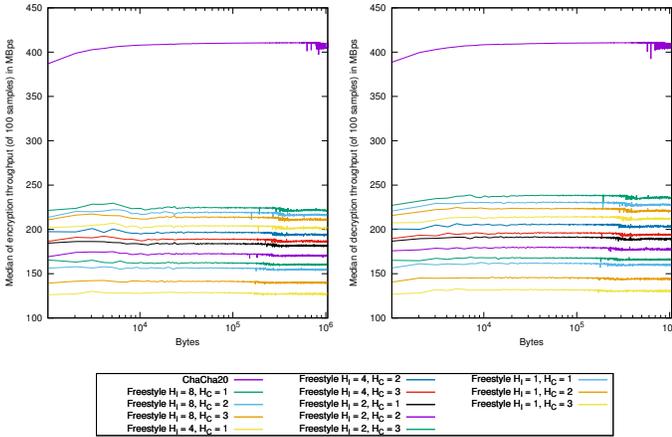


Fig. 6. Performance comparison of Freestyle with $R_{min} = 8$, $R_{max} = 32$ vs ChaCha20 on Intel Core i5-6300HQ processor without accounting for the time taken for cipher initialization

Such attacks can be resisted by obscuring look-up table indices by XOR-ing it with a 16-bit random mask. The random mask is to be generated for each block of *message* to be encrypted/decrypted and provides 2^{16} different timing variations. Though, such techniques may resist timing attacks; but do not guarantee protection against such attacks. Also, computation and memory overheads must be taken into account before considering such implementations.

Observing the time taken for encryption/decryption may be used by an adversary to predict the *pepper* and thus reducing

KGP. To prevent such attacks, the *pepper* value can be shared with the receiver through a secure channel. However, this approach is equivalent to increasing the *key* size by I_C bits.

V. RELATED WORK

A. Randomized encryption schemes

Use of randomized encryption schemes have been in practice for many years, and a taxonomy of randomized ciphers is presented in [39]. Also, some approaches to randomized encryption for public-key cryptography was proposed in [40]–[42]. Approaches based on chaotic systems for probabilistic encryption were also proposed [43]. However, the main concern with some of the existing approaches are high bandwidth expansion factor and computational overhead [39], [44].

The key difference between existing approaches and the current work is: the random bytes are never sent to the receiver in plain nor in the encrypted form. The random bytes are to be computed by the receiver from the initial 28 *hashes*. The initial 28 *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 2^{128} 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 $\approx 3.125\%$.

B. 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-stretching [45] or key-setup phase [46] using a cost-factor. Also approaches that consume large amount of memory have also been proposed [36], [47]. Another related area is use of client puzzles [48] and proof-of-work (e.g. Bitcoin [49]) to delay cryptographic operations.

The *hash based halting condition* described in section III-A, on a high-level uses similar principle as the Halting key derivation function (HKDF) proposed in [48]. In HKDF, a sender using a password and a 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 [48] 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 a 28 number of 16-bit *hashes* for initialization and a 16-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. Similarly, Freestyle offers flexibility in choosing the complexity of hash function using a hash complexity (H_C) parameter; and (vi) Freestyle forces the cipher initialization with $R_{min} = 12$ and $R_{max} = 36$, thus ensures enough randomness even in cases where user provides insecure parameters for cipher initialization; and (vi) Freestyle offers the possibility of much higher KGP by allowing the sender to choose a right-skewed distribution to generate *pepper* and R_i .

C. Freestyle vs ChaCha

When compared to ChaCha, Freestyle offers better security for 128-bit keys (section IV-D). It also provides the possibility of generating 2^{128} ciphertexts for a given *message* even if *nonce* and *key* is reused (section IV-A). 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 IV-B1). Freestyle offers the possibility of $KGP > 1$, which makes it resistant to brute-force and dictionary based attacks (section IV-C).

On the other hand, Freestyle is 1.6 to 3.2 times slower than ChaCha (section IV-E), and also has a higher cost of initialization (section III-E). In terms of bandwidth overhead, Freestyle

generates $\approx 3.125\%$ larger ciphertext. And, 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 assumes that the sender has a good source of random numbers.

VI. CONCLUSION

In this paper we have introduced Freestyle, a novel randomized cipher capable of generating 2^{128} 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 to make the time taken for decryption attempt 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, pp. 3–5, 2008.
- [2] A. Langley, W. Chang, N. Mavrogiannopoulos, J. Strombergson, and S. Josefsson, "Chacha20-poly1305 cipher suites for transport layer security (tls)," tech. rep., 2016.
- [3] D. Miller and 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.
- [4] D. Miller, "chacha20poly1305 protocol." <https://cvswb.openssl.org/cgi-bin/cvswb/src/usr.bin/ssh/PROTOCOL.chacha20poly1305?annotate=HEAD>, Last accessed 1.12.2018.
- [5] "eBACS: ECRYPT Benchmarking of Cryptographic Systems." <https://bench.cr.yp.to/results-stream.html>.
- [6] E. Bursztein, "Speeding up and strengthening HTTPS connections for Chrome on Android," Apr. 2014. Google security blog, <https://security.googleblog.com/2014/04/speeding-up-and-strengthening-https.html>.

- [7] "Vulnerability Note VU#307015, Infineon RSA library does not properly generate RSA key pairs," Oct 2017. CVE-2017-15361, <https://www.kb.cert.org/vuls/id/307015>.
- [8] S. H. Kim, D. Han, and D. H. Lee, "Predictability of android openssl's pseudo random number generator," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pp. 659–668, ACM, 2013.
- [9] L. Bello, M. Bertacchini, and 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, and 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*, pp. 15–27, ACM, 2009.
- [11] A. Lenstra, J. P. Hughes, M. Augier, J. W. Bos, T. Kleinjung, and C. Wachter, "Ron was wrong, whit is right," tech. rep., IACR, 2012.
- [12] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman, "Mining your ps and qs: Detection of widespread weak keys in network devices," in *USENIX Security Symposium*, vol. 8, 2012.
- [13] M. Vanhoef and 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.
- [14] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and 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*, pp. 535–552, IEEE, 2015.
- [15] 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*, pp. 5–17, ACM, 2015.
- [16] W. Gu, Y. Huang, R. Qian, Z. Liu, and R. Gu, "Attacking crypto-1 cipher based on parallel computing using gpu," in *International Conference on Applications and Techniques in Cyber Security and Intelligence*, pp. 293–303, Springer, 2017.
- [17] G. Agosta, A. Barengi, and G. Pelosi, "High speed cipher cracking: the case of keeloq on cuda," 2013.
- [18] V. Chiriaco, A. Franzen, R. Thayil, and X. Zhang, "Finding partial hash collisions by brute force parallel programming," in *Systems, Applications and Technology Conference (LISAT), 2017 IEEE Long Island*, pp. 1–6, IEEE, 2017.
- [19] F. Wiemer and R. Zimmermann, "High-speed implementation of bcrypt password search using special-purpose hardware," in *ReConFigurable Computing and FPGAs (ReConFig), 2014 International Conference on*, pp. 1–6, IEEE, 2014.
- [20] K. Malvoni, D. Solar, and 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.
- [21] Z. Liu, J. Großschädl, Z. Hu, K. Järvinen, H. Wang, and I. Verbauwhede, "Elliptic curve cryptography with efficiently computable endomorphisms and its hardware implementations for the internet of things," *IEEE Transactions on Computers*, vol. 66, no. 5, pp. 773–785, 2017.
- [22] K. Javeed, X. Wang, and M. Scott, "High performance hardware support for elliptic curve cryptography over general prime field," *Microprocessors and Microsystems*, 2016.
- [23] A. Khalid, G. Paul, and A. Chattopadhyay, "Rc4-accsuite: A hardware acceleration suite for rc4-like stream ciphers," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 3, pp. 1072–1084, 2017.
- [24] F. K. Gürkaynak, R. Schilling, M. Muehlberghuber, F. Conti, S. Mangard, and 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*, pp. 19–24, ACM, 2017.
- [25] W. Kim, A. Chattopadhyay, A. Siemon, E. Linn, R. Waser, and V. Rana, "Multistate memristive tantalum oxide devices for ternary arithmetic," *Scientific reports*, vol. 6, 2016.
- [26] S. Jain, A. Ranjan, K. Roy, and A. Raghunathan, "Computing in memory with spin-transfer torque magnetic ram," *arXiv preprint arXiv:1703.02118*, 2017.
- [27] A. Sebastian, T. Tuma, N. Papandreou, M. L. Gallo, L. Kull, T. Parnell, and E. Eleftheriou, "Temporal correlation detection using computational phase-change memory," *Nature Communications*, 2017.
- [28] W. Buchanan, "When Slow Is Good - The Great Slowcoach: Bcrypt," July 2015. <https://www.linkedin.com/pulse/when-slow-good-great-slowcoach-bcrypt-william-buchanan>.
- [29] D. J. Bernstein, "Salsa20 specification," *eSTREAM Project algorithm description*, <http://www.ecrypt.eu.org/stream/salsa20pf.html>, 2005.
- [30] D. J. Bernstein, "The salsa20 family of stream ciphers," *Lecture Notes in Computer Science*, vol. 4986, pp. 84–97, 2008.
- [31] Y. Nir and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols," tech. rep., 2015.
- [32] F. Denis, "The xchacha20-poly1305 construction." https://download.libsodium.org/doc/secret-key_cryptography/xchacha20-poly1305_construction.html.
- [33] "Libsodium v1.0.12 and v1.0.13 security assessment," tech. rep., 2017. <https://www.privateinternetaccess.com/blog/wp-content/uploads/2017/08/libsodium.pdf>.
- [34] T. De Raadt, "arc4random - randomization for all occasions," 2014.
- [35] G. Kedem and Y. Ishihara, "Brute force attack on UNIX passwords with SIMD computer," 1999.
- [36] C. Forler, S. Lucks, and J. Wenzel, "Catena: A memory-consuming password-scrambling framework," tech. rep., Citeseer, 2013.
- [37] T. Ylonen and C. Lonvick, "The secure shell (ssh) transport layer protocol, rfc 4253," 2006.
- [38] D. J. Bernstein, "The poly1305-aes message-authentication code.," in *FSE*, vol. 3557, pp. 32–49, Springer, 2005.
- [39] R. L. Rivest and A. T. Sherman, "Randomized encryption techniques," in *Advances in Cryptology*, pp. 145–163, Springer, 1983.
- [40] S. Goldwasser and S. Micali, "Probabilistic encryption," *Journal of computer and system sciences*, vol. 28, no. 2, pp. 270–299, 1984.
- [41] T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," *IEEE transactions on information theory*, vol. 31, no. 4, pp. 469–472, 1985.
- [42] R. Cramer and V. Shoup, "A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack," in *Annual International Cryptology Conference*, pp. 13–25, Springer, 1998.
- [43] S. Papadimitriou, T. Bountis, S. Mavroudi, and 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*, vol. 11, no. 12, pp. 3107–3115, 2001.
- [44] S. Li, X. Mou, B. L. Yang, Z. Ji, and J. Zhang, "Problems with a probabilistic encryption scheme based on chaotic systems," *International Journal of Bifurcation and Chaos*, vol. 13, no. 10, pp. 3063–3077, 2003.
- [45] J. Kelsey, B. Schneier, C. Hall, and D. Wagner, "Secure applications of low-entropy keys," in *International Workshop on Information Security*, pp. 121–134, Springer, 1997.
- [46] N. Provos and D. Mazieres, "Bcrypt algorithm," USENIX, 1999.
- [47] C. Percival and S. Josefsson, "The scrypt password-based key derivation function," tech. rep., 2016.
- [48] X. Boyen, "Halting password puzzles," in *Proc. Usenix Security*, 2007.
- [49] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

P. Arun Babu is currently a Member of Technical Staff at Robert Bosch Center for Cyber-physical Systems at the Indian Institute of Science, Bengaluru, India. Arun holds a Ph.D in Engineering Sciences from Indira Gandhi Centre for Atomic Research, Kalpakkam, India. His areas of research include cyber-security and software engineering.

Jithin Jose Thomas is currently a Senior systems software engineer at the Trakray Innovations, Bengaluru, India. Jithin holds a B.Tech degree in Electronics and Communications Engineering from National Institute of Technology, Calicut, India. He was with the Department of Electrical Communication Engineering, Indian Institute of Science, Bengaluru at the time of this work. His areas of research include cryptography, wireless networks, and data analytics.