On Constructing a Secure and Fast Key Derivation Function Based on Stream Ciphers

Chai Wen Chuah*[1], Janaka Alawatugoda[2], Nureize Arbaiy[3]
Guangdong University of Science & Technology, Dongguang, Guangzhou, China[1]
Research & Innovation Centers Division, Rabdan Academy Abu Dhabi, UAE[2],
Institute for Integrated and Intelligent Systems, Griffith University, Nathan, Queensland, Australia[2]
Faculty of Computer Science & Information Technology, Universiti Tun Hussein Onn Malaysia, Parit Raja, Malaysia[3]

***Abstract*—In order to protect electronic data, pseudorandom cryptographic keys generated by a standard function known as a key derivation function play an important role. The inputs to the function are known as initial keying materials, such as passwords, shared secret keys, and non-random strings. Existing standard secure functions for the key derivation function are based on stream ciphers, block ciphers, and hash functions. The latest secure and fast design is a stream cipher-based key derivation function ($\mathrm{SCKDF_2}$). The security levels for key derivation functions based on stream ciphers, block ciphers, and hash functions are equal. However, the execution time for key derivation functions based on stream ciphers is faster compared to the other two functions. This paper proposes an improved design for a key derivation function based on stream ciphers, namely $\mathrm{I{-}SCKDF_2}$. We simulate instances for the proposed $\mathrm{I{-}SCKDF_2}$ using Trivium. As a result, $\mathrm{I{-}SCKDF_2}$ has a lower execution time compared to the existing $\mathrm{SCKDF_2}$. The results show that the execution time taken by $\mathrm{I{-}SCKDF_2}$ to generate an n-bit cryptographic key is almost 50 percent lower than $\mathrm{SCKDF_2}$. The security of $\mathrm{I{-}SCKDF_2}$ passed all the security tests in the Dieharder test tool. It has been proven that the proposed $\mathrm{I{-}SCKDF_2}$ is secure, and the simulation time is faster compared to $\mathrm{SCKDF_2}$.**

Keywords*—Key derivation functions; extractors; expanders; stream ciphers; hash functions; symmetric-key cryptography

I. Introduction

A Key Derivation Function (KDF) is a standard function to generate one or more pseudorandom cryptographic keys from a initial keying material. The initial keying material of the KDF consists of a non-random secret string and publicly known string. The output of the KDF is an arbitrary length of pseudorandom cryptographic key. The example of the secret string (p) can be user password, a random seed value from some entropy source, or output value such as shared secret from Diffie-Hellman (DH) key agreement [1], [2], [3]. The example of the public string is a random salt value (s) or context information (c) [4].

To date, two-phase KDFs are categorized into stream cipher-based [4], [5], hash function-based [6], [7], and block cipher-based KDFs. These KDFs consist of an extractor and an expander. The extractor takes as input a secret string and a publicly known random string, generating a pseudorandom or close-to-uniform string [8], [9], [10] (PRK) as its output. The PRK and public context information [11] serve as inputs for the expander, which produces the secret keying material. The input size can be of arbitrary length, and it is divided into equally-sized blocks for both hash function-based KDFs and block cipher-based KDFs. Padding is required for the last block to ensure consistency in block sizes.

The output of hash function-based KDFs and block cipher-based KDFs is of a fixed block size. If the derived cryptographic key output has excess bits, these additional bits are discarded, which is not an efficient use of computational resources

In this paper, we construct a stream cipher-based KDF ($\mathrm{SCKDF_2}$) using the keystream generator (KG) [5], [4]. The authors incorporated KG into the KDF designs because its properties are similar to those of KDF. For example, KG takes two inputs: the initialization vector (IV) and the secret key to generate arbitrary lengths of pseudorandom output [12], [13]. In the extractor of $\mathrm{SCKDF_2}$, the original inputs for the pseudorandom keystream generator, the key and the IV, are replaced with p and s, respectively, to generate an intermediate value, PRK.

For the expander of $\mathrm{SCKDF_2}$, the key and IV are the inputs to KG, which are replaced with PRK and c, respectively. With these inputs, the pseudorandom KG produces an n-bit cryptographic key. The findings in Chuah et al.'s work [5] demonstrate that the security level of $\mathrm{SCKDF_2}$ is similar to that of block cipher-based KDFs and hash function-based KDFs. In terms of execution time, $\mathrm{SCKDF_2}$ executes faster compared to block cipher-based KDFs and hash function-based KDFs.

The KDF is widely used in Internet protocols [14], [15], [16]. Mobile devices and Internet of Things (IoT) are increasingly used to access the Internet. These devices are designed with low processing power and limited memory size. Therefore, the KDF must be both secure and responsive. In this paper, we extend the work of Chuah et al. [5] to propose an improved design for KDF based on stream ciphers while maintaining the security level and improving execution speed. We name it $\mathrm{I{-}SCKDF_2}$.

The remainder of this paper is organized as follows: Section II presents the background information of key derivation functions. In Section III, we provide information about keystream generators for stream ciphers. Section IV introduces the research framework for the modal structure used to construct the improved stream cipher-based KDF. Sections V and VI respectively provide security and performance analyses of the improved stream cipher based on KDF. Finally, in Section VII, we present the paper's conclusion.

II. Key Derivation Functions

A Key Derivation Function (KDF) is a function that generates one or more cryptographic keys from a source of initial keying material. The initial keying material of the KDF consists of a secret string and a public string. The output of the KDF is an arbitrary length of the cryptographic key.

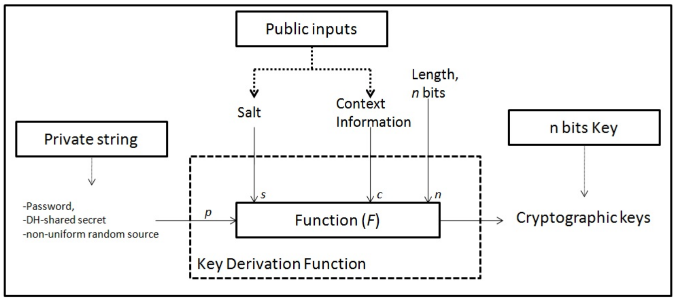

Fig. 1. Key derivation function - Single-phase KDF.

Fig. 1 shows the design of the KDF model, $K \leftarrow \mathrm{F}(p, s, c, n)$. The private or secret string is p and the public strings are s and c. F is the function. Based on these inputs, F generates n-bits of a cryptographic key, K. The value of length, n must be in a positive integer. The value of the p must be kept secret from the adversary such as user password, a random seed value from some entropy source, or output value (shared secret) from Diffie-Hellman (DH) key agreement [11]. The example of s is random salt value and c is context information [4]. The distribution for the string of s is usually close to uniform. The string of c is an application specific data such as session identifier of the communicating parties [11].

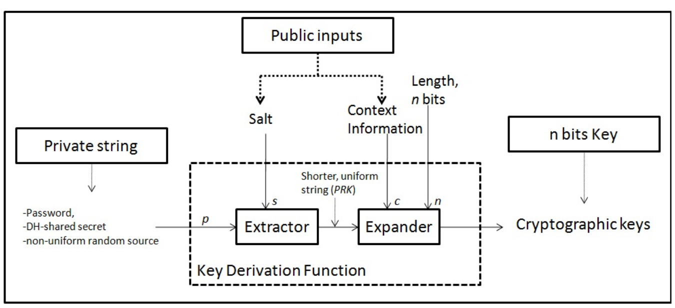

Fig. 2. Key derivation function - Two-phase KDF.

There is a two-phase KDF as shown in Fig. 2. It consists of two independent processes: the extractor function, Ext, and the expander function, Exp. The goal of the extractor and expander phases is to generate an output that is computationally indistinguishable from a random binary string of equal length [8]. This output is expressed as $K = \mathrm{Exp}\left(\mathrm{Ext}(p, s), c, n\right)$.

In the first phase, Ext extracts an amount of entropy from p and s as the input to produce an intermediate value. We denote the intermediate value as PRK. PRK is private, and the distribution of PRK is close to uniform.

Definition 1. *[Extractor] Let p and s are chosen uniform probability over $\{0,1\}^{pn}$ and $\{0,1\}^{sn}$ respectively.* $\mathrm{Ext} : \{0,1\}^{pn} \times \{0,1\}^{sn} \rightarrow \{0,1\}^{kn}$ *is a (t, ϵ)-extractor. The output is PRK is chosen with uniform probability from $\{0,1\}^{kn}$.*

The second phase involves using a standard expansion scheme, denoted as Exp, which takes the intermediate value PRK and c as inputs to derive one or more n-bit cryptographic keys.

Definition 2. *[Expander] A function* $\mathrm{Exp} : \{0,1\}^{kn} \times \{0,1\}^{cn} \rightarrow \{0,1\}^*$ *from a set $PRK \in \{0,1\}^{kn}$ mapping to an arbitrary length of string $\{0,1\}^*$ which should be indistinguishable from the random strings of the same length in time polynomial.*

III. Keystream Generator for Stream Ciphers

A stream cipher is a symmetric key system consisting of a keystream generator, plaintext, and XOR operation. The stream cipher performs both encryption and decryption using the same secret key (K). The keystream generator (KG) is utilized to generate an n-bit keystream (K_i) from an initial keying material.

The stream cipher's encryption process involves XOR ($\oplus$) between the plaintext (PT_i) and the keystream (K_i) to generate the ciphertext (CT_i). The decryption process consists of XORing the ciphertext with the identical keystream to produce the plaintext. It's important to note that P, K, and C have the same arbitrary length (n bits).

Stream ciphers are well-suited for real-time applications due to their low complexity and fast operation speed;

$$CT_i = PT_i \oplus K_i \ , \tag{1}$$

$$PT_i = CT_i \oplus K_i \ . \tag{2}$$

Stream cipher uses a KG to generate keystream for both encryption and decryption. The KG is a critical component of a stream cipher as the pseudorandomness of the keystream may protect the secrecy of the output of the stream cipher [17]. The KG outputs a keystream: $k_1, k_2, k_3, \ldots, k_i \in K$. The keystream is XORed with a stream of plaintext bits, $pt_1, pt_2, pt_3, \ldots, pt_i \in PT$, to produce the stream of ciphertext bits $ct_1, ct_2, ct_3, \ldots, ct_i \in PT$ [17]. The security of a stream cipher relies on its KG to generate pseudorandom keystream. For example, a keystream with an endless stream of zeros will produce a ciphertext that is equal to the plaintext. This will make the whole encryption useless. Thus, the KG should produce a pseudorandom bits to have perfect security.

There are two major processes in the generation of a keystream which are initialization and keystream generation process as shown in Fig. 3. In the initialization process, the inputs consist of a secret key and publicly known initial vector (IV). These inputs are mixed in the mixing process. The initialization process is to diffused pair of secret key and IV in order to harden the process for the attacker to find the correlation between the secret key and IV with it associate keystream. Upon the completion of mixing process, KG now is in internal state which is ready for keystream generation process. We denoted the internal state as IS and the size of internal state as r. The value of internal state is the output from mixing process. The output function takes the internal value to generate the keystream character. At the same time, the next state function utilizes the internal value to generate a new internal state. It should be noted that the keystream generation state update function may be different or similar to the initialisation state update function.

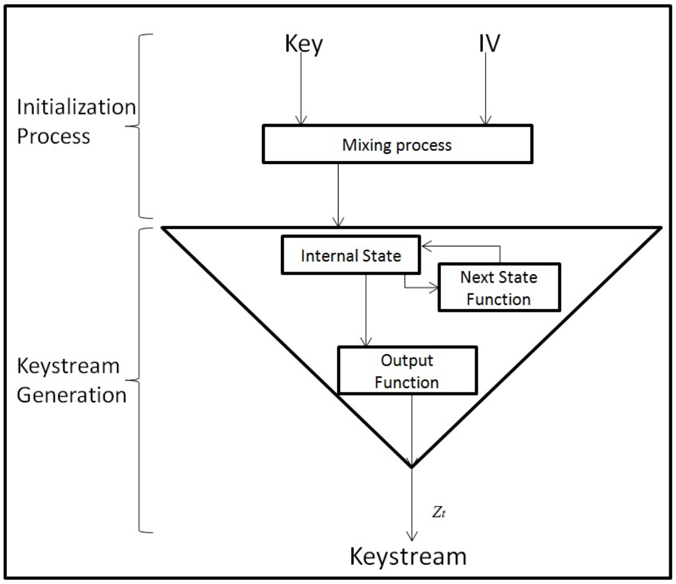

Fig. 3. Keystream generator [17].

Definition 3. *[Pseudorandom generator] There is no polynomial time algorithm that can distinguish between the output sequence of a keystream generator and a truly random sequence with probability significantly greater than $\frac{1}{2}$, where there length of these sequences are same, then the keystream generator is considered a pseudorandom generator that passes all statistical tests which are conducted within the polynomial-time framework [18].*

Definition 4. *[Pseudorandom generator] Let internal state has a set space over $\{0,1\}^{IS}$. A keystream generator is a pseudorandom generator (Definition 3) that mixing and diffusing the string from internal state, from which is mapping to an arbitrary length of pseudorandom keystream.*

In order to gain confidence that such keystream is pseudorandom, the keystream sequences should be Schneier [17] and Stallings [19]:

- Large period: Any infinite binary sequence produced by a deterministic process is ultimately periodic. If the same keystream is repeated in side of a cryptogram it may be possible to do a ciphertext only attack [20].
- High linear complexity: A short key is used as the input to the pseudorandom function such as keystream generators to produce the keystream. The linear complexity of a pseudorandom sequence is the length (L) in bits of the shortest linear feedback shift register which will produce this sequence. If $2L$ consecutive of keystream are known then the internal state of the generators can be found by using Berlekamp-Massey algorithm [21]. So, to avoid such an attack the linear complexity should be high.
- White noise: The keystream is trying to "appear" like a random sequence namely one-time pad [17], [19]. The measure of the closeness of sequence to a random sequence is called white noise characteristics.

A. *Existing* KDF *Proposals*

Existing KDF proposals mainly are two-phase design with extractor function and expander function. The cryptographic primitives to construct these extractor function and expander function can be block ciphers, hash functions and stream ciphers.

- Block ciphers [11]: Advanced encryption standard - CMAC (AES−CMAC) is the cryptographic primitive that has three different key length and one block size. The key length can be 128-bits, 192-bits and 256-bits. The block size is 128-bits. The extractor based on AES−CMAC can use the key length of 128-bits, 192-bits and 256-bits. But, the expander based on AES−CMAC is limited to the key length of 128-bits. Eq. (3) is the extractor based AES−CMAC The inputs for the extractor function is p and s. The p is divided equal size of 128-bits, we denote the block as D and $1 \leq i < \frac{pn}{128}$. $PRK_0 = 0^{128}$ and N can be 128-bits, 192-bits or 256-bits;
$$PRK_i \Leftarrow \text{AES}{-}\text{CMAC}_s(PRK_{i-1} \oplus D_i) \ . \quad (3)$$
Eq. (3) is the expander based AES−CMAC. The inputs for the expander function is PRK and c. The PRK is the output from expander which is 128-bits. The c is divided into block with each size of 128-bits, we denote the block as D and $1 \leq i < \frac{cn}{128}$. $K_0 = 0^{128}$ and N is 128-bits;
$$K_i \Leftarrow \text{AES}{-}\text{N}{-}\text{CMAC}_{PRK}(K_{i-1} \oplus D_i) \ . \quad (4)$$
The last block D_t requires addition subkey one or subkey two, we denote it as SK, $b \in \{1,2\}$. The algorithm subkey generation as show in Barker *et al.* [11];
$$K_i \Leftarrow \text{AES}{-}\text{N}{-}\text{CMAC}_{PRK}(K_{i-1} \oplus D_i \oplus SK_b) \ . \quad (5)$$
If $n > 128$, additional iterations are performed until the desired length is achieved. Extract the leftmost n bits from the output and discard any remaining bits.
- Hash function [8]: The propose KDF based on hash functions consists of extractor function and expander function. The hash function is using HMAC_{SHA} families.
Eq. (6) is the extractor function which generates PRK from the inputs of p and s. The output for this phase PRK is based on the length of hash digest (hn) of SHA families. The s is proposed has the same length as the hash digest of HMAC_{SHA}. If the length of s is shorter or longer, then s is hashed using the equivalent SHA function;
$$PRK \Leftarrow \text{HMAC}_{\text{SHA}}(s \oplus opad) \parallel \text{HMAC}_{\text{SHA}}(s \oplus ipad \parallel p) \ . \quad (6)$$

Eq. (7) is the expander function. The PRK and c are the inputs to the expander function. The expander produces n-bits of cryptographic key from these inputs. The cryptographic key is the concatenation string such that $K_1 \parallel K_2 \parallel \ldots \parallel K_{t-1}, 1 \leq i < t$, where $t = \lceil \frac{n}{hn} \rceil$. The first n bits are used as the cryptographic key and the remaining bits are discarded;

$$\begin{aligned} K_{i+1} \Leftarrow \mathrm{HMAC_{SHA}}(PRK \oplus opad) \parallel \\ \mathrm{HMAC_{SHA}}(PRK \oplus ipad) \parallel K_i \parallel c \parallel i \ . \end{aligned} \quad (7)$$

Noted that for both extractor function and expander function, the $opad$ is the outer padding with one block long hexadecimal of $0x5c5c \ldots 5c$ and the $ipad$ is the inner padding with one block long hexadecimal of $0x3636 \ldots 36$.

- Stream cipher [5]: The $\mathrm{SCKDF_2}$ uses pseudorandom KG to construct both extractor function and expander function. The input for the extractor is p and s, which results in the output sequence PRK. The length of s can be vary, but it must not exceed the length of pn or be null.
  Eq. (8) shows the extractor function which XOR p and s as the input to the KG. If the length of p is longer then the key and IV of KG, it repeats the loop;

$$\begin{aligned} PRK_1 \Leftarrow \mathrm{SCKDF_2}(p_1 \oplus s_1) \ , \\ PRK_i \Leftarrow \mathrm{SCKDF_2}(p_i \oplus s_i \oplus PRK_{i-1}) \ . \end{aligned} \quad (8)$$

  Eq. (9) shows the expander function. The length of c is arbitrary or null. If c is not null, it is divided into the total length of key and IV of KG. The c is XORed with PRK and c as the input to the KG. If the length of c is longer then the key and IV of KG, it repeats the loop. After completion the loop, the $\mathrm{SCKDF_2}$ generates the n-bits cryptographic key;

$$\begin{aligned} K_1 \Leftarrow \mathrm{SCKDF_2}(PRK_1 \oplus s_1) \ , \\ K_i \Leftarrow \mathrm{SCKDF_2}(K_{i-1} \oplus c_i) \ . \end{aligned} \quad (9)$$

IV. Improved KDF based on Stream Ciphers: $\mathrm{I{-}SCKDF_2}$

In this section, we modify the pseudorandom KG to construct two-phase $\mathrm{I{-}SCKDF_2}$. The input of the proposed extractor is p and s, which results in the output sequence of PRK. The block size of p is r and the r is considered as the length of the internal state. In $\mathrm{I{-}SCKDF_2}$ scheme, during the extractor phase, PRK is generated such that its length is equal with the size of the internal state of the pseudorandom KG used in the expander phase. Fig. 4 depicts our proposed $\mathrm{I{-}SCKDF_2}$ based extractor. The extractor process is as in Algorithm 1.

The output of extractor and arbitrary length of c are the inputs to the expander $\mathrm{I{-}SCKDF_2}$. The expander for $\mathrm{I{-}SCKDF_2}$ produces n-bits pseudorandom cryptographic key as shown in Fig. 5 and Algorithm 2.

V. Security Analysis

Here, we show a statistical test and a formal security proof for our propose $\mathrm{I{-}SCKDF_2}$ in section V-A and section V-B respectively.

A. Statistical Test

Dieharder test suite requires 1.25 mega bytes input to test either the string is pseudorandom or non-random. We use Diehارder security test to test the pseudorandomness of cryptographic key which is generated by $\mathrm{I{-}SCKDF_2}$. To generate these long cryptographic key, we use 500 strings of p, for each string only one bit changes, s and c are same. For each p with corresponding s and c as inputs to $\mathrm{I{-}SCKDF_2}$ and generates 2500 bytes of cryptographic key. All these cryptographic keys are concatenated as in total 1.25 mega bytes. The cryptographic key is converted to 32-bits unsigned integer. The result of security analysis for $\mathrm{I{-}SCKDF_2}$ is shown in Fig. 6.

The result shows that $\mathrm{I{-}SCKDF_2}$ passed all the security tests in Dieharder test suite. This indicates that the cryptographic key which is generated by $\mathrm{I{-}SCKDF_2}$ is pseudorandom. This would imply that and a polynomial-time A is unable to differentiate whether the given string is either the n-bit cryptographic key which is derived from secret string p or just an n-bit random string. The best A can differentiate the given string with only probability greater than $\frac{1}{2} + \epsilon$, where ϵ is negligible.

B. The Security of $\mathrm{I{-}SCKDF_2}$

Theorem 1. *Let* Ext *be a (t_X, ϵ_X)-extractor w.r.t to the secret string p and* Exp *a (t_P, q_P, ϵ_P)-secure variable-length-output pseudorandom function family, then the above extract-then-expand* KDF *scheme is ($\min\{t_X, t_P\}$, q_P, $\epsilon_X + \epsilon_P$)-secure w.r.t the secret string p [8].*

Generic $\mathrm{I{-}SCKDF_2}$ is two-phase KDF which consists of extractor function and expander function. Hence, it follows the Theorem 1. This mean that the generic $\mathrm{I{-}SCKDF_2}$ is a secure extract-then-expand KDF. The proof of Theorem 1 can be seen at the paper of Krawczyk [8].

In this section we review the properties of KG based on Definition 3 and Definition 4. We build a $\mathrm{I{-}SCKDF_2}$ with extractor function Ext and expander function Exp. Both functions are built using the KG which is from the family of pseudorandom KG that fulfil Definition 3 and Definition 4. Hence, it should be collision resistance such that $\mathrm{KG}(p) = \mathrm{KG}(p')$ where $p \neq p'$.

Lemma 1. *If* KG *is a secure pseudorandom* KG*, then extractor build from* KG *is a secure (t_T, ϵ_T)-extractor.*

Proof: If there is an adversary A_T that can break the Ext built from KG, then there is another adversary B_T who is able to break the security of pseudorandom generator. Hence, on the basis of A_T we build the B_T against the KG based extractor for the $\mathrm{I{-}SCKDF_2}$. Extractor generates PRK from p and s: $\mathrm{Ext}(p, s) \rightarrow PRK$. Once you get the PRK, you will be able to find p, such that $\mathrm{Ext}(PRK, s) \rightarrow p$. This means, A_T is able to find the p by invert the input and output of extractor in polynomial time t_T. This indicating the pseudorandom generator is not a one way function. This is contradicts our assumption of ideal KG. Hence, if KG is a secure pseudorandom generator, then extractor built from KG is a (t_T, ϵ_T)-extractor. ■

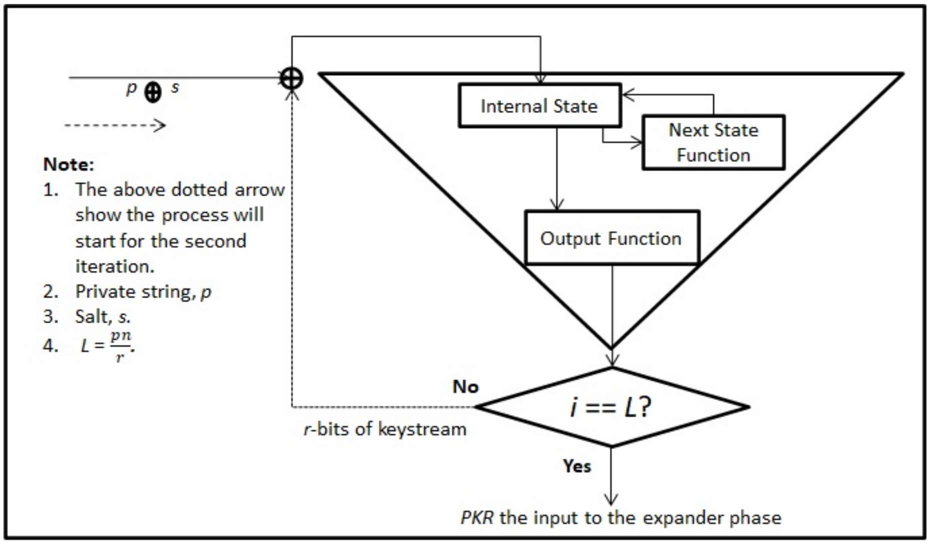

Fig. 4. Extractor of $\mathrm{I{-}SCKDF_2}$.

Algorithm 1 Extractor of $\mathrm{I{-}SCKDF_2}$

Require: Input: p, s, pn, sn, r.
Ensure: Split p into blocks such $L = \frac{pn}{r}$. L is the number of total blocks. The r is the size of internal state. D_i denote the i^{th} block of p. If the length of the last block, D_L, is shorter than r bits, the block is padded with '0's.

1: **if** s is null **then**
2:     Go to Step 8.
3: **else if** s is not null, $sn < pn$ **then**
4:     Divide the s into block, $J = \frac{sn}{r}$.
5:     Denote the i^{th} block of s as E_i. If the length of the last block E_i is shorter than r-bits, the blocks is padded with '0's.
6:     Perform XOR operation between the D_1 and E_1.
7: **end if**
8: **for** $i \leftarrow 1$ to L **do**
9:     **if** $i = L$ **then**
10:         The input of the pseudorandom KG is r-bits internal state.
11:         The pseudorandom KG produces r-bits of keystream.
12:         Go to Step 24.
13:     **else if** $i < L$ **then**
14:         The input of the pseudorandom KG is r-bits internal state.
15:         The pseudorandom KG produces r-bits of keystream.
16:         **if** $i <= J$ **then**
17:             Perform XOR operation between r-bits of keystream, D_{i+1} and E_{i+1}.
18:         **end if**
19:         **if** $i > J$ **then**
20:             Perform XOR operation between r-bits of keystream and D_{i+1}.
21:         **end if**
22:     **end if**
23: **end for**
24: **Output:** r-bits PRK.

Lemma 2. *If* KG *is a secure pseudorandom generator, then expander built from* KG *is a secure* (t_P, q_P, ϵ_P) *arbitrary length output pseudorandom* KG *function family.*

Proof: If there is an adversary A_P that can break the Exp built from the KG, then there is another adversary B_P that can break the pseudorandom generator. Hence, on the basis of A_P we build B_P against the KG based expander for the $\mathrm{I{-}SCKDF_2}$. PRK and c are the inputs to the expander, then produces n-bits of cryptographic key, such that $\mathrm{Exp}(PRK, c) \rightarrow K$. This means, A_P is able to distinguish the n-bits cryptographic key which is generated from two different string of c in polynomial time t_P, after q_P test queries, such that $\mathrm{Exp}(PRK, c) = \mathrm{Exp}(PRK, c')$ where $c \neq c'$. This indicating collision is happening, the pseudorandom generator is not a one way function. Again, this is contradicts our assumption of ideal KG. Hence, if KG is a secure pseudorandom generator, then expander built from KG is a secure (t_P, q_P,

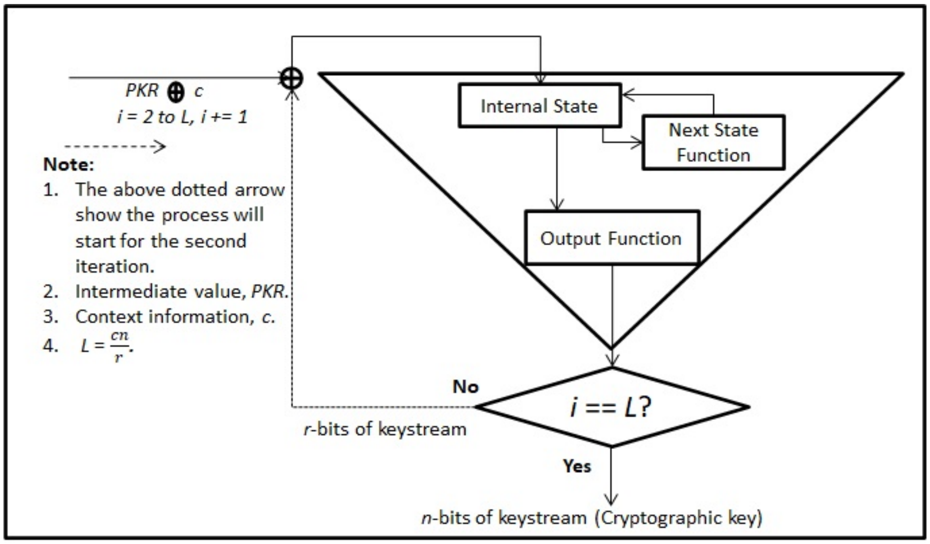

Fig. 5. Expander of $\mathrm{I{-}SCKDF_2}$.

Algorithm 2 Expander of $\mathrm{I{-}SCKDF_2}$.

Require: Input: PKR, c, cn, n.
1: **if** c is null **then**
2: The input for the pseudorandom KG is the r-bits of PRK.
3: The pseudorandom KG produces n-bit of keystream.
4: Go to Step 29.
5: **else if** c is not null **then**
6: Split c into blocks such that $L = \frac{cn}{r}$. L is the number of total blocks.
7: Denote the i^{th} block of c as D_i. If the length of the last block, D_L, is shorter than r bits, the block is padded with '0's.
8: XOR the r bits of PRK (from the extractor phase) with D_1 of c.
9: **if** $L = 1$ **then**
10: The input for the pseudorandom KG is the r-bits of PRK.
11: The pseudorandom KG produces n-bit of keystream.
12: Go to Step 29.
13: **else if** $L > 1$ **then**
14: The input for the pseudorandom KG is the r-bits of PRK.
15: The pseudorandom KG produces n-bit of keystream.
16: Go to Step 19.
17: **end if**
18: **end if**
19: **for** $i \leftarrow 2$ to L **do**
20: **if** $i = L$ **then**
21: Perform an XOR operation between the r-bits of keystream and D_i of c. The output is the input for the pseudorandom keystream generator.
22: The pseudorandom KG produces n-bit of keystream.
23: Go to Step 29.
24: **else if** $i < L$ **then**
25: Perform an XOR operation between the r-bits of keystream and D_i of c. The output is the input for the pseudorandom keystream generator.
26: The pseudorandom KG produces n-bit of keystream.
27: **end if**
28: **end for**
29: **Output:** n-bits cryptographic key.

ϵ_P) arbitrary length output pseudorandom KG function family. ■

Corollary 1. *The extract-then-expand* $\mathrm{I{-}SCKDF_2}$ *built from* KG *is* ($\min\{t_T, t_P\}$, q_P, $\epsilon_T + \epsilon_P$)*-secure w.r.t the secret string* p *if* KG *is a secure pseudorandom generator.*

Proof: This is an immediate result from Lemma 1, Lemma 2 and Theorem 1. ■

```
#=============================================================================#
#            dieharder version 3.31.1 Copyright 2003 Robert G. Brown          #
#=============================================================================#
   rng_name    |           filename             |rands/second|
     file_input|                 triviumtest.txt|  5.69e+06  |
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
   diehard_birthdays|   0|       100|     100|0.22765395|  PASSED
      diehard_operm5|   0|   1000000|     100|0.44061577|  PASSED
  diehard_rank_32x32|   0|     40000|     100|0.16152269|  PASSED
    diehard_rank_6x8|   0|    100000|     100|0.96725029|  PASSED
   diehard_bitstream|   0|   2097152|     100|0.73777773|  PASSED
        diehard_opso|   0|   2097152|     100|0.26088111|  PASSED
        diehard_oqso|   0|   2097152|     100|0.62091416|  PASSED
         diehard_dna|   0|   2097152|     100|0.36128116|  PASSED
diehard_count_1s_str|   0|    256000|     100|0.21853634|  PASSED
diehard_count_1s_byt|   0|    256000|     100|0.58628181|  PASSED
 diehard_parking_lot|   0|     12000|     100|0.67934334|  PASSED
    diehard_2dsphere|   2|      8000|     100|0.15889293|  PASSED
    diehard_3dsphere|   3|      4000|     100|0.61050166|  PASSED
     diehard_squeeze|   0|    100000|     100|0.50220585|  PASSED
        diehard_sums|   0|       100|     100|0.02178726|  PASSED
        diehard_runs|   0|    100000|     100|0.50234739|  PASSED
       diehard_craps|   0|    200000|     100|0.87970912|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.60785646|  PASSED
         sts_monobit|   1|    100000|     100|0.23980723|  PASSED
            sts_runs|   2|    100000|     100|0.83592643|  PASSED
          sts_serial|   1|    100000|     100|0.23980723|  PASSED
         rgb_bitdist|   1|    100000|     100|0.54809660|  PASSED
rgb_minimum_distance|   2|     10000|    1000|0.07758290|  PASSED
    rgb_permutations|   2|    100000|     100|0.30264181|  PASSED
      rgb_lagged_sum|   0|   1000000|     100|0.96261442|  PASSED
     rgb_kstest_test|   0|     10000|    1000|0.50961402|  PASSED
     dab_bytedistrib|   0|  51200000|       1|0.29473786|  PASSED
             dab_dct| 256|     50000|       1|0.75523077|  PASSED
        dab_filltree|  32|   5000000|       1|0.32434695|  PASSED
       dab_filltree2|   0|   5000000|       1|0.84601385|  PASSED
        dab_monobit2|  12|  65000000|       1|0.24737411|  PASSED
```

Fig. 6. Result of dieharder security test for $\mathrm{I{-}SCKDF_2}$.

VI. Performance Analysis and Discussion

In Chuah *et al.* [4], there are simulation results of KDF based on hash functions, block ciphers and stream ciphers. The execution time for KDF based on stream ciphers are running faster compare with KDF based on hash functions and block ciphers, especially Trivium based KDFs. Therefore, we only simulate $\mathrm{I{-}SCKDF_2}$ using Trivium. Table I is eight experiments parameters taken from Heer *et al.* [22] and Zhu *et al.* [23]. The parameters are measured with bytes.

TABLE I. The Parameter Experiments

Experiment	Parameters			
	n	p	s	c
1	64	128	8	32
2	192	128	8	32
3	64	256	8	32
4	192	256	8	32
5	64	128	null	64
6	192	128	null	64
7	64	256	null	64
8	192	256	null	64

All the experiments are simulated in a computer with the following specification: Intel Core i7, NVIDIA GEFORCE 940MX, 8GB RAM. Each experiment is executed 100 times, average execution time is recorded.

Fig. 7 depicts the simulation results. The result shows that the proposed $\mathrm{I{-}SCKDF_2}$ is relatively executes faster compared with existing $\mathrm{SCKDF_2}$. This is because the design of $\mathrm{I{-}SCKDF_2}$ reduces the number round of looping for the extractor function and expander function compares with $\mathrm{SCKDF_2}$. For example, $\mathrm{SCKDF_2}$ needs to perform sevan rounds in extractor (128 bytes of p, 8 bytes of s) and two rounds in expander (32 bytes of c) to produce 64 bytes of cryptographic key. While $\mathrm{I{-}SCKDF_2}$ needs to perform only four rounds in extractor and one round in expander for the same length of inputs and output. The results also indicate the propose $\mathrm{I{-}SCKDF_2}$ executes faster compare with KDF based on hash functions and block ciphers.

VII. Conclusions

We propose an improved KDF based on stream ciphers, denoted as $\mathrm{I{-}SCKDF_2}$. We have demonstrated that $\mathrm{I{-}SCKDF_2}$

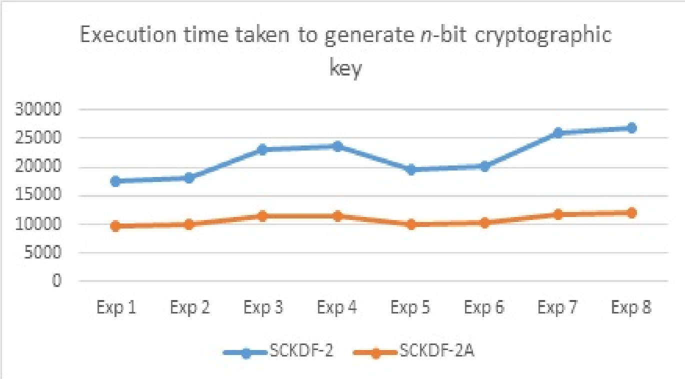

Fig. 7. Simulation results.

is theoretically secure, provided that the underlying KG used to construct $\mathrm{I-SCKDF_2}$ belongs to the family of pseudorandom functions. Therefore, careful selection of the KG type is essential for building KDF. To assess the pseudorandomness of the cryptographic key derived from $\mathrm{I-SCKDF_2}$, we utilized the Dieharder test suite. $\mathrm{I-SCKDF_2}$ successfully passed all the tests. Additionally, we conducted experiments to simulate the execution time of $\mathrm{I-SCKDF_2}$ across eight different parameter configurations, including p, s, c, and n. The results demonstrate that $\mathrm{I-SCKDF_2}$ executes more quickly in comparison to the existing KDF based on stream ciphers, denoted as $\mathrm{SCKDF_2}$.

ACKNOWLEDGMENT

The authors would like to thank Guangdong University of Science & Technology, China, Rabdan Academy, United Arab Emirates and Universiti Tun Hussein Onn Malaysia.

REFERENCES

[1] M. A. Mobarhan and S. Tian, *REPS-AKA5: A robust group-based authentication protocol for IOT applications in lte system*, Internet of Things, 22, 100700, 2023.

[2] S. Duttagupta, E. Marin, D. Singel ee and B. Preneel, *HAT: secure and practical key establishment for implantable medical devices*, Proceedings of the Thirteenth ACM Conference on Data and Application Security and Privacy, 2023.

[3] G. Fedrecheski, L. C. Costa, S. Afzal, J. M. Rabaey, R. D. Lopes and M. K. Zuffo, *A low-overhead approach for self-sovereign identity in IoT*, Internet of Things: 5th The Global IoT Summit, 2023.

[4] C. W. Chuah, E. Dawson and L. Simpson, *Key derivation function: the SCKDF scheme*, 28th IFIP TC 11 International Conference, 2013.

[5] C. W. Chuah, *Key derivation function based on stream ciphers*, Ph.D. dissertation, Queensland University of Technology, 2014.

[6] R. Housley, *Algorithm identifiers for the hmac-based extract-and-expand key derivation function (HKDF)*, Internet Engineering Task Force (IETF), Tech. Rep., 2019.

[7] J. M. Mcginthy and A. J. Michaels, *Further analysis of PRNG-based key derivation functions*, IEEE Access, 7, 95978–95986, 2019.

[8] H. Krawczyk, *Cryptographic extraction and key derivation: The HKDF scheme*, CRYPTO, 2010.

[9] C. W. Chuah, E. Dawson, J. M. González Nieto and L. Simpson, *A framework for security analysis of key derivation functions*, Information Security Practice and Experience: 8th International Conference, 2012.

[10] W. W. Koh and C. W. Chuah, *Robust security framework with bit-flipping attack and timing attack for key derivation functions*, IET Information Security, 14(5), 562–571, 2020.

[11] E. Barker, L. Chen and R. Davis, *SP 800-56C Rev. 2 recommendation for key-derivation methods in key-establishment schemes*, NIST Special Publication, 41, 2020.

[12] D. Watanabe, S. Furuya , H. Yoshida , K. Takaragi and B. Preneel, *A new keystream generator MUGI*, 9th International Workshop Fast Software Encryption, 2002.

[13] E. Dawson, A. Clark, J. Golic, W. Millan, L. Penna and L. Simpson, *The LILI-128 keystream generator*, NESSIE Workshop, 2000.

[14] F. Hauser, M. Häberle, M. Schmidt and M. Menth, *P4-IPSEC: Site-to-site and host-to-site vpn with IPSEC in p4-based SDN*, IEEE Access, 8, 139567–139586, 2020.

[15] L. Hornquist Astrand, L. Zhu, M. Cullen and G. Hudson, *RFC 8636: Public key cryptography for initial authentication in kerberos (PKINIT) algorithm agility*, 2019.

[16] L. Malina, G. Srivastava, P. Dzurenda, J. Hajny and R. Fujdiak, *A secure publish/subscribe protocol for internet of things*, Proceedings of the 14th international conference on availability, reliability and security, 2019.

[17] B. Schneier, *Applied cryptography: protocols, algorithms, and source code in C*, 2015.

[18] A. Menezes, P. Van Oorschot and S. Vanstone, *Handbook of applied cryptography*, 1997.

[19] W. Stallings, *Cryptography and Network Security: Principles and Practices, Fourth Edition*, 2006.

[20] E. Dawson and L. Nielsen, *Automated cryptanalysis of XOR plaintext strings*, Cryptologia, 20(2), 165–181, 1996.

[21] J. Massey, *Shift-register synthesis and BCH decoding*, IEEE Transactions on Information Theory, 15(1), 122–127, 1969.

[22] T. Heer, P. Jokela, T. Henderson and R. Moskowitz, *Host identity protocol version 2 (HIPv2)*, Internet Engineering Task Force (IETF), Tech. Rep., 2012.

[23] L. Zhu, M. Wasserman and L. Astrand, *PKINIT algorithm agility*, Internet Engineering Task Force (IETF), Tech. Rep., 2012.