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

Chai Wen Chuah1, Janaka Alawatugoda2, Nureize Arbaiy3
Guangdong University of Science & Technology, Dongguang, Guangzhou, China1
Research & Innovation Centers Division, Rabdan Academy Abu Dhabi, UAE2,
Institute for Integrated and Intelligent Systems, Griffith University, Nathan, Queensland, Australia2
Faculty of Computer Science & Information Technology, Universiti Tun Hussein Onn Malaysia, Parit Raja, Malaysia3

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 (SCKDF2). 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 1−SCKDF2. We simulate instances for the proposed 1−SCKDF2 using Trivium. As a result, 1−SCKDF2 has a lower execution time compared to the existing SCKDF2. The results show that the execution time taken by 1−SCKDF2 to generate an n-bit cryptographic key is almost 50 percent lower than SCKDF2. The security of 1−SCKDF2 passed all the security tests in the Dieharder test tool. It has been proven that the proposed 1−SCKDF2 is secure, and the simulation time is faster compared to SCKDF2.

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 used as a password, a random seed value from some entropy source, or output value such as a 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 (SCKDF2) using the keystream generator (KG) [5], [4]. The authors incorporated KG into the KDF design 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 SCKDF2, 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 SCKDF2, 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 SCKDF2 is similar to that of block cipher-based KDFs and hash function-based KDFs. In terms of execution time, SCKDF2 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 1−SCKDF2.

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.

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 \( \text{Exp} : \{0,1\}^{kn} \times \{0,1\}^{cn} \rightarrow \{0,1\}^* \) 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 \( (n \text{ bits}) \).

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

\[
CT_i = PT_i \oplus K_i ,
\]

\[
PT_i = CT_i \oplus K_i .
\]

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 its associated 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.

**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 \( \{0,1\}^L \). 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. 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 \( 1 \leq i < \frac{m}{128} \). \( PRK_0 = 0^{128} \) and \( N \) can be 128-bits, 192-bits or 256-bits;

  \[
  PRK_i \leftarrow AES-CMAC_s(PRK_{i-1} \oplus D_i). \quad (3)
  \]

  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 \( 1 \leq i < \frac{m}{128} \). \( K_0 = 0^{128} \) and \( N \) is 128-bits;

  \[
  K_i \leftarrow AES-N-CMAC_{PRK}(K_{i-1} \oplus D_i). \quad (4)
  \]

  The last block \( D_i \) requires addition subkey one or subkey two, we denote it as \( SK_b \), \( b \in \{1,2\} \). The algorithm subkey generation as show in Barker et al. [11];

  \[
  K_i \leftarrow AES-N-CMAC_{PRK}(K_{i-1} \oplus D_i \oplus SK_b). \quad (5)
  \]

  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 (\( ln \)) 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:

  \[
  PRK \leftarrow HMAC_{SHA}(s \oplus opad) \parallel \ HAMAC_{SHA}(s \oplus ipad \parallel p). \quad (6)
  \]
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 $t = \lceil \frac{m}{n} \rceil$. The first $n$ bits are used as the cryptographic key and the remaining bits are discarded:

$$K_{i+1} \leftarrow \text{HMAC}_{SHA}(PRK \oplus \text{opad}) \parallel \text{HMAC}_{SHA}(PRK \oplus \text{ipad}) \parallel K_i \parallel c \parallel i.$$  \hspace{1cm} (7)

Noted that for both extractor function and expander function, the $\text{opad}$ is the outer padding with one block long hexadecimal of $0e5c5c\ldots5c$ and the $\text{ipad}$ is the inner padding with one block long hexadecimal of $0e3636\ldots36$.

- **Stream cipher [5]:** The SCKDF2 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:

$$PRK_1 \leftarrow \text{SCKDF}_2(p_1 \oplus s_1),$$

$$PRK_i \leftarrow \text{SCKDF}_2(p_1 \oplus s_1 \oplus PRK_{i-1}).$$  \hspace{1cm} (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 SCKDF2 generates the $n$-bits cryptographic key:

$$K_1 \leftarrow \text{SCKDF}_2(PRK_1 \oplus s_1),$$

$$K_i \leftarrow \text{SCKDF}_2(K_{i-1} \oplus c_i).$$  \hspace{1cm} (9)

IV. IMPROVED KDF BASED ON STREAM CIPHERS: I−SCKDF2

In this section, we modify the pseudorandom KG to construct two-phase I−SCKDF2. 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 I−SCKDF2 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 I−SCKDF2 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 I−SCKDF2. The expander for I−SCKDF2 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 I−SCKDF2 in section V-A and section V-B respectively.
**Algorithm 1** Extractor of $I$–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 $(t_P, q_P, \epsilon_P)$ 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 $I$–SCKDF$_2$. $PRK$ and $c$ are the inputs to the expander, then produces $n$-bits of cryptographic key, such that $\text{Exp}(PRK, c) \rightarrow K$. 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 $\text{Exp}(PRK, c) = \text{Exp}(PRK, c')$ 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 $(t_P, q_P,$
Algorithm 2 Expander of I−SCKDF₂.

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.

\( \epsilon_P \) arbitrary length output pseudorandom KG function family. \( p \) if KG is a secure pseudorandom generator.

**Corollary 1.** The extract-then-expand I−SCKDF₂ built from KG is \( (\min\{t_T, t_P\}, q_P, \epsilon_T + \epsilon_P) \)-secure w.r.t the secret string

**Proof:** This is an immediate result from Lemma 1, Lemma 2 and Theorem 1.
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 compared with KDF based on hash functions and block ciphers, especially Trivium based KDFs. Therefore, we only simulate I–SCKDF\textsubscript{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>
<thead>
<tr>
<th>Experiment</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>n</td>
</tr>
<tr>
<td>1</td>
<td>64</td>
</tr>
<tr>
<td>2</td>
<td>192</td>
</tr>
<tr>
<td>3</td>
<td>64</td>
</tr>
<tr>
<td>4</td>
<td>192</td>
</tr>
<tr>
<td>5</td>
<td>64</td>
</tr>
<tr>
<td>6</td>
<td>192</td>
</tr>
<tr>
<td>7</td>
<td>64</td>
</tr>
<tr>
<td>8</td>
<td>192</td>
</tr>
</tbody>
</table>

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 I–SCKDF\textsubscript{2} is relatively executes faster compared with existing SCKDF\textsubscript{2}. This is because the design of I–SCKDF\textsubscript{2} reduces the number round of looping for the extractor function and expander function compares with SCKDF\textsubscript{2}. For example, SCKDF\textsubscript{2} needs to perform seven 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 I–SCKDF\textsubscript{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 I–SCKDF\textsubscript{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 I–SCKDF\textsubscript{2}. We have demonstrated that I–SCKDF\textsubscript{2}...
is theoretically secure, provided that the underlying KG used to construct I–SCKDF2 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 I–SCKDF2, we utilized the Dieharder test suite. I–SCKDF2 successfully passed all the tests. Additionally, we conducted experiments to simulate the execution time of I–SCKDF2 across eight different parameter configurations, including p, s, c, and n. The results demonstrate that I–SCKDF2 executes more quickly in comparison to the existing KDF based on stream ciphers, denoted as SCKDF2.

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