Key Derivation Function: The SCKDF Scheme

Chai Wen Chuah, Edward Dawson, Leonie Simpson

Key Derivation Function: The SCKDF Scheme

Chuah Chai Wen, Edward Dawson, and Leonie Simpson

Queensland University of Technology,
chaiwen.chuah, e.dawson, lr.simpson@qut.edu.au

Abstract

A key derivation function is used to generate one or more cryptographic keys from a private (secret) input value. This paper proposes a new method for constructing a generic stream cipher based key derivation function. We show that our proposed key derivation function based on stream ciphers is secure if the underlying stream cipher is secure. We simulate instances of this stream cipher based key derivation function using three eStream finalist: Trivium, Sosemanuk and Rabbit. The simulation results show these stream cipher based key derivation functions offer efficiency advantages over the more commonly used key derivation functions based on block ciphers and hash functions.

Keywords: Key derivation function, cryptographic key, stream cipher.

1 Introduction

A key derivation function (*KDF*) is a basic component of a cryptographic system. It is used to generate one or more cryptographic keys from a private input string; such as a password, Diffie-Hellman (DH) shared secret or non-uniformly random source material [12,13,16,24]. The derived cryptographic keys are then used for maintaining information security and protecting electronic data when it is stored or transmitted. To prevent an adversary gaining any useful information about the private string, it is essential that the cryptographic keys generated by the *KDF* are computationally indistinguishable from a binary random string [15]. That is, given a binary string the adversary may not be able to distinguish whether the string is the cryptographic key generated by the *KDF* or a random string of the same length.

For *KDF*s, the inputs consist of a private string and a public string. The public string consists of a random string or a concatenation of counter, session identifier or the identities of communicating parties. Where the cryptographic keys are obtained directly from the inputs without any intermediate step, this is refered to as a single phase *KDF* (see for example [1,7,14,23]). A more recent *KDF* design trend is the two phase *KDF* [9,15], where the phases consist of an extractor and an expander. The inputs to the extractor are the private string and a non-secret random string, while the inputs to the expander are the output from the extractor and the context information. In this design, the extractor and expander are two independent sub-functions, which can be designed and analysed separately. This permits mixing and matching of different types of extractor and

expander functions to form good extract-then-expand *KDF* proposals, in terms of both security and/or performance.

Many existing *KDF* proposals (both single and two phase) are composed using either hash functions or block ciphers. Both hash functions and block ciphers divide the input into a series of equal-sized blocks, with some padding necessary if the last block input is not of the appropriate length. The input blocks are processed in sequence with a one-way compression function, and the output is a fixed block size. A *KDF* should be able to generate cryptographic keys of arbitrary length. Where the required length is not a multiple of the output block size, modification is necessary. Generally, the approach is to produce multiple output blocks until the required length has been obtained and to discard any bits in excess of the required length. This may be regarded as wasteful.

*KDF*s are widely used in Internet protocols [12,13,16,24]. Mobile devices like smartphones are increasingly used to access the Internet. These devices have low processing power, so efficiency is important. There is increasing interest in the design of more efficient *KDF*s for use in mobile devices or similar applications.

Stream ciphers are often used for encryption in resource constrained devices due to their speed and simplicity of implementation in hardware. Hash functions and block ciphers are often slower and require more resources than stream ciphers. Thus, a *KDF* based on stream ciphers may provide a more efficient alternative to the current block cipher or hash function based *KDF*s.

This paper proposes a new secure and efficient *KDF* based on the keystream generator of a stream cipher. We refer to this proposal as *SCKDF*. We present a generic model for a stream cipher based *KDF* which is secure if the underlying stream cipher is secure. We implement this generic *SCKDF* for three stream ciphers proposals: Trivium [6], Sosemanuk [4] and Rabbit [5]. The results show that the *SCKDF* is executes faster compared to existing *KDF*s based on hash functions and block ciphers.

This paper is organized as follows. We provide our notation and some background information on *KDF*s in Section 2. Section 3 reviews the properties of keystream generators. Our new proposal, a generic *SCKDF*, is presented in Section 4. The security proof for this construction is given in Section 5. Performance measurements to permit comparison of stream cipher, block cipher and hash function based *KDF*s for common applications scenarios are given in Section 6.

2 Backgroud for KDFs

Before we present the formal definition of a key derivation function, we recall the notion of min-entropy, as presented in [15].

Definition 1 *(min-entropy)[15]. A probability distribution $\mathcal{X}$ has min-entropy (at least) m if for all a in the support of $\mathcal{X}$ and for random variable X drawn according to $\mathcal{X}$, Prob(X=a) $\leq 2^{-m}$.*

In our case, X is the random variable represented by the private string and $\mathcal{X}$ is the probability distribution for possible values of X.

Definition 2 *(Key derivation function). A key derivation function is defined as:* $K \leftarrow KDF(p, s, c, n)$*, where*

- p *is a private string, which is chosen from the space of all possible private strings PSPACE. We denote the length of* p *as* pl*.*
- s *is a salt, a public random string chosen from the salt space SSPACE. We denote the length of* s *as* sl*;*
- c *is a public context string chosen from a context space CSPACE. The length of* c *is* cl*.*
- n *is a positive integer that indicates the number of bits to be produced by the KDF;*
- K *is the derived* n *bit cryptographic key.*

The basic operation of a KDF is to transform the secret p *and the public inputs (*s *and/or* c*) into an* n *bit string which can be used as a cryptographic key.*

Note that all inputs are publicly known, except for the private string p. The salt is uniformly random and is used to create a large set of possible keys corresponding to a given p [23]. Context information is arbitrary but application specific data; for example, a session identifier or the identities of communicating parties [2,3]. Similar definition are used in other *KDF* proposals. See for example [1,7,14,23,9] and [15].

Definition 3 *A KDF function is called* (t, q, ϵ)*m-entropy secure if it is* (t, q, ϵ)*-secure with respect to all (computational)m-entropy sources, where the derived cryptographic key of the KDF from m-entropy sources is computationally indistinguishable from a binary random string. That is, when the adversary is given a limited number of queries (*q *in total) to polynomial time algorithm* t*, the adversary can distinguish between the cryptographic key derived from the KDF or a random string of the same length with negligible probability* ϵ*[15].*

2.1 Single Phase KDFs

A single phase *KDF* uses a pseudorandom function that takes the private input and public inputs and transforms these inputs directly into one or more variable length computationally indistinguishable cryptographic keys. Figure 1 depicts a single phase *KDF*.

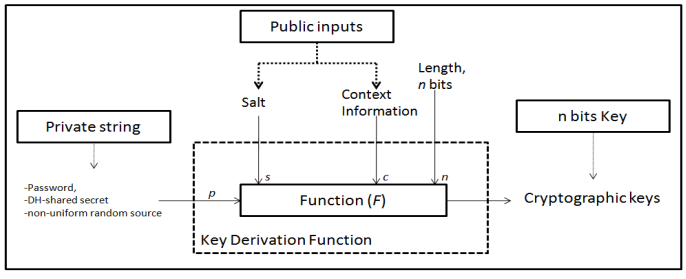

Fig. 1. Single phase model for KDFs.

Definition 4 *(Single phase KDF). A KDF is a function of* $F : \{0,1\}^{pl} \times \{0,1\}^{sl} \times \{0,1\}^{cl} \rightarrow \{0,1\}^{*}$ *from a set* $p \in_R$ *PSPACE mapping to an arbitrary length of string* $\{0,1\}^{*}$. *The string should be indistinguishable from random strings of the same length in polynomial time.*

2.2 Two Phase KDFs

A two phase *KDF* is the composition of two subfunctions: an extractor (*Ext*) and an expander (*Exp*). Note that the output of the extractor is an input to the expander as shown in Figure 2. The typical construction of a two-phase *KDF* is: $KDF(p, s, c, n) = Exp\ (\{Ext\ (p\ ,\ s)\},\ c,\ n)$. We discuss each phase below.

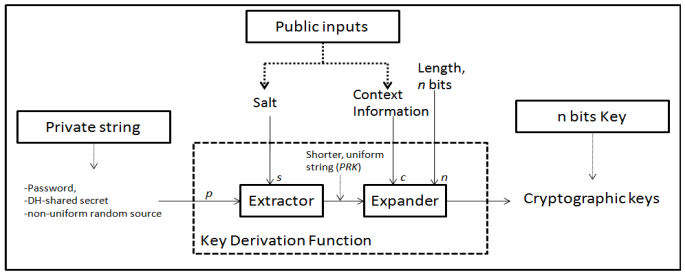

Fig. 2. Extract-then-expand model for KDFs.

Extractor The aim of the extractor is to transform the private input p into close to uniformly random output, which we denote as *PRK*. In this research, we generate the *PRK* from p using a computational extractor.

Definition 5 *(Computational extractor)[15]. Let PSPACE and SSPACE be set spaces of* $\{0,1\}^{pl}$ *and* $\{0,1\}^{sl}$ *respectively. A function* $Ext : \{0,1\}^{pl} \times \{0,1\}^{sl} \rightarrow \{0,1\}^{kl}$ *is called a* (t_X, ϵ_X)*-computational extractor if an adversary A running in polynomial time* t_X *can distinguish between PRK (derived from p) or a random string of the same length, with probability not larger than* $(\frac{1}{2} + \epsilon_X)$ *where p is chosen from* $\{0,1\}^{pl}$ *and s chosen from* $\{0,1\}^{sl}$. *If Ext is a* (t_X, ϵ_X)*-computational extractor with min-entropy m we call it a* (m, t_X, ϵ_X)*-computational extractor.*

Expander The expander takes the arbitrary length output from the extractor phase, *PRK*, as an input together with other public input material (context information) and generates one or more arbitrary length computationally indistinguishable cryptographic key(s).

Definition 6 *(Expander)[15]. An expander is a* (t_Y, q_Y, ϵ_Y)*-secure variable-length-output pseudorandom function family if an adversary A running in polynomial time* t_Y *and making at most* q_Y *queries to the expander can distinguish the cryptographic key is generated by the expander or a random string of the same length with probability not larger than* $(\frac{1}{2} + \epsilon_Y)$.

2.3 The Security of *KDF*

The major security goal for a *KDF* is that the cryptographic keys generated by the *KDF* are indistinguishable from truly random binary strings of the same length, even when the public inputs are provided to the adversary. We follow the approach of Krawczyk [15] and define the *KDF* security through a distinguishing game played between a challenger C and an adversary A in polynomial time algorithm t. The *KDF* is considered secure if no A can win the distinguishing game with probability significantly greater than the probability of winning by guessing randomly.

The game runs in three major stages: the learning stage, the challenge stage and the adaptive stage as shown in Table 1. During the learning stage, A is allowed to interact with C to demand cryptographic keys corresponding to A's choice of public input c with p and s chosen by C. In this game, p is secret known only to C, while s is known by A. At the challenge stage, A is provided a challenge output K'. After receiving the challenge output, A is in the adaptive stage. A can continue the same process as in the learning stage, subject to the choice of public input (c) being different from the public input chosen in the challenge stage. Lastly, A has to distinguish whether the challenge output is the derived cryptographic key from the *KDF* or just a random string. We describe a *KDF* for which A cannot win this game with the probability not larger than $(\frac{1}{2} + \epsilon)$ as CCS-secure.

Definition 7 *(CCS-secure) The KDF is (t, q, ϵ) CCS-secure if for all probabilistic polynomial-time t adversaries A can make at most $q < |CSPACE|$ queries to the KDF who can win the following indistinguishability game with probability not larger than $(\frac{1}{2} + \epsilon)$.*

<table>
<tr><td>Learning stage</td><td>1. C chooses $p \leftarrow PSPACE$.
2. C chooses $s \xleftarrow{R} SSPACE$.
3. A is provided with the value s.
4. For $i = 1, \ldots, q' \leq q$,</td><td>

(4.1) A chooses $c_i \leftarrow CSPACE$.
(4.2) C computes $K_i = F(p, s, c_i, n)$.
(4.3)A is provided the derived cryptographic key, K_i.</td></tr>
<tr><td>Challenge stage</td><td>1. A chooses $c \leftarrow CSPACE$
(subject to restriction $ctx \notin c_i, \ldots, c'_q$).
2. C chooses $b \xleftarrow{R} \{0, 1\}$.

5. C sends K' to A.</td><td>

(2.1) If $b = 0$, C outputs $K' = F(p, s, c, n)$,
(2.2) else C outputs $K' \xleftarrow{R} \{0,1\}^n$.</td></tr>
<tr><td>Adaptive stage</td><td colspan="2">1. Step 4 in Learning stage is repeated for up to $q - q'$ queries (subject to restriction $c_i \neq c$).
2. A outputs $b' = 0$, if $\mathcal{A}$ believes that K' is cryptographic key, else outputs $b' = 1$.</td></tr>
<tr><td colspan="3">A wins the game if $b' = b$.</td></tr>
</table>

Table 1. CCS-secure.

In [15], Krawczyk showed the condition under which a two-phase *KDF* can be considered CCS-secure as follows in Theorem 1.

Theorem 1 *Let Ext be a (t_X, ϵ_X)-computational extractor with the respect to the private string p and Exp a (t_Y, q_Y, ϵ_Y)-secure variable-length-output pseudorandom function family, then the above extract-then-expand KDF scheme is (min{t_X, t_Y}, q_Y, $\epsilon_X+\epsilon_Y$)-CCS secure with the respect the private string p [15].*

2.4 Existing KDF Proposals

To date, both single phase and two-phase proposals of *KDF*s have been based on cryptographic hash functions[15] and block ciphers[8]. Hash functions are widely used for data authentication and block ciphers for data confidentiality. We describe two specific well-known two-phase *KDF* proposals, one based on hash functions and the other one based on block ciphers, in the remainder of this section.

Hash Functions: In [15], Krawczyk formalized a *KDF* using HMAC-SHA families (HKDF) and proved that HKDF is CCS-secure. The proposed HKDF consists of a computational extractor and a pseudorandom expander. The extractor function is $PRK \Leftarrow Ext_p(s) : F((s \oplus opad)\|F((s \oplus ipad)||p))$, where F denotes a hash function, $\oplus$ denoter exclusive or (XOR), and $\|$ denoter concatenation.

The expander phase of the HKDF functions is $Exp_{PRK}(c, n)$: $K(1) \Leftarrow F(PRK \oplus opad)\|F((PRK \oplus ipad)||c||0)$ and F is the hash function. If $n > fl$, two or more iterations are necessary until the required length has been obtained: $K(i+1) \Leftarrow F((PRK \oplus opad)\|F((PRK \oplus ipad)||K(i)||c||i)), 1 \leq i < t$, where $t = \lceil \frac{n}{fl} \rceil$. The first n bits of the outputs $K(1)||K(2)||\ldots||K(t-1)$ are used as the cryptographic key, and the remaining bits are discarded.

Block Ciphers: The AES-CMAC based *KDF* is described in NIST SP800-108 [8]. The AES block cipher supports key sizes of 128, 192 and 256 bit and has an output size of 128 bits. The AES-CMAC based extractor can be either AES-128, 192 or 256, but the expander is fixed to use AES-128.

During the extraction phase, the input p is broken up into 128 bit blocks denoted as D_i, $1 \leq i < t$, $t = \lceil \frac{pl}{128} \rceil$; and the salt s is used as the AES key. The D_i are processed sequentially by using AES. The process is $PRK_i = F_s(PRK_{i-1} \oplus D_i)$, where F is AES (128 or 192 or 256), $1 \leq i < t$ and $PRK_0 = 0^{128}$.

During the expansion, the PRK and c are the inputs to the expander phase, where c is broken into D_i blocks, $1 \leq i < t$, $t = \lceil \frac{cl}{128} \rceil$. PRK is used as the AES key. The extractor function is as below: $K(i) \Leftarrow F_{PRK}(K_{i-1} \oplus D_i)$ where F is AES-128, $1 \leq i < t$ and $K(0) = 0^{128}$. The last block of operation is $K(t) = F_{PRK}(K_{t-1} \oplus D_t \oplus K_b)$, $b \in \{1, 2\}$. If $n > 128$, more iterations are performed until the length of output obtained exceeds the required length. Then, the left-most n bits of the output are used as the cryptographic key and the remaining bits are discarded.

3 Keystream Generator

A pseudorandom keystream generator is one of the components of a stream cipher. The inputs to the pseudorandom keystream generator are a secret key and a known initial value (IV) and the output is a pseudorandom keystream as shown in Figure 3. The aim of the secure stream cipher is to use a pseudorandom keystream generator which approximate an ideal pseudorandom as defined in Definition 8 and Definition 9. Note that although the keystream output can be produced in bits, bytes or words, we consider the keystream as a binary string: $Z_1, Z_2, \ldots, Z_t$, where $Z_i \in \{0,1\}, i = 1, 2, \ldots, t$.

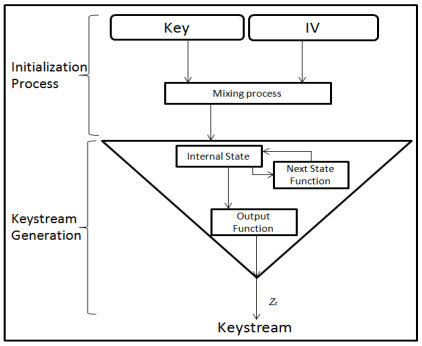

Fig. 3. Keystream Generator [22]

Definition 8 *(Keystream generator). Let KEYSPACE, IVSPACE, ISSPACE, ZSPACE be a set space over* $\{0,1\}^k$, $\{0,1\}^i$, $\{0,1\}^{is}$ *and* $\{0,1\}^*$ *respectively. A keystream generator is a pseudorandom generator (Definition 9) that takes the inputs key and IV and generates arbitrary length of keystream. Pseudorandom keystream generator:* $\{0,1\}^k \times \{0,1\}^i \rightarrow \{0,1\}^{is} \rightarrow \{0,1\}^*$.

Definition 9 *(Ideal pseudorandom generator) [17]. An ideal pseudorandom generator is said to pass all polynomial-time statistical tests if no polynomial-time algorithm can correctly distinguish between an output sequence of the generator and a truly random sequence of the same length with probability not larger than* $\frac{1}{2} + \epsilon$, *for some negligible value* ϵ.

4 Stream Cipher Based KDF

Our proposed *SCKDF* is a two-phase model where both the extractor and the expander are based on keystream generators for stream ciphers. For stream ciphers, the pseudorandom keystream generator takes two inputs: a key and an IV. In our *SCKDF*, we replace the pair of inputs to the pseudorandom keystream generator (key, IV) with the input pair (p, s) for the extractor phase and the input pair (PRK, c) for the expander phase. Detailed descriptions and specification for these phases are as follows.

4.1 Extractor

In this section, we propose an extractor based on the pseudorandom keystream generator for a stream cipher. The extractor takes p and s as the inputs and produces an output sequence PRK. Let v and w denote the key size and IV size respectively, for the stream cipher. Similarly, let r denote the key size of the stream cipher in the expander phase. (Note that is possible the same stream cipher may be used for both extractor phase and expander phase, but this is not necessary.) Figure 4 depicts our proposed stream cipher based extractor. The extractor process is as follows.

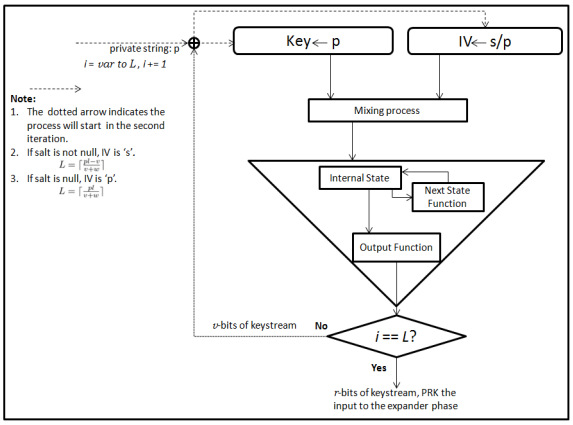

Fig. 4. Extractor based on stream ciphers

1. **Input:** p, s, pl, sl, r.
2. **Process:**
   (a) **If s is null**.
      i. Divide private string p into blocks, where each block is of length of $v + w$. Let D_i denote the i^{th} block of p. The total number of blocks is $L = \lceil \frac{pl}{v+w} \rceil$. If the length of the last block D_L is less than $v + w$ bits, the block is padded with '0's. Go to Step 2c.
   (b) **Else (if s is not null)**. Public string s is proposed to have same length as w of pseudorandom keystream generator. However, if $sl < w$, set the remaining bits with '0's.
      i. **If** $pl < v$.
         A. Pad the remaining bits of p with '0's.
         B. Use the p as the key and s as the IV for the pseudorandom keystream generator.
         C. Generate r bits of keystream.
         D. **Proceed** to Step 3.
      ii. **Else, if** $pl > v$.
         A. Use the first v bits of p as the key and s as the IV for the pseudorandom keystream generator.

B. Generate $v + w$ bits of keystream.
C. The remaining bits of p are divided into blocks, where each block is of length of $v + w$. Let D_i denote the i^{th} block of p. The total number of blocks is $L = \lceil \frac{pl-v}{v+w} \rceil$. If the length of the last block D_L is less than $v + w$ bits, the block is padded with '0's.
D. XOR the $v + w$ bits of keystream produced in Step 2(b)iiB with D_1 of p.
E. **Go** to Step 2c.

(c) For $i = 1$ to L, do the following:
i. **If** $i = L$. Use the first v bits of D_i as the key and remaining w bits of D_i as the IV and generate r bits of keystream. **Proceed** to Step 3.
ii. **Else, if** $i > L$.
A. Use the first v bits of D_i as the key and remaining w bits of D_i as the IV for the pseudorandom keystream generator.
B. Generate $v + w$ bits of keystream.
C. The $v + w$ bits of keystream is XORed with D_{i+1} of p.
D. $i := i + 1$.

3. **Output:**
– An r-bit string, denoted PRK.

4.2 Expander

In this section, we describe a stream cipher based expander. This function takes inputs the output of extractor phase PRK, together with an arbitrary length binary string c, the context information. The expander output is a pseudorandom binary string. Let v and w denote the key size and IV size respectively for the stream cipher. Figure 5 illustrates our proposed stream cipher based expander. The expander process is as follows.

1. **Input:** PRK, c, cl, and n.
– If c is null, then c is padded with '0's, $cl = w$.
2. **Process:**
(a) The context information c is divided into blocks, where each block size is of length of w. Let D_i denote the i^{th} block of c. The total number of blocks is $L = \lceil \frac{cl}{w} \rceil$. If the length of the last block D_L is less than w bits, the block is padded with '0's.
i. **If** $L = 1$.
A. Use PRK (from the extractor phase) as the key and c as the IV for the pseudorandom keystream generator.
B. Generate n bits of keystream.
C. **Proceed** to Step 3.
ii. **Else, if** $L > 1$.
A. Use PRK (from the extractor phase) as the key and the first block D_1 as the IV for the pseudorandom keystream generator.
B. Generate v bits of keystream.

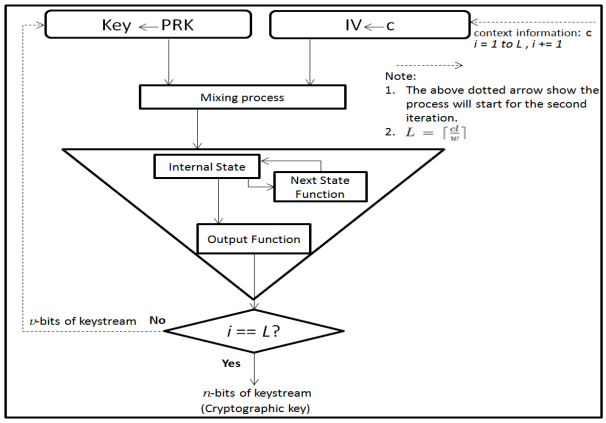

Fig. 5. Expander based on stream ciphers

C. **Proceed** to Step 2b.

(b) For $i = 2$ to L, do the following:

i. **If** $i = L$.

A. Use v bits of keystream as the key and the D_i as the IV for the pseudorandom keystream generator.

B. Generate n bits of keystream.

C. **Proceed** to Step 3.

ii. **Else, if** $i > L$.

A. Use v bits of keystream as the key and the D_i the IV for the pseudorandom keystream generator.

B. Generate v bits of keystream.

C. $i := i + 1$.

3. **Output:** An n-bit binary string suitable for use as a cryptographic key.

5 The Security of SCKDF

Our proposed two-phase (extract then expand) *SCKDF* makes use of the pseudorandom keystream generator of a stream cipher in each phase. We assume this is an ideal keystream generator (satisfying Definition 8 and Definition 9 in Section 3). Note that, a similar assumption of an ideal primitive was also made by Krawczyk in proving Theorem 1 in [15].

For such a pseudorandom keystream generator the *SCKDF* proposed in Section 4 can be considered as CCS-secure.

Theorem 2 *Let pseudorandom keystream generator be a keystream generator from a family of pseudorandom keystream generator which satisfy Definition 8 and Definition 9. If an extract-then-expand SCKDF is built from the pseudorandom keystream generator, then the extract-then-expand SCKDF scheme is* $(min\{t_X, t_Y\}, q_Y, \epsilon_X + \epsilon_Y)$*-CCS secure with the respect to the private string* p.

Proof: To satisfy the conditions of Theorem 1, we need to show,

i The extractor is a (t_X, ϵ_X)-computational extractor.
ii The expander is a (t_Y, q_Y, ϵ_Y)-secure variable-length-output pseudorandom function family.

To prove (i) we assume that extractor is not a (t_X, ϵ_X)-computational extractor. This would imply that an adversary A has a polynomial time method to distinguish whether PRK is derived from p or a random string of the same length. For the underlying pseudorandom keystream generator this would then imply that the adversary has a polynomial time method to distinguish between PRK and a truly random string. This contradicts the assumption that pseudorandom keystream generator satisfies Definition 9. Hence (i) is true. Similarly, we can show (ii) is true. Hence by Theorem 1 the *SCKDF* built from pseudorandom keystream generator is $(\min\{t_X, t_Y\}, q_Y, \epsilon_X + \epsilon_Y)$-CCS secure with the respect to the private string p. □

6 Performance Measurement

In order to compare the performance of stream cipher, hash function and block cipher based *KDF*s, we conducted experiments involving measuring the execution time taken to generate n bits of cryptographic key from p, s and c. The stream ciphers include the e-Stream finalists Trivium [6], Sosemanuk [4] and Rabbit [5]. It should be noted that to date these has been no significant security flaws discovered with any of these three stream ciphers. Hence any of these three stream ciphers seem to offer suitable pseudorandom keystream generator generators on which to build our *SCKDF* model in Section 4. The hash functions are SHA families and block cipher used is AES128. The code of the stream ciphers, hash functions and block ciphers are retreived from [18], [10] and [21] respectively. The lengths of the four parameters (p, s, c and n) are taken from the applications below:

- **Application 1**: Host identity protocol version 2(HIPv2) is based on the DH shared secret key exchange protocol, which provides secure communications and maintains shared IP-layer state between two separate parties [13]. The cryptographic keys are generated using *KDF* and the inputs are as below:
  - Exp 1 : p = 128 bytes, s = 8 bytes, c= 32 bytes, n = 64 bytes
  - Exp 2 : p = 128 bytes, s = 8 bytes, c= 32 bytes, n = 192 bytes
  - Exp 3 : p = 256 bytes, s = 8 bytes, c= 32 bytes, n = 64 bytes
  - Exp 4 : p = 256 bytes, s = 8 bytes, c= 32 bytes, n = 192 bytes
- **Application 2**: PKINIT is applied in Kerberos protocol [24]. The inputs to the *KDF* are as below:
  - Exp 5 : p = 128 bytes, s = null, c= 64 bytes, n = 64 bytes
  - Exp 6 : p = 128 bytes, s = null, c= 64 bytes, n = 192 bytes
  - Exp 7 : p = 256 bytes, s = null, c= 64 bytes, n = 64 bytes
  - Exp 8 : p = 256 bytes, s = null, c= 64 bytes, n = 192 bytes

– **Application 3**: The tunneled extensible authentication method (TEAM) is a method that securing communication between peer and server by using transport layer security (TLS) to establish a mutually authenticated tunnel[12]. The inputs to the *KDF* are as below:
  - Exp 9 : p = 40 bytes, s = 32 bytes, c= null, n = 128 bytes

6.1 Software Performance

For all nine experiments the time is recorded for each of 100 trials. The average time (mean) and standard deviation for each experiment are presented in Table 6.1. The execution time was captured using CLOCK_MONOTONIC (which can be found in the programming language C library). All the simulations were performed at a machine with the following specifications: Intel (R) core (TM) 2 duo CPU E8400 @ 3.00GHz 2.99 GHz, 4GB RAM and in 64 bit OS.

Table 6.1 shows the software performance of *KDF*s based on three different cryptographic primitives. The three cryptographic primitives are stream cipher, hash function and block cipher. The execution time for Exp 9 was relatively faster compare with Exp 1-8 for all *KDF* proposals. This is due to the inputs length in Exp 9 being shorter than the input lengths for Exp 1-8. Overall, the execution time for all types of *KDF* increases, when the lengths of the inputs (p, s, c or n) increase.

Another observation from this table is the performance results show that all three stream cipher based *KDF*'s were significantly more efficient in software than either the hash function or block cipher based *KDF*'s. While, the most efficient *KDF* is the Trivium based *KDF* and the slowest *KDF* is block cipher based *KDF*.

KDFs/Exp		Exp 1	Exp 2	Exp 3	Exp 4	Exp 5	Exp 6	Exp 7	Exp 8	Exp 9
Trivium	$\overline{x}$	12185.15	12886.61	19798.71	20623.36	14967.07	15829.43	21396.72	22628.53	4900.05
	S	1761.62	2575.09	1293.19	2178.31	1276.21	2605.27	219.91	4169.30	311.00
Sosemanuk	$\overline{x}$	18494.99	19237.54	30231.75	30153.16	20828.66	21714.94	33749.51	32449.32	8089.25
	S	2182.31	2040.69	5459.62	2906.47	2199.21	2547.35	8761.57	3162.79	165.06
Rabbit	$\overline{x}$	26307.11	26296.78	33691.33	36346.68	29845.23	31898.28	43679.73	43559.88	7825.69
	S	7024.60	1482.73	297.45	4245.17	277.67	1904.01	6523.79	3069.62	231.41
SHA1	$\overline{x}$	39583.75	79485.76	41267.27	83951.21	45068.91	96129.51	47816.59	99010.69	56364.39
	S	6129.62	1341.97	1422.08	2333.11	3489.43	1558.77	1429.24	712.13	5777.41
SHA224	$\overline{x}$	39327.14	77271.78	44264.8	82551.63	43258.77	80919.54	35060.03	74216.05	48076.72
	S	2453.69	1789.89	6180.79	16942.54	6144.87	1749.19	1519.05	668.50	1592.78
SHA256	$\overline{x}$	29756.25	68713.71	33140.46	72019.25	32685.35	72101.38	36264.51	75328.13	40487.08
	S	1581.87	2270.14	1384.51	1903.90	5763.42	4108.59	5491.55	3262.97	1540.99
SHA384	$\overline{x}$	82538.02	137821.08	84262.56	143402.09	83547.43	142936.8	89696.46	149876.24	96146.69
	S	3711.42	25900.93	1900.20	58864.25	1111.95	8541.55	1959.77	13847.19	556.55
SHA512	$\overline{x}$	54947.72	116787.15	60843.54	119729.85	59245.68	119497.2	60870.97	122340.7	74657.5
	S	1926.55	29908.35	1965.87	59220.04	5220.38	6334.94	2106.51	6051.33	1491.43
AES128	$\overline{x}$	236657.29	500199.78	330521.48	580804.49	322538.33	753565.93	410619.75	830594.7	148952.35
	S	12451.69	25568.83	41150.53	25316.09	10485.70	22020.28	21595.07	4468.46	52170.44

*Performance time is in nanosecond. $\overline{x}$ and S are sample mean and standard deviation respectively

Table 2. Software Performance of KDF.

6.2 Hardware Performance

This section presents hardware implementation and performance metrices for stream ciphers, hash functions and block ciphers. Note that, the hardware performance comparison is not the hardware performance of the actual *KDF* proposals. Rather it represents the hardware performance of the underlying cryptographic primitives obtained from existing literature. The result shows that Trivium requires less resource and has highest throughput, while SHA384 and SHA512 requires the highest resource in hardware. These results indicate that a hardware based *KDF* designed from Trivium using the *SCKDF* model would offer significant advantages over other designs in hardware.

	Trivium x64	Sosemanuk	Rabbit	SHA1	SHA224	SHA256	SHA384	SHA512	AES	Better is:
Gates	4921	18819	28000	9859	15329	15329	27297	27297	5398	Lower
Throughputs (Mb/s)	22300	6062	473.6	2006	2370	2370	2909	2909	311.09	Higher
Technology	0.13μm	0.13μm	0.18μm	0.13μm	0.13μm	0.13μm	0.13μm	0.13μm	0.11μm	
Reference	[11]	[11]	[5]	[19]	[19]	[19]	[19]	[19]	[20]	

Table 3. Hardware Performance of Hash Functions, Block Ciphers and Stream Ciphers.

7 Conclusion

A *KDF* is an essential component in generating cryptographic keys for safeguarding data storage and transmission over insecure channel. To be capable of working better on mobile devices such as smartphones, pocket PC and mobile phones, we proposed a lightweight *KDF* based on stream ciphers. Stream ciphers are often faster and require less resources which are suitable operated at low processing power and memory constrained devices, for example mobile devices. In this research, we have demonstrated that our newly proposed *KDF* based stream ciphers are secure if the underlying stream cipher are secure and more efficient compared to existing *KDF* proposals. From our analysis to date a *SCKDF* design based on Trivium cipher would be secure and efficient both in software and hardware.

References

1. C. Adams, G. Kramer, S. Mister, and R. Zuccherato. On the Security of Key Derivation Functions. *Information Security*, 3225:134–145, 2004.
2. X. ANSI. 9.42. *American National Standard for Financial Services-Public Key Cryptography for the Financial Services Industry: Agreement of Symmetric Keys Using Discrete Logarithm Cryptography*, 2001.
3. E.B. Barker, D. Johnson, and M.E. Smid. SP 800-56A. Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography (Revised). 2007.
4. C. Berbain, O. Billet, A. Canteaut, N. Courtois, H. Gilbert, L. Goubin, A. Gouget, L. Granboulan, C. Lauradoux, M. Minier, et al. Sosemanuk, a fast software-oriented stream cipher. *New Stream Cipher Designs*, pages 98–118, 2008.

5. M. Boesgaard, M. Vesterager, and E. Zenner. The Rabbit stream cipher. *New Stream Cipher Designs*, pages 69–83, 2008.
6. C.D. Canniere and B. Preneel. Trivium specifications. *citeseer. ist. psu. edu/734144. html*, 2005.
7. L. Chen. Recommendations for Key Derivation Using Pseudorandom Functions. *NIST Special Publication*, 800:108, 2008.
8. L. Chen. Recommendation for Key Derivation Using Pseudorandom Functions. *NIST Special Publication*, 800:108, 2009.
9. L. Chen. SP 800-56C. Recommendation for Key Derivation through Extraction-then-Expansion. 2011.
10. D. Eastlake and T. Hansen. US secure hash algorithms (SHA and SHA-based HMAC and HKDF). 2011.
11. T. Good and M. Benaissa. Hardware results for selected stream cipher candidates. *State of the Art of Stream Ciphers*, pages 191–204, 2007.
12. D. Harkins. Network Working Group G. Zorn Internet-Draft Network Zen Intended status: Standards Track Q. Wu Expires: September 9, 2011 Huawei. 2011.
13. T. Heer, P. Jokela, T. Henderson, and R. Moskowitz. Host Identity Protocol Version 2 (HIPv2). 2012.
14. B. Kaliski. PKCS# 5: Password-based cryptography specification version 2.0. Technical report, RFC 2898, September 2000, 2000.
15. H. Krawczyk. Cryptographic Extraction and Key Derivation: The HKDF Scheme. *Advances in Cryptology–CRYPTO 2010*, pages 631–648, 2010.
16. D. McGrew and B. Weis. Key Derivation Functions and Their Uses, 2010. Online available at url http://www.ietf.org/id/draft-irtf-cfrg-kdf-uses-00.txt.
17. A.J. Menezes, P.C. Van Oorschot, and S.A. Vanstone. *Handbook of applied cryptography*. CRC, 1997.
18. M. Robshaw. The eSTREAM Project. *New Stream Cipher Designs*, pages 1–6, 2008.
19. A. Satoh and T. Inoue. ASIC-hardware-focused Comparison for Hash Functions MD5, RIPEMD-160, and SHS. *INTEGRATION, the VLSI journal*, 40(1):3–10, 2007.
20. A. Satoh, S. Morioka, K. Takano, and S. Munetoh. A Compact Rijndael Hardware Architecture with S–box Optimization. *Advances in Cryptology–ASIACRYPT 2001*, pages 239–254, 2001.
21. J.H. Song, R. Poovendran, J. Lee, and T. Iwata. INTERNET DRAFT Ibaraki University Expires: May 6, 2006 November 7 2005 The AES-CMAC Algorithm draft-songlee-aes-cmac-02. txt. 2005.
22. W. Stallings. *Cryptography and Network Security: Principles and Practices, Fourth Edition*. Pearson Education India, 2006.
23. F.F. Yao and Y.L. Yin. Design and Analysis of Password-based Key Derivation Functions. *Topics in Cryptology–CT-RSA 2005*, pages 245–261, 2005.
24. L. Zhu, M. Wasserman, and L.H. Astrand. PKINIT Algorithm Agility. 2012.