Technische Universität Ilmenau
Fakultät für Informatik und Automatisierung
Fachgebiet Telematik/Rechnernetze

Research Project

Design and Evaluation of Key Chains for Symmetric Key Management

Submitted By:	Prateek Banerjee
Submitted On:	August 28, 2024
Born On:	
Degree Programme:	M.Sc. in RCSE
Responsible University Professor:	Prof. Dr.-Ing. Günter Schäfer
Scientific Supervisor:	M.Sc. David Schatz

Abstract

Cryptographic key chains can be referred as cryptographic tools which allow a user to provide input parameters to an existing state of a key chain and generate output keys which can be used for cryptographic purposes. Such cryptographic key chains can be used for dynamic key management and are observed to be used in online messaging platforms like Signal, WhatsApp, etc. which critically ensures the end-to-end encryption of the individual chats. This study primarily aims to investigate the different cryptographic primitives that can be used to design such cryptographic key chains. Additionally, a comparative performance overview is also provided to emphasize the usability of the different cryptographic primitives for generating the key chains. To date, we have only observed the realization of cryptographic key chains based on HMAC-Based Key Derivation Function (HKDF) but we demonstrate that key chains can also be generated using a Pseudorandom Generator (PRG) and an XOF-Based Deterministic Random Bit Generator (XDRBG). And, we can indeed say that, when aiming for 16 bytes (128 bits) of classical security provided by the output keys, the XDRBG-based key chain demonstrates the best performance depending on the choice of the underlying Extendable Output Function (XOF). For 32 bytes (256 bits) and 64 bytes (512 bits) of classical security, the HKDF-based key chain demonstrates the best performance, depending on the choice of the hash functions.

Contents

1 Introduction 2
1.1 About (Cryptographic) Key Chain . . . 2
1.2 Our Contribution . . . 4

2 Background 5
2.1 Sponge-Based Extendable Output Function (XOF) . . . 5
2.2 HMAC-Based Key Derivation Function (HKDF) . . . 6
2.2.1 Hash Message Authentication Code (HMAC) . . . 6
2.2.2 Extract-then-Expand Key Derivation Function (KDF) . . . 8
2.2.3 HMAC-Based Instantiation of KDF . . . 8
2.3 The Advanced Encryption Standard Scheme . . . 10
2.3.1 Overview of the Block Cipher . . . 10
2.3.2 AES Block Cipher in Counter (CTR) Mode . . . 10
2.4 Circulant . . . 11
2.4.1 Randomness Extraction . . . 11
2.4.2 Circulant Construction . . . 14

3 Requirements and Related Work 15
3.1 Requirements of a Cryptographic Key Chain . . . 15
3.2 Related Work . . . 16
3.2.1 Double Ratchet . . . 16
3.2.2 XOF-Based Deterministic Random Bit Generator (XDRBG) . . . 17
3.2.3 Pseudorandom Generator (PRG) . . . 18

4 Key Chain Constructions 21
4.1 Additional Context Regarding Design of the Key Chains . . . 21
4.2 Key Chain Based on HKDF . . . 22
4.3 Key Chain Based on XDRBG . . . 24
4.4 Key Chain Based on PRG . . . 27

5 Evaluation 31
5.1 Performance Evaluation . . . 31
5.1.1 Interpretation of Performance Evaluation Without Persistent Storage . . . 33
5.1.2 Interpretation of Performance Evaluation With Persistent Storage 36
5.2 Security Discussions . . . 36
5.2.1 For the HKDF-Based Key Chain . . . 37
5.2.2 For the XDRBG-Based Key Chain . . . 37
5.2.3 For the PRG-Based Key Chain . . . 37
5.3 Evaluation of Entropy Loss . . . 37
5.3.1 The Analysis Methodology . . . 38
5.3.2 The Two Idealizations . . . 39
5.3.3 Applying the Idealizations to the Cryptographic Primitives . . . 40
5.3.4 Idealization Results . . . 42

6 Conclusion 46

A Other Variety of KDFs 47

Bibliography 49

List of Abbreviations 52

Chapter 1

Introduction

With the advancements in the field of quantum computing, existing cryptographic algorithms are at peril in the face of an adversary possessing a quantum computer. Although such quantum computers are currently bound to their theoretical existence only, but progress is being made to realize them. The strength of asymmetric (also known as public-key) cryptography primarily relies on being one-way functions, i.e., it is computationally infeasible for an adversary (with access to any classical computer) to reverse it. Considering the RSA algorithm, which involves the multiplication of two very large (secret) prime numbers to generate a (publicly known) prime modulus. It is easy to derive said prime modulus, but till date it is next to impossible to factorize (the prime modulus) using a classical computer and identify the two (correct) individual prime numbers whose product resulted in that prime modulus. Quantum computers leveraging quantum mechanics along with Shor's algorithm [1], [2] can compromise the security provided by an asymmetric cryptographic primitive, as it allows a fairly easy reversal (in comparison to a classical computer) of such one-way functions. In contrast to the impact of quantum computing on asymmetric cryptography, symmetric cryptography is expected to thrive comparatively better [3]. This is because, algorithms such as Grover's algorithm [4] can only weaken the existing symmetric key cryptographic primitives but that too at a significant cost to an adversary. Till date, there is no evidence that Grover's algorithm [4] has been able to successfully circumvent any symmetric key cryptographic primitive [3]. Furthermore, according to [3], to fortify and future-proof the existing symmetric key cryptographic primitives, moving to larger key sizes would be enough of a precaution against any quantum threat to symmetric cryptography.

While symmetric cryptography is inherently more resistant to quantum adversaries as opposed to asymmetric cryptography, the security of symmetric cryptography can be further fortified by employing dynamic key management techniques in addition to moving to larger key sizes. Such dynamic key management can be achieved by using cryptographic key chains. A cryptographic key chain can be considered as a sequence of (cryptographic) keys that can be used for encryption and decryption purposes, which are dynamically generated by combining arbitrary input parameters to an existing state of the cryptographic key chain. These key chains can prove to be of significant usage in applications or any cryptosystems where it is integral for the cryptographic keys to be updated regularly or where new keys must be generated periodically to uphold the security of the system.

1.1 About (Cryptographic) Key Chain

A cryptographic key chain commences upon the establishment of an initial (and ideally secure) state of the key chain which is generated by instantiating the key chain by providing an initial input to a suitable cryptographic primitive. Going further, it allows a user of the key chain to provide arbitrary input parameters to the (same) cryptographic primitive in addition to the existing state of the key chain which results in the generation

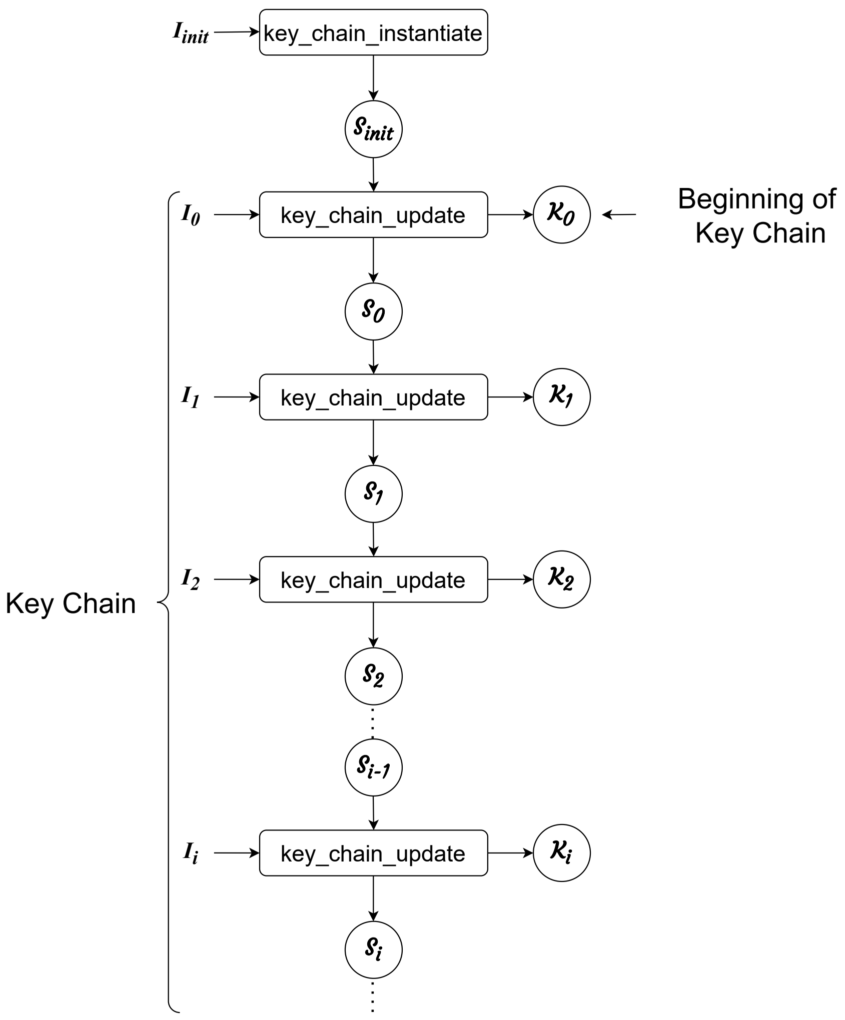

Figure 1.1: Pictorial Description of the Generic Interface of a Cryptographic Key Chain

of a new cryptographic key along with a new state of the key chain. The state of the key chain is expected to offer security for the key chain itself, which is considered to be equivalent to that of the actual cryptographic key intended to be used for any cryptographic purposes. More often than not, these key chains are (practically) realized by means of Key Derivation Functions (KDFs) as the suitable cryptographic primitive. One concrete example of usage of key chains is the Double Ratchet algorithm [5] in the predominantly used messaging platforms like Signal, WhatsApp etc. which is based on the Hash-based Message Authentication Code (HMAC)-based Key Derivation Function (KDF), i.e., HKDF. Although there are other types of KDFs (mentioned in Appendix A) as well in addition to HKDF [6], we have not yet seen those KDFs being used for the realization of any key chains.

Generic Interface of a Cryptographic Key Chain

In Figure 1.1, considering the cryptographic primitive is indeed a KDF (like the HKDF), then the key chain is instantiated (using $key\_chain\_instantiate(I_{init})$) with an initial input referred as I_{init} of size n bytes to generate an initial state $\mathcal{S}_{init}$ of the key chain. Starting from $\mathcal{S}_{init}$, generation of the actual cryptographic keys (using $key\_chain\_update(I_i, \mathcal{S}_{i-1})$) begins which takes $I_i, i \in \{0, 1, 2...\}$ as the arbitrary input parameter of size m bytes and the existing state of the key chain $\mathcal{S}_{i-1}$ as inputs (to the KDF) and generates the respective (cryptographic) key $\mathcal{K}_i, i \in \{0, 1, 2...\}$ and the (new) state of the key chain $\mathcal{S}_i, i \in \{0, 1, 2...\}$. Both the cryptographic key $\mathcal{K}_i$ and the state of the key chain $\mathcal{S}_i$ are a part of the total output generated from the cryptographic primitive (which in this case is the KDF). Amongst these, the key $\mathcal{K}_i$ is expected to be used for encryption and decryption purposes and the state $\mathcal{S}_i$ of the key chain can be stored in a persistent storage and this particular $\mathcal{S}_i$ acts as a starting point for the next cryptographic key in the key chain. Suppose two users wish to use such key chains for symmetric encryption, then they must ensure that the input parameter I_{init} to $key\_chain\_instantiate(I_{init})$ and the other input parameters, i.e., $I_i, i \in \{0, 1, 2...\}$ to $key\_chain\_update(I_i, \mathcal{S}_{i-1})$ must be the same and are provided to the cryptographic primitive in the exact same order to ensure the synchronicity of their key chains by generating the same cryptographic key $\mathcal{K}_i$. One important thing to be aware of is that, in the real world, the input parameters I_{init} and $I_i, i \in \{0, 1, 2...\}$ might be drawn from an imperfect source, so they can be potentially compromised or may not have a sufficient degree of randomness. In such scenarios, usage of a KDF (like the HKDF) is a safe choice for the generation of cryptographic keys because a KDF particularly facilitates in deriving cryptographically strong keys $\mathcal{K}_i$ irrespective of the source from where the particular input parameter is drawn. By cryptographically strong, we wish to emphasize that the output key $\mathcal{K}_i$ must be completely indistinguishable from random to any external observer, especially an adversary. If the input parameters have a sufficient amount of vetted randomness or are uncompromised, then they could be directly used as a seed for a PRG or a Deterministic Random Bit Generator (DRBG) [6] of whose random outputs can be used a cryptographic key.

1.2 Our Contribution

We mentioned earlier that KDFs are one potential cryptographic primitive that could be used to construct a cryptographic key chain. And, based on our knowledge, till date key chains have been realized only with the usage of HKDF like the Double Ratchet [5]. But, we believe that there are other cryptographic primitives, namely a PRG and an XDRBG which will also prove to be useful in the successful realization of a cryptographic key chain. To that end, we have provided a design of a cryptographic key chain based on a PRG and an XDRBG in addition to a key chain based on HKDF along with their corresponding implementations to demonstrate that there are indeed other ways of realizing cryptographic key chains as opposed to only HKDF-based key chains. Furthermore, we also provide a comparative performance overview of the three different key chain constructions based on HKDF [6], XDRBG [7] and PRG [8].

Further Report Organization: The remainder of this work is further organized as follows: In chapter 2, we provide a detailed overview of the different building blocks. These building blocks are used for constructing the cryptographic primitives such as XDRBG [7] and PRG [8] which are mentioned as related works in chapter 3 as they play a vital role in the key chain construction. In chapter 4, we provide the design and construction of the cryptographic key chain using the HKDF [6], XDRBG [7] and PRG [8]. Then, chapter 5 provides our implementation details and the performance evaluation of the key chain built using the different cryptographic primitives, and we provide our concluding comments in chapter 6.

Chapter 2

Background

In this chapter we describe in detail about the different building blocks namely, XOF [9], Advanced Encryption Standard (AES) Block Cipher in Counter Mode [10] and the cryptographic primitive HKDF [6]. We additionally provide an overview about randomness extraction along with construction of a randomness extractor [11] as well.

2.1 Sponge-Based Extendable Output Function (XOF)

Bertoni et al. (2007) [12] proposed a new way of constructing hash functions which are different from the design of classical hash functions from the SHA-2 family [13]. These (new) hash functions are referred as *sponge functions* which can take a string s of length $l_1 \geq 1$ as input and produce an output string of arbitrary-length l_2. A sponge function requires the following components [12], [14]:

1. Let AL be a set of *alphabet group* where AL represents both input and output characters of a sponge function. These input and output characters can be considered as bit strings $\{0,1\}^*$ of variable lengths. This set AL forms the first part of the internal state $\mathcal{S}$ of the sponge function.
2. Let $\mathcal{C}$ be a finite set of elements which represents the second part of the internal state $\mathcal{S}$ of the sponge function. This internal state of the sponge function is represented as $\mathcal{S} = (\mathcal{S}_{AL}, \mathcal{S}_{\mathcal{C}}) \in AL \times \mathcal{C}$, whose initial value is (0,0).
3. Let f be a bijective function defined as $f : \{0,1\}^b \to \{0,1\}^b$ which maps a bit string of length b to bit strings of same length.

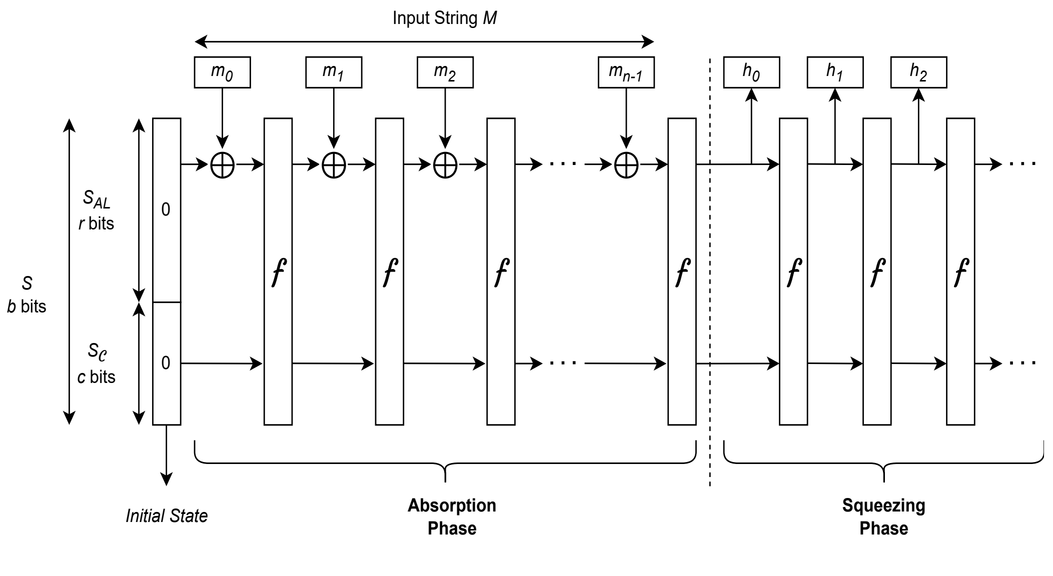

Figure 2.1: Phases of a Sponge-Based XOF [15].

The memory of the internal state $\mathcal{S}$ of a sponge-based XOF is of $b = r + c$ bits where $r = \log_2 A,\ A = |AL|$ and $c = \log_2 C,\ C = |\mathcal{C}|$. Generally, r is referred as the (bit) *rate* and c is referred as the *capacity* of the sponge function. This means that, $\mathcal{S}_{AL}$ will be of r bits and $\mathcal{S}_{\mathcal{C}}$ will be of c bits. The first part of the whole state $\mathcal{S}$ of the sponge function, i.e., $\mathcal{S}_{AL}$ determines the efficiency of the sponge function in terms of the number of bits that it can absorb or squeeze out in a single call of the aforementioned (item 3) function f and the second part $\mathcal{S}_{\mathcal{C}}$ determines the security provided by the sponge function [12], [15]. This resulting broad class of cryptographic functions was referred as Extendable Output Functions (XOFs) in [9]. More generally speaking, they are also referred as sponge-based XOFs [7]. The most notable implementation of such sponge-based XOFs are observed in the SHA-3 family [14] of cryptographic hash functions and the ASCON-XOF in [16]. As depicted in Figure 2.1, there are two phases when using such a sponge-based XOF which are as follows [12], [15], [17]:

1. Absorption: The sponge-based XOF absorbs all blocks of the input string as follows:
    a) The input string M is divided into fixed-size blocks.
    b) Each block is XORed ($\oplus$) with the first part of the internal state, i.e., $\mathcal{S}_{AL}$.
    c) The internal state $\mathcal{S} = (\mathcal{S}_{AL}, \mathcal{S}_{\mathcal{C}})$ is then transformed using the function f as defined earlier (in item 3). This process is continued until all the blocks of the input string have been processed.
2. Squeezing: The sponge-based XOF now produces (or squeezes out) the output which is only extracted from the first part of the internal state $\mathcal{S}$ of the sponge function, i.e., $\mathcal{S}_{AL}$. The second part of the internal state, i.e., $\mathcal{S}_{\mathcal{C}}$ is completely untouched when generating the output from the sponge function, as it is vital for ensuring the security provided by the sponge function, i.e. no output is generated from the second part of the internal state. After each block of generated output, the function f as defined earlier (in item 3) is applied again to squeeze out further output blocks. This process continues until the desired length (l_2) of output is generated.

2.2 HMAC-Based Key Derivation Function (HKDF)

In this section, we initially provide a brief overview of HMAC [18] and an abstraction of a KDF based on the *extract-then-expand* approach. Finally, we provide details about an instantiation of the KDF based on HMAC, i.e. HKDF from [6].

2.2.1 Hash Message Authentication Code (HMAC)

HMAC is a cryptographic (message) authentication function that uses hash functions from the SHA-2 [13] or SHA-3 [14] family along with a secret input key K on a particular input string (also referred as the message) M [19]. Each hash function has a particular block size denoted as b bytes and a digest size d bytes where $b > d$ [13], [14] and our choice of hash functions in this report (mentioned in Table 4.2) also follow the same convention where $b > d$. The block size b determines the size of the input block that the hash function can process at a time (which we can also refer as the rate r of the hash function), and the digest size d determines the (fixed) length of the generated output, i.e., the hash digest. This means that a hash function defined as $h \leftarrow H(M)$

takes M as input and generates a hash digest h (of the message M) of length d as output. The HMAC function is figuratively depicted in Figure 2.2 and is also defined below [18], [19]:

$$\text{HMAC}(K, M) = H\Big((K' \oplus opad) \parallel H(K' \oplus ipad) \parallel M\Big)$$

$$K' = \begin{cases} H(K) \| \Big((b-d) \text{ zeroes}\Big) & \text{if length of } K > b, \\ K \| \Big((b - len(K)) \text{ zeroes}\Big) & \text{if length of } K < b, \\ K & \text{otherwise} \end{cases}$$

where

H is the hash function, K is the secret input key

M is the input string which is to be hashed

K' is the derived secret key equal to the block size b

$\|$ denotes concatenation

$\oplus$ denotes XOR operation

len denotes the length

$$\left.\begin{array}{l} opad \text{ denotes an outer padding with byte value } 0x5c \\ ipad \text{ denotes an inner padding with byte value } 0x36 \end{array}\right\} \text{ of length } = b$$

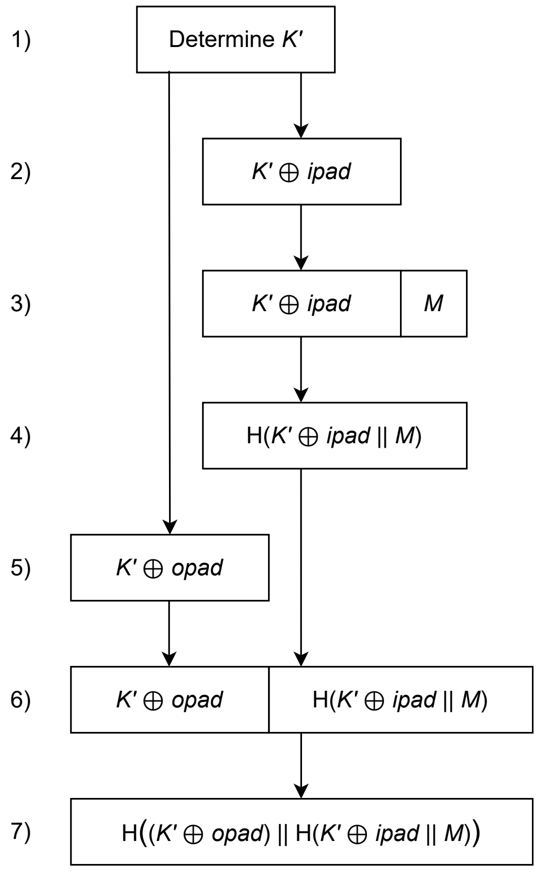

Figure 2.2: Illustration of HMAC Construction [19].

2.2.2 Extract-then-Expand Key Derivation Function (KDF)

According to [6], a KDF takes an input parameter (ideally with a good amount of randomness) and generates cryptographically strong keys. By cryptographically strong, we mean that the generated key should be completely indistinguishable from a random value/random string of the same length to an external observer, especially an adversary. Some properties of KDFs are as follows [20]:

1. Determinism: For the same input, the KDF should always produce the same output. This ensures consistency in key generation.
2. Pseudorandomness: For any input, the output of the KDF must always be computationally indistinguishable from a truly random value.
3. Collision-Resistance: It must be computationally infeasible (accidentally or intentionally by anyone, especially an adversary) to identify two inputs for which the KDF generates the same output.
4. Resistant to Known Attacks: It must be computationally infeasible for an adversary to predict the output key of a KDF provided the adversary $\mathcal{A}$ is oblivious to the input or parts of the input.

A KDF based on the *extract-then-expand* approach is a cryptographic algorithm which accepts four inputs, namely, an input parameter SKM sampled from a source keying material, a parameter l which defines the total desired length of the output from the KDF, an extractor salt XTS, and a contextual info parameter $info$. Both the XTS and $info$ are optional parameters, so they can be either set to a constant or a null string as well [6]. The salt XTS can also be a non-secret value and if the contextual info parameter $info$ is not a null string, then it is recommended (in [6]) that it must be set to a value which is uniquely (and cryptographically) bound to each of the output generated from the KDF. The security of the KDF is critically dependent on the input parameter SKM sampled from the source keying material [6]. Such an *extract-then-expand* KDF comprises of two modules [6]:

- An extractor XTR defined as $PRK \leftarrow \text{XTR}(XTS, SKM)$ which takes the (optional) salt XTS and the input parameter SKM as inputs and generates a pseudorandom key PRK as output. It is expected that the PRK will be computationally indistinguishable from random to any external observer.
- A Pseudorandom Function (PRF) defined as $KM \leftarrow \text{PRF}(PRK, info, l)$ takes the pseudorandom key PRK, contextual info parameter $info$ (if provided), and the total desired output length l from the KDF as inputs and generates KM as the output. The PRF basically expands the PRK to generate an output of length l. We wish to explicitly mention that the total output KM can be segregated into multiple outputs each of certain length $< l$ such that all of them sum-up to the length l.

2.2.3 HMAC-Based Instantiation of KDF

According to [6], an *extract-then-expand* KDF (like the one mentioned in subsection 2.2.2) can be instantiated with HMAC (from subsection 2.2.1). Then, HMAC can serve as both the extractor XTR and the pseudorandom function PRF and the resultant scheme can be denoted as HKDF. The authors of [6] use the notation $\text{HMAC}(p, q \parallel r)$ which means that, for a chosen hash function H, the HMAC function $\text{HMAC}(K, M)$ as defined in subsection 2.2.1 takes the (secret) key p as the parameter K and a concatenation (denoted using $\parallel$) of q and r as the parameter M. Then, the HKDF scheme which is mentioned in Algorithm 1 is defined as [6]:

$$\begin{aligned} KM &\leftarrow \text{HKDF}(XTS, SKM, info, l) \\ KM &= K(1) \parallel K(2) \parallel \dots K(i) \parallel \dots K(t) \end{aligned}$$

where each $K(i)$ is generated as follows [6]:

$$\begin{aligned} PRK &= \text{XTR}(XTS, SKM) = \text{HMAC}(XTS, SKM) \\ KM &= \text{PRF}(PRK, info, l) \\ K(1) &= \text{HMAC}(PRK, K(0) \parallel info \parallel 1), \\ K(2) &= \text{HMAC}(PRK, K(1) \parallel info \parallel 2), \\ &\vdots \\ K(i) &= \text{HMAC}(PRK, K(i-1) \parallel info \parallel i), \\ K(i+1) &= \text{HMAC}(PRK, K(i) \parallel info \parallel i+1), 1 \le i \le t \end{aligned}$$

where $K(0)$ is a null string, so we can also say that $K(1) = \text{HMAC}(PRK, info \parallel 1)$ and $t = \lceil l/d \rceil$ with d being the digest size of the hash function H. The parameter t denotes the number of blocks needed to generate the total output of the desired length l provided by the user. According to [6], the salt XTS can be at most d bytes (if at all provided) and each $K(i)$ (from above) does not correspond to individual cryptographic keys. Instead, these $K(i)$'s are individual output blocks generated from the HKDF which are concatenated together to form the final output KM of length l. We wish to clarify that, if the desired length of the output, i.e., l is completely divisible by the digest size d of the hash function H, then each block $K(i)$ where $1 \le i \le t$ will consist of exactly d bytes, otherwise the last block of output, i.e., $K(t)$ is truncated to its first $l \bmod d$ bytes.

Algorithm 1 HKDF Operations [6]

```
1:  function XTR($XTS, SKM$)                          ▷ HKDF extract function
2:      HMAC($XTS, SKM$)                              ▷ Keyed with the salt $XTS$
3:      return $PRK$
4:  end function
5:  function PRF($PRK, info, l$)                      ▷ HKDF expand function
6:      $t \leftarrow \lceil l/d \rceil$                                  ▷ $d$ is the hash digest size
7:      $KM \leftarrow$ b""
8:      $K(0) \leftarrow$ b""
9:      for $i$ in range(1,$t$) do                         ▷ Both 1 and $t$ included
10:         $K(i) \leftarrow \text{HMAC}(PRK, K(i-1) \parallel info \parallel i)$
11:         $KM \leftarrow KM \parallel K(i)$
12:     end for
13:     return $KM$
14: end function
```

2.3 The Advanced Encryption Standard Scheme

In this section, we initially provide a very brief overview of the AES encryption scheme [21], [22] and then describe about the AES block cipher in counter mode [10].

2.3.1 Overview of the Block Cipher

The authors of [21] proposed the Rijndael family of block ciphers, amongst which three were chosen to be formalized by the National Institute of Standards and Technology (NIST) in 2001 [22] as the AES. The AES encryption scheme can be represented as $C \leftarrow E_K(P)$ where C denotes the ciphertext generated upon encrypting the plaintext P using the AES encryption scheme E with the input key K. The three different variants of AES are mentioned in Table 2.1 along with their fixed parameter combinations.

Table 2.1: Key-Block-Round Combinations in AES [21], [22]

AES Variant	Key Length in Bits	Block Size in Bits	Number of Rounds
AES-128	128	128	10
AES-192	192		12
AES-256	256		14

2.3.2 AES Block Cipher in Counter (CTR) Mode

The authors of [10] proposed a new mode of operation of AES, namely the counter mode, formalized by NIST [23] in 2001. The AES encryption scheme E when used in counter mode can be represented as $E_K(ctrblk)$ which denotes the encryption of a counter block *ctrblk* with an input key K. When using AES block cipher in counter mode, the same parameter combinations from Table 2.1 are directly applicable here and the security of AES block cipher in counter mode is critically dependent on the uniqueness of each counter block *ctrblk* which must be of length exactly 128 bits (16 bytes). This counter block *ctrblk* is generated by the concatenating (denoted using $\|$) a *nonce* of 96 bits (12 bytes) and a counter value *ctr* of 32 bits (4 bytes) where the initial value of *ctr* is set to 0. To uphold the security of the scheme, it must be ensured that the same *ctrblk* is not reused for any block $P_{i \in \{0,\ldots,n-1\}}$ of a plaintext message P. Encryption of a plaintext message P using AES block cipher in counter mode (shown in Figure 2.3) involves the following steps [10], [23]:

1. Generate a *nonce* which will be fixed for every counter block *ctrblk* and set the initial value of *ctr* to 0.
2. Generate a (unique) counter block as $ctrblk_i \leftarrow nonce \| ctr$ where $i \in \{0, \ldots, n-1\}$.
3. For encryption of each block $P_{i \in \{0,\ldots,n-1\}}$ of a plaintext message P:
   a) Compute $KS_i \leftarrow E_K(ctrblk_i)$ where $i \in \{0, \ldots, n-1\}$. Each KS_i can be referred as a key-stream block which is of 128 bits (16 bytes).
   b) Then, compute $C_i \leftarrow KS_i \oplus P_i$ where $i \in \{0, \ldots, n-1\}$ which is a ciphertext block of 128 bits (16 bytes) and concatenate C_i with the previous ciphertext block C_{i-1} which will be possible from $i \in \{1, \ldots, n-1\}$.

c) Increase *ctr* by 1 to ensure the uniqueness of the next counter block $ctrblk_{i+1}$.

4. Repeat the previous two steps (item 2 with the incremented *ctr* from item 3c and item 3 as whole) until all the plaintext blocks $P_{i\in\{0,\dots,n-1\}}$ have been encrypted.

Handling the Encryption of the Last Block of the Plaintext Message

For encrypting a plaintext message P, it is divided into blocks $P_{i\in\{0,\dots,n-1\}}$ of 128 bits (16 bytes) each according to the (fixed) block size mentioned in Table 2.1. If the size of plaintext P is exactly divisible by 128, then each plaintext message block P_i is also of exactly 128 bits (16 bytes), whereas, if the size of plaintext P is not exactly divisible by 128 then the last plaintext message block P_{n-1} is considered as a partial block of (say) u bits where $u < 128$. Then, P_{n-1} can be padded to satisfy the property of a plaintext message block being equal to 128 bits [23]. Although plaintext formatting is outside the scope of [23], but padding is not recommended for the last plaintext message block (by [23]) in case of using AES block cipher in counter mode. Instead, for AES block cipher in counter mode, if the last plaintext message block P_{n-1} is a partial block of $u < 128$ bits, then the last ciphertext block C_{n-1} is computed as [23]:

$$\begin{aligned} KS_{n-1} &\leftarrow E_K(ctrblk_{n-1}) \\ C_{n-1} &\leftarrow \mathrm{MSB}_u(KS_{n-1}) \oplus P_{n-1} \end{aligned}$$

where $\mathrm{MSB}_u(KS_{n-1})$ denotes the most significant u bits of the key-stream block KS_{n-1} which is XORed ($\oplus$) with the last (partial) plaintext message block P_{n-1} also of u bits. In this scenario, with the AES block cipher in counter mode, the remaining $128 - u$ bits of the key-stream block KS_{n-1} is discarded.

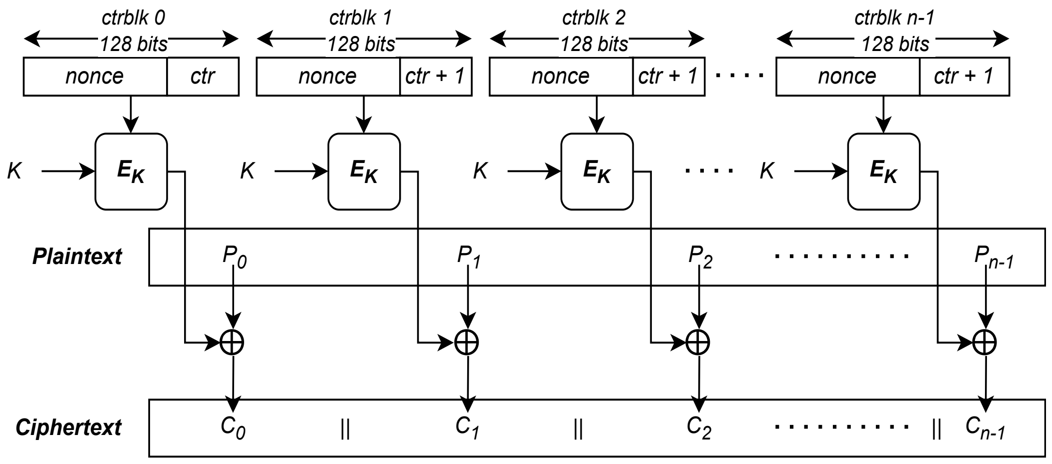

Figure 2.3: AES Block Cipher in Counter Mode [10], [24].

2.4 Circulant

In this section, we initially provide an overview of what a randomness extractor is, and then we mention the details of the randomness extractor named Circulant [11].

2.4.1 Randomness Extraction

A randomness extractor acts as an integral part in many cryptographic applications. It can be considered as a function which transforms a *weak* (input) string (which is also referred as the source) and generates an output string of *near-perfect* randomness [11]. By a weak string, it is meant that the input only has some minimum entropy and the near-perfect random output means that the generated output must be completely indistinguishable from random, which is quantified by the extractor error ε. Some randomness extractors also require a *seed* (in addition to the input string) and this seed is referred as a weak seed if it only has some minimum entropy, i.e., the seed itself is not near-perfect random [11].

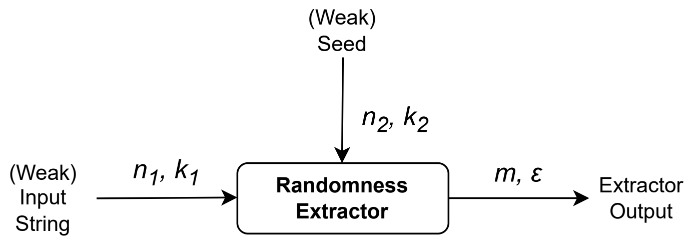

Figure 2.4: A Randomness Extractor Processing Two Inputs to Generate an Output [11].

As illustrated in Figure 2.4, the randomness extractor is processing two inputs, i.e., an input string and a weak seed to generate a near-perfect random output. The parameters n_1 and n_2 denote the bit length of the input string and the seed respectively and k_1 and k_2 (respectively) denotes their minimum entropy. The generated output is of m bits which is ε-perfectly random [11].

What Does it Mean to be ε-Perfectly Random?

Before we delve into the details of ε-perfect randomness, we wish to provide the details of some notation from [11] which will be required going further:

- We denote random variables using upper case, e.g., X and the values that this variable takes is denoted using lower case, like x. The probability that X takes x is denoted as $\Pr(X = x) = p_X(x)$. Here, $p_X(x)$ is the probability mass function, which basically states the probability of X being equal to x [25]. For clarification, let's say that X can take only three values a, b and c with a probability of 0.1, 0.6, and 0.3 respectively, then $p_X(a) = 0.1$, $p_X(b) = 0.6$ and $p_X(c) = 0.3$.
- For two such random variables, say X and E holding values x and e respectively, we denote [26]:
  - Conditional probability as $\Pr(X = x|E = e) = p_{X|E=e}(x)$. This defines the probability of the event where the variable X takes the value x provided the variable E takes the value e.
  - Joint probability as $\Pr(X = x, E = e) = p_{X,E}(x, e)$ which defines the probability of the two events, i.e., X taking the value x and E taking the value e occurs simultaneously.
- The notation $p_{guess}(X) = \max_x p_X(x)$ denotes the (maximum) guessing probability. Based on our understanding this means that, let's say that X can only take three values a, b and c with a probability of 0.1, 0.6, and 0.3 respectively which would mean that $p_X(a) = 0.1$, $p_X(b) = 0.6$ and $p_X(c) = 0.3$. Then, $\max_x p_X(x) = p_X(b) = 0.6$ which means that $p_{guess}(X) = 0.6$. So, we can say that $p_{guess}(X)$ denotes that how predictable is the random variable X, which

will be determined by the highest probability of the value x that the variable X can take amongst all the possible values x.

- The notation $p_{guess}(X|E) = \max_{x,e} p_{X|E=e}(x)$ denotes the maximum guessing probability of an event where the random variable X takes the value x provided the variable E takes the value e. This extends the concept from the previous point by considering the predictability of X in context of availability of additional information E. Akin to the concept that $p_{guess}(X)$ denoting the predictability of X which is determined by the highest probability of the value x that the variable X can take, $p_{guess}(X|E)$ similarly depicts the predictability of X when conditioned on E which determines the highest probability of the value x that the variable X can take provided the variables E holds the value e.

- The symbol Δ denotes the statistical distance between two variables, say X and Y. It is a measure of the difference of the distributions of X and Y, i.e., it represents that how different or alike X and Y are for someone to be able to successfully distinguish them. The notation p_{dist} denotes the distance measurement.

A Simple Analogy of the Security Condition

The output generated from a randomness extractor (say) $OP_{extractor}$ is only considered to be useful in cryptographic applications, if it fulfils a security condition. This condition is best realized as a hypothetical game involving a *computationally unbounded* adversary $\mathcal{A}$ who is provided two values, namely the randomness extractor output $OP_{extractor}$ and an output OP_{random} from a completely random source, i.e., OP_{random} is indeed completely random. The adversary $\mathcal{A}$ wins this game iff $\mathcal{A}$ is able to distinguish between $OP_{extractor}$ and OP_{random}. For the adversary $\mathcal{A}$, a trivial way of distinguishing between $OP_{extractor}$ and OP_{random} would be to just take a random guess, which means that the probability of successfully distinguishing between $OP_{extractor}$ and OP_{random} by guessing is $\frac{1}{2}$. For a secure protocol where $OP_{extractor}$ is to be used, it is expected that any computationally unbounded adversary $\mathcal{A}$ cannot distinguish between $OP_{extractor}$ and OP_{random} with a success probability $> \frac{1}{2} + p_{dist}$ where p_{dist} is negligible. This means that any strategy taken by the adversary $\mathcal{A}$ is not (much) better than the trivial one, which ensures that indeed the output $OP_{extractor}$ is completely indistinguishable from random [11]. Here the authors consider a *computationally unbounded* adversary as opposed to a *computationally bounded* adversary because the randomness extractor is considered to provide *information-theoretic security* which means that it is impossible for any adversary $\mathcal{A}$ to circumvent the security of the randomness extractor by being able to successfully distinguish between $OP_{extractor}$ and OP_{random}.

Statistical Distance

The aforementioned analogy of security is formalised based on the (statistical) distance measurement p_{dist} between two random variables $X, Y \in \{0,1\}^*$ where we can say that $X = OP_{extractor}$ and $Y = OP_{random}$. The statistical distance Δ between X and Y when both of them are conditioned on another random variable E is mathematically represented as $\Delta(X, Y|E)$ [11]. Considering that X and Y fulfil the condition $\Delta(X, Y|E) \leq \varepsilon$, then we can say that X and Y are ε-close. In simpler terms, we can also say that, for a possible value e taken by the variable E, the maximum difference between X and Y is at most ε (where ε is expected to be extremely small) [11]. This means that X and Y can be distinguished only with a limited advantage of $p_{dist} \leq \frac{\varepsilon}{2}$. With respect to the aforementioned analogy of the security condition, if ε is extremely small (like $\varepsilon = 2^{-32}$), then p_{dist} will be negligible, thus ensuring that the adversary $\mathcal{A}$ only has a success probability $\leq \frac{1}{2}$. In the context of randomness extractors ε is referred as the (tolerable) extractor error, which is a fixed security parameter in [11], like setting $\varepsilon = 2^{-32}$. So, when we say that the output generated from a randomness extractor is ε-perfectly random, we mean that the probability of an adversary $\mathcal{A}$ successfully distinguishing between $OP_{extractor}$ and OP_{random} is $\leq \frac{1}{2} + p_{dist} \approx \frac{1}{2}$ where p_{dist} will indeed be negligible if ε is extremely small (like $\varepsilon = 2^{-32}$). This means that there is no difference between the adversary $\mathcal{A}$ taking a random guess at distinguishing $OP_{extractor}$ & OP_{random} and actually performing some computation to distinguish between $OP_{extractor}$ & OP_{random}.

2.4.2 Circulant Construction

With respect to Figure 2.4, let's consider:

- The (weak) input string as $x = x_0, x_1, \cdots, x_i, \cdots, x_{n-2} \in \{0,1\}^{n-1}$ of length $n-1$.
- The (weak) seed as $y = y_0, y_1, \cdots, y_i, \cdots, y_{n-1} \in \{0,1\}^n$ of length n.

Here $n \in \mathbb{P}$, i.e. n is a prime value. Considering the strings x and y have a minimum entropy of k_1 and k_2 bits respectively and the (tolerable) extractor error is ε, then the randomness extraction function Circulant is defined as [11]:

$$\text{Circulant}(x, y) : \{0,1\}^{n-1} \times \{0,1\}^n \to \{0,1\}^m$$

where $m \leq k_1 + 2\log_2(\varepsilon)$ is the output length. The workflow of the Circulant extractor is as follows [11]:

1. Generate another string $x' \leftarrow x \parallel 0$ where $\parallel$ denotes concatenation. Upon concatenating 0 to the string $x = x_0, x_1, \cdots, x_i, \cdots, x_{n-2} \in \{0,1\}^{n-1}$ of length $n-1$, the resultant string x' is of length equal to n. In simpler terms, we can also say that the string $x' = x'_0, x'_1, \cdots, x'_i, \cdots, x'_{n-2}, 0$ where the first $n-1$ bits are from the string x and the n^{th} bit, i.e., $x'_{n-1} = 0$. We wish to explicitly clarify that $x'_i \in x' = x_i \in x, 0 \leq i \leq n-2$. It is just denoted using x'_i because, firstly, it is a bit in the bit string x' and secondly, the Circulant extractor internally works with the string x' and not x.

2. The above Circulant function, i.e., $\text{Circulant}(x, y) : \{0,1\}^{n-1} \times \{0,1\}^n \to \{0,1\}^m$ can then be implemented as [11]:

$$\text{Circulant}(x, y) = (\text{circ}(x')y)_{0:m-1}$$

where

$$\text{circ}(x') = \begin{bmatrix} x'_0 & x'_1 & \cdots & x'_i & \cdots & x'_{n-2} & x'_{n-1} \\ x'_{n-1} & x'_0 & x'_1 & \cdots & x'_i & \cdots & x'_{n-2} \\ & \ddots & \ddots & \ddots & \ddots & \ddots & \\ x'_1 & x'_2 & \cdots & x'_i & \cdots & x'_{n-1} & x'_0 \end{bmatrix}^{n \times n}$$

3. The matrix $\text{circ}(x')$ is a special type of matrix where each row is a cyclic permutation of the row above it. A matrix-vector multiplication is performed between $\text{circ}(x')$ and the string y (which is considered as a vector) where each element of the resulting vector is taken mod 2. The subscript $0 : m-1$ denotes the extraction of only the first m elements (out of n elements) of the resulting vector as the final output, i.e., $OP_{extractor}$ is a bit-string $\{0,1\}^m$ of length m.

Chapter 3

Requirements and Related Work

In this chapter, at first we describe the different requirements of a cryptographic key chain, and then we mention the related works.

3.1 Requirements of a Cryptographic Key Chain

For the realization of a (cryptographically secure) key chain, it must fulfil the following:

1. Security Requirements:
    a) With respect to Figure 1.1, even if an adversary $\mathcal{A}$ at a later point in time somehow gets to know a state $\mathcal{S}_i$ of the key chain, it must not be able to reconstruct an output $\mathcal{K}_j$ where $0 \leq j \leq i$ solely based on the knowledge of this value, i.e. the past output $\mathcal{K}_j$ where $0 \leq j \leq i$ must still seem completely random to $\mathcal{A}$. This property can also be referred as *forward secrecy.*
    b) With respect to Figure 1.1, even if an adversary $\mathcal{A}$ possesses knowledge about a particular state $\mathcal{S}_{i-1}$, it must not be able to derive $\mathcal{K}_i$ solely based on the knowledge about $\mathcal{S}_{i-1}$ if additional entropy in the form of I_i is provided. This property can also be referred as *backward secrecy.*
    c) With respect to Figure 1.1, if an adversary $\mathcal{A}$ is oblivious of the current state $\mathcal{S}_{i-1}$ of the key chain, the output $\mathcal{K}_i$ must seem completely indistinguishable from random. It must also hold even if the adversary $\mathcal{A}$ has complete control over the input parameter I_i which is used during *key_chain_update*$(I_i, \mathcal{S}_{i-1})$. This property can also be referred as the *resilience* of the key chain.
    d) There must be as little entropy loss as possible from the cryptographic keys when using a particular cryptographic primitive to generate the keys. This property can also be referred as *entropy preservation.*
2. Functional Requirements:
    a) With respect to Figure 1.1 each state $\mathcal{S}_i$ of the key chain which is generated as part of the total output from a cryptographic primitive from which the corresponding key $\mathcal{K}_i$ is also generated must be stored in a persistent storage. This is to ensure that, even after rebooting the system, one does not have to go through the hassle of starting the key chain by generating an initial state $\mathcal{S}_{init}$ at first. So, this state $\mathcal{S}_i$ serves as the last known (secure) state of the key chain which allows the user to just provide the input parameter I_{i+1} along with this state $\mathcal{S}_i$ to generate the next cryptographic key $\mathcal{K}_{i+1}$ along with the new state $\mathcal{S}_{i+1}$ of the key chain. For the key chain construction, we also refer to this as the *persistent storage requirement.*
    b) We consider an *interface compliance* requirement. It means that the key chain must follow the generic interface mentioned in Figure 1.1 involving the following operations:

- The key chain is at first instantiated using $key\_chain\_instantiate(I_{init})$ to generate the initial state of the key chain, i.e. $\mathcal{S}_{init}$ with a certain amount of minimum entropy. Here, the input parameter I_{init} is of size n bytes.
- In order to generate an actual cryptographic key $\mathcal{K}_i$ in the key chain, the $key\_chain\_update(I_i, \mathcal{S}_{i-1})$ method is used only after the establishment of $\mathcal{S}_{init}$ which takes an input parameter $I_i, i \in \{0, 1, 2...\}$ of size m bytes along with the current state of the key chain $\mathcal{S}_{i-1}$ as inputs.

The concrete value for the size of the input parameters, i.e., I_{init} and I_i where $i \in \{0, 1, 2...\}$ is explicitly provided in chapter 4 depending on the cryptographic primitive using which the key chain is realized.

3. Non-Functional Requirement: The key chain must be easy to use and computationally efficient (especially if it is to be used in a time-critical system).

We wish to reiterate that, in real world, if two users wish to use a cryptographic key chain following the interface from Figure 1.1, then they must use the same input parameter I_{init} for instantiation and the same input parameters $I_i, i \in \{0, 1, 2...\}$ in the exact same order to ensure the synchronicity of their key chains by generating the same cryptographic key $\mathcal{K}_i$ for symmetric encryption.

3.2 Related Work

Here, we briefly mention the Double Ratchet [5] at first and then describe in detail about XDRBG [7] and PRG [8] owing to their usage in the key chain's construction.

3.2.1 Double Ratchet

Perrin and Marlinspike proposed the Double Ratchet algorithm [5] which is used to provide end-to-end encryption on instant messaging platforms. KDF-based chains are an integral part of the Double Ratchet algorithm, and in a double ratchet session, each of the two interacting parties have a root chain, sending chain, and receiving chain. The output keys of a user's root chain become the new KDF key for the sending and receiving chains and it is called the Diffie-Hellman ratchet. The output keys (different from chain keys) of a user's sending and receiving chain are used for encrypting the messages, and it is called the symmetric-key ratchet. The double ratchet is an elegant combination of the Diffie-Hellman ratchet and the symmetric-key ratchet. We believe the Double Ratchet algorithm to be the closest related work with respect to what we also intend to achieve. One drawback of the Double Ratchet is the Diffie-Hellman key exchange being baked into the design of the Double Ratchet algorithm itself which is needed for the (symmetric) key management, i.e, to ensure that the same cryptographic keys are generated in one user's sending chain and the other user's receiving chain. However, the Diffie-Hellman key exchange is not quantum resistant, thereby leaving the Double Ratchet vulnerable to quantum adversaries. In contrast, our key chain construction does not use Diffie-Hellman key exchange but still allows dynamic key management provided the users use the same input parameters in the same order for the different cryptographic primitives (as reiterated above). It is out of the scope of this work as to how exactly two users will obtain the same input parameters.

3.2.2 XOF-Based Deterministic Random Bit Generator (XDRBG)

A DRBG is a cryptographic tool designed to generate a sequence of random bits from an initial value sampled from a source of randomness. This initial value can also be referred as a seed [7], [27] and the most essential property of this source of randomness is its (minimum) entropy. According to [28], entropy in information theory refers to the degree of unpredictability or randomness of a particular data. And, a DRBG based on an underlying XOF is referred as an XDRBG [7]. For an XDRBG, there are two thresholds for the minimum entropy from where the seed must be drawn, which are as follows [7]:

1. H_{init} : When the XDRBG is instantiated, the seed (SD_{init}) must be drawn from a randomness source with at least H_{init} bits of minimum entropy. This means that SD_{init} will have H_{init} bits of minimum entropy.
2. H_{rsd} : When the XDRBG is reseeded, the seed (SD_{rsd}) must be drawn from a randomness source with at least H_{rsd} bits of minimum entropy. This means that SD_{rsd} will have H_{rsd} bits of minimum entropy.

A DRBG based on an XOF, i.e., an XDRBG has the following operations (also mentioned in Algorithm 2) for a user to interact with it [7]:

1. $\mathcal{S} \leftarrow$ INSTANTIATE(SD_{init}, α) takes the initialization seed SD_{init} and an optional parameter α as input and generates $\mathcal{S}$ as an output which is the initial state of the XDRBG. Here, SD_{init} must be retrieved from a source having a minimum entropy of H_{init} which also means that SD_{init} must be at least H_{init} bits long.
2. $\mathcal{S} \leftarrow$ RESEED($\mathcal{S}', SD_{rsd}, \alpha$) takes the current XDRBG state $\mathcal{S}'$, the seed for reseeding SD_{rsd} and an optional parameter α as input and generates $\mathcal{S}$ as an output which is the reseeded state of the XDRBG. Here, SD_{rsd} must be retrieved from a source having a minimum entropy of H_{rsd} which also means that SD_{rsd} must be at least H_{rsd} bits long.
3. $(\mathcal{S}, \mathcal{K}) \leftarrow$ GENERATE($\mathcal{S}', l, \alpha$) takes the current XDRBG state $\mathcal{S}'$, the desired length l of the output (in bits) and an optional parameter α as input and generates $\mathcal{S}$ and $\mathcal{K}$ as outputs, where $\mathcal{S}$ is the new state of the XDRBG and $\mathcal{K}$ refers to the random output bits. The random output $\mathcal{K}$ must be exactly l bits long and indistinguishable from random.

There are some properties that an XDRBG should fulfil, which are as follows [7]:

- Forward Security: Even if an external observer gets to know about the current XDRBG state (say) $\mathcal{S}_i$, then also a past output $\mathcal{K}_{i-1}$ from the XDRBG (GENERATE call) looks completely random to the observer (also mentioned as *forward secrecy* in section 3.1).
- Backward Security: Even if an external observer is aware of a particular XDRBG state $\mathcal{S}_{i-1}$, then also a random output $\mathcal{K}_i$ generated from an XDRBG GENERATE call in future looks completely random to the observer if there is a RESEED (or INSTANTIATE) call before the GENERATE call to provide $\geq H_{rsd}$ (or $\geq H_{init}$) bits of minimum entropy (also mentioned as *backward secrecy* in section 3.1).

The ENCODE Function

The authors of [7] mention the usage of an ENCODE function for the XDRBG which is evident from Algorithm 2 and is defined as:

$$\text{ENCODE} : \{0,1\}^* \times \{0,1\}^* \times 0,1,2 \rightarrow \{0,1\}^*$$

such that $\forall\ S \leftarrow \{0,1\}^*, \alpha \leftarrow \{0,1\}^*, n \leftarrow \{0,1,2\}$ where $(S, \alpha, n) \neq (S', \alpha', n')$, it must hold that, $\text{ENCODE}(S, \alpha, n) \neq \text{ENCODE}(S', \alpha', n')$. The authors of [7] recommend the aforementioned encoding for the XDRBG while also explicitly mentioning that different ENCODE functions can also be used, provided the encoding is unambiguous, i.e., for each unique input combination of S, α and n, the encoded output is also unique and that it does not introduce any trivial collisions. The parameter α used in the ENCODE function is the same α which is defined as the optional parameter in the aforementioned XDRBG operations. The length of the α parameter in [7] must be at most 84 bytes, i.e., $|\alpha|/8 \in \{0, 1, \cdots, 84\}$. Then the following encode function unambiguously encodes the inputs S, α and n by concatenating them (denoted as $\|$) while adding only a single byte of stretch:

$$\text{ENCODE}(S, \alpha, n) = (S \ \| \ \alpha \ \| \ (n * 85 + |\alpha|/8)_8)$$

The 8-bit (or 1 byte) value computed as $(n{*}85{+}|\alpha|/8)_8$ is referred as the stretch, and this is the value which ensures that the encoding is unambiguous. The subscript 8 notation in $(n * 85 + |\alpha|/8)_8$ denotes the equivalent bytes representation of the corresponding bit value generated from the computation of $(n * 85 + |\alpha|/8)$.

Algorithm 2 XDRBG Operations [7]

```
1:  function INSTANTIATE($SD_{init}, \alpha$)
2:      $\mathcal{S} \leftarrow \mathsf{xof}(\text{ENCODE}(SD_{init}, \alpha, n = 0), |\mathcal{S}|)$
3:      return $\mathcal{S}$
4:  end function
5:  function RESEED($\mathcal{S}', SD_{rsd}, \alpha$)
6:      $\mathcal{S} \leftarrow \mathsf{xof}(\text{ENCODE}((\mathcal{S}' \| SD_{rsd}), \alpha, n = 1), |\mathcal{S}|)$
7:      return $\mathcal{S}$
8:  end function
9:  function GENERATE($\mathcal{S}', l, \alpha$)
10:     $T \leftarrow \mathsf{xof}(\text{ENCODE}(\mathcal{S}', \alpha, n = 2), l + |\mathcal{S}|)$
11:     $\mathcal{S} \leftarrow$ first $|\mathcal{S}|$ bits of $T$
12:     $\mathcal{K} \leftarrow$ last $l$ bits of $T$
13:     return $(\mathcal{S}, \mathcal{K})$
14: end function
```

3.2.3 Pseudorandom Generator (PRG)

A PRG can be considered as a deterministic polynomial time algorithm denoted as G which takes a short random input (seed) of length (say) l_1 and generates a pseudorandom output of length l_2 where $l_1 < l_2$ [29]. Such a function G is defined as follows [29]:

$$\text{G} : \{0,1\}^{l_1} \rightarrow \{0,1\}^{l_2}; l_2 > l_1$$

According to the architecture of a PRG provided by Barak and Halevi (2005) [8] which (also) uses a function G as mentioned above, they refer to this G as a (non-robust) cryptographic PRG. To avoid any confusion between (the architecture of) a PRG itself and the function G, from here onwards we will refer G using the term *generator function*. In [8], such a generator function is defined as G : $\{0,1\}^{\lambda} \to \{0,1\}^{2\lambda}$ where λ is the security parameter. Although we believe that such a generator function G should be able to generate an output bit-string of arbitrary length and not of just $\{0,1\}^{2\lambda}$, we are convinced that the authors in [8] just chose to use a fixed output length of 2λ bits for their construction. A PRG based on the architecture defined in [8] has the following two operations (also mentioned in Algorithm 3) for a user to interact with it [8]:

1. $\mathcal{S} \leftarrow \text{REFRESH}(\mathcal{S}', x)$ takes a current PRG state $\mathcal{S}'$ of length λ bits and a random additional input x and generates $\mathcal{S}$ of length λ bits as output which is the refreshed PRG state.

2. $(\mathcal{K}, \mathcal{S}) \leftarrow \text{NEXT}(\mathcal{S}')$ only takes the current PRG state $\mathcal{S}'$ of length λ bits as input and generates an output of length 2λ bits, amongst which the first λ bits are the random output bits referred as $\mathcal{K}$ and the last λ bits refer to the new PRG state $\mathcal{S}$. The random output $\mathcal{K}$ must be exactly λ bits long and indistinguishable from random.

A PRG based on the architecture from [8] must fulfil the following properties [8]:

- Forward Security: Even if an external observer gets to know about the current PRG state $\mathcal{S}_i$, then also a past output $\mathcal{K}_{i-1}$ from the PRG (NEXT call) looks completely random to the observer (also mentioned as *forward secrecy* in section 3.1).

- Backward Security/Break-In Recovery: Even if an external observer is aware of a particular PRG state $\mathcal{S}_{i-1}$, then also a random output $\mathcal{K}_i$ generated from a PRG NEXT call in future looks completely random to the observer if there is a REFRESH call before the NEXT call (also mentioned as *backward secrecy* in section 3.1).

- Resilience: The output from the generator is completely indistinguishable from random to an external observer with no knowledge of the (internal) state of the PRG. This property also holds even if the said observer has complete control over the data x which is used to refresh the (internal) PRG state $\mathcal{S}$ (also mentioned as *resilience* in section 3.1).

We follow the recommendation provided in [8] and choose to proceed with usage of AES block cipher in Counter Mode from subsection 2.3.2 as the generator function G. Based on the parameters mentioned in Table 2.1 and Algorithm 3, the usage of AES Counter mode as the generator function G means the following:

- In the REFRESH call, the resultant value from $\mathcal{S}' \oplus X$ acts as the input key K to the AES block cipher in counter mode defined in subsection 2.3.2. So, it is imperative that the resultant value from $\mathcal{S}' \oplus X$ must be of length $\in \{16, 24, 32\}$ bytes or ($\in \{128, 192, 256\}$ bits respectively) and nothing else.

- In the NEXT call, the state $\mathcal{S}'$ acts as the input key K to the AES block cipher in counter mode defined in subsection 2.3.2. It is imperative for this state $\mathcal{S}'$ to be of length $\in \{16, 24, 32\}$ bytes or ($\in \{128, 192, 256\}$ bits) respectively.

We wish to clarify that, nothing is explicitly mentioned in [8] regarding consideration of a plaintext P with respect to the usage of AES block cipher in counter mode as the generator function G. But, to fit our needs, we do consider a plaintext $P \leftarrow \{0\}^{2\lambda}$ which a bit string of all zeroes of length 2λ where each plaintext block $P_{i\in\{0,\ldots,(2\lambda/128)-1\}}$ is XORed ($\oplus$) with the respective key-stream block $KS_{i\in\{0,\ldots,(2\lambda/128)-1\}}$ which can be considered equivalent to using the key-stream block itself as the pseudorandom output from the generator function G. Furthermore, in [8], no explicit INSTANTIATE method is mentioned for the PRG. And, the initial state of the PRG is considered to be $\mathcal{S} \leftarrow \{0\}^{\lambda}$ which is a bit-string all zeroes of length equal to the security parameter λ.

Algorithm 3 PRG Operations [8]

```
1: function REFRESH($\mathcal{S}'$, $x$)
2:     $X \leftarrow$ EXTRACT($x$)
3:     $\mathcal{S} \leftarrow$ G($\mathcal{S}' \oplus X$)
4:     return $\mathcal{S}$          ▷ Only the first $\lambda$ bits out of the generated $2\lambda$ bits
5: end function
6: function NEXT($\mathcal{S}'$)
7:     $T \leftarrow$ G($\mathcal{S}'$)
8:     $\mathcal{K} \leftarrow$ first $\lambda$ bits of $T$
9:     $\mathcal{S} \leftarrow$ last $\lambda$ bits of $T$
10:    return ($\mathcal{K}$, $\mathcal{S}$)
11: end function
```

The EXTRACT Function

Upon closely observing REFRESH($\mathcal{S}', x$) in Algorithm 3, it uses an EXTRACT function whose output is XORed ($\oplus$) with the current PRG state $\mathcal{S}'$ and the resulting value is the input for the generator function G. This randomness extractor function EXTRACT should take an (ideally high entropy) input referred as the parameter x and generate a random output shorter than x, but it does not need to be cryptographic at all [8]. To be precise, the parameter x that is drawn from a randomness source must have at least λ bits of minimum entropy, so the resulting refreshed state $\mathcal{S}$ from REFRESH($\mathcal{S}', x$) is expected to be unpredictable for an adversary. For a security parameter λ, the extract function is defined as [8]:

$$\text{EXTRACT} : x \in \{0,1\}^{\geq\lambda} \rightarrow X \in \{0,1\}^{\lambda}$$

The output from the EXTRACT(x) function, i.e., X must be such that it is completely indistinguishable from random to an external observer. As there is no particular mandate in [8] for the usage of a specific randomness extractor, in our case of using the PRG as a building block for the cryptographic key chain, we indeed use the randomness extractor Circulant as mentioned in subsection 2.4.2 for the EXTRACT function using which we generate the parameter X which can be used for refreshing the PRG state.

Usability of the XDRBG and the PRG: We acknowledge that inherently the XDRBG and the PRG do not directly resemble with any (cryptographic) key chain, but we show (in chapter 4) that the different operations of the XDRBG and the PRG allows us to reuse them to fit our needs and design an XDRBG-based key chain and a PRG-based key chain.

Chapter 4

Key Chain Constructions

In this chapter, we initially mention a unit measurement notation that we use for our key chain construction, along with some additional context regarding the design of the key chains. Then, using the cryptographic primitives HKDF [6], XDRBG [7], and PRG [8] mentioned in section 2.2, subsection 3.2.2 and subsection 3.2.3 respectively we provide three instantiations of the cryptographic key chain.

Unit Measurement Notation: For our key chain construction using the aforementioned cryptographic primitives, we have considered the unit of representation to be *bytes* instead of *bits* and the same convention follows in our implementation as well. To remove any confusion, let's consider Algorithm 2 where we mentioned that, from a GENERATE call, we obtain a random output $\mathcal{K}$ of l bits and a state $\mathcal{S}$ of $|\mathcal{S}|$ bits. Here, we use the same symbols, i.e., l and $|\mathcal{S}|$ but we mention the random output $\mathcal{K}$ to be l bytes and the state $\mathcal{S}$ to be $|\mathcal{S}|$ bytes instead of bits. In our key chain construction, this means that we use the equivalent bytes representation of l bits or $|\mathcal{S}|$ bits, i.e., if a particular value is of 192 bits and if that value is represented in bytes then it will be of $192/8 = 24$ bytes, so we write 24 bytes as the length of that particular value instead of writing 192 bits. This is why, all the parameter measurements in the different tables in the following sections and the majority of the symbols are represented in bytes from here onwards unless explicitly represented in bits.

4.1 Additional Context Regarding Design of the Key Chains

For the key chain constructions, we have to use certain input parameters referred as I_{init} and I_i (from Figure 1.1) which in our case is obtained from the randomness extractor Circulant (mentioned in subsection 2.4.2). Although it is not mandatory for a user of the key chain to only use Circulant as the means of obtaining these input parameters, but it must be ensured that these input parameters (wherever they are obtained from) do have a certain amount of minimum entropy which will vary depending on the underlying cryptographic primitive using which the key chain has been constructed. Additionally, it has been a matter of debate in the literature with respect to the frequency of RESEED and REFRESH calls regarding the usage of XDRBG and PRG respectively. This is because, after the usage of an XDRBG and a PRG for a certain amount of time, such a RESEED and REFRESH call will facilitate in replenishing the XDRBG and the PRG respectively with a certain amount of minimum entropy which is vital to uphold the security provided by the XDRBG and PRG. This is because of the output generated from such a RESEED and REFRESH call, i.e., the reseeded and the refreshed state $\mathcal{S}$ (denoted in Algorithm 2 and Algorithm 3). But, if such an XDRBG and PRG is to be used in a real-life application, it is imperative to decide on a reseed and refresh interval which strikes a balance between the security of the cryptographic primitive and its usage. As no details have been explicitly mentioned neither in [7] nor in [8] regarding the RESEED and REFRESH intervals for the XDRBG and the PRG respectively, it is left up to the discretion of the user of the cryptographic primitive. In order to perform these RESEED and REFRESH calls, we need input parameters referred as I_i. Going further, we are going to provide a key chain construction based on XDRBG and PRG, and in order to ensure that the key chain construction is indeed resilient (based on the resilience requirement from section 3.1) even when an adversary $\mathcal{A}$ has control over majority or some of the input parameters I_i (from Figure 1.1), then the more the RESEED and REFRESH calls for the XDRBG and the PRG respectively, the better in terms of the provided security.

For our purposes, the frequency of these RESEED and REFRESH calls is limited by the fact that, when two users are maintaining their own key chains and using them (for symmetric encryption), the key chains have to be synchronized with one another (like we mentioned in the end of section 3.1). This calls for the necessity of generating these input parameters I_i and exchanging this shared input material to ensure the generation of the same output key $\mathcal{K}_i$. But, the generation and the exchange of these input parameters will come with some computation cost and communication overhead respectively, which is unavoidable.

4.2 Key Chain Based on HKDF

With respect to the standard interface of the key chain in Figure 1.1, the HKDF operations in Algorithm 1 and our vision of the HKDF-based key chain in Figure 4.1:

1. For the key chain instantiation using $key\_chain\_instantiate(I_{init})$, the parameter I_{init} refers to the *SKM* from Algorithm 1. The parameter I_{init} is of n bytes, and it generates the initial state $\mathcal{S}_{init}$ of the key chain and then the actual key chain starts.
2. After establishment of $\mathcal{S}_{init}$, the $key\_chain\_update(I_i, \mathcal{S}_{i-1})$ method is used which involves the following steps:
   a) The extractor XTR (also referred as the HKDF Extract call) is used at first, which only takes the *SKM* (excluding the optional salt *XTS*) as input to generate the pseudorandom key *PRK* as the output. The *SKM* is generated by concatenating the arbitrary input parameter $I_i, i \in \{0, 1, 2...\}$ of m bytes and the current state of the key chain, i.e., $SKM \leftarrow I_i \parallel \mathcal{S}_{i-1}$.
   b) The pseudorandom function PRF (also referred as the HKDF Expand call) is used which takes the above generated pseudorandom key *PRK* and the length l (excluding the optional *info* parameter) as inputs to generate the output *KM* of length $l \leftarrow |\mathcal{S}| + |\mathcal{K}|$ which is segregated as follows:
      i. The parameter $\mathcal{K}_i$ of size $|\mathcal{K}|$ bytes represents the random output which can be used for cryptographic purposes.
      ii. The parameter $\mathcal{S}_i$ of size $|\mathcal{S}|$ bytes represents the state of the key chain which will be persistently stored and will act as the starting point for the next cryptographic key $\mathcal{K}_{i+1}$ in the key chain following the analogy mentioned as the *persistent storage requirement* in section 3.1.
3. Over time, the key chain using HKDF is formed by the chronological repetition of all the steps from the preceding point (item 2).

In our key chain construction, the length of I_{init} and each I_i is kept the same, i.e., $n = m$, and the concrete values of these parameters for the different hash functions used with the HKDF is mentioned in Table 4.1.

Table 4.1: Input Parameter Size in Bytes for HKDF-Based Key Chain

Hash Function	I_{init} (n bytes)	I_i (m bytes)
SHA-256	32	
SHA3-256		
SHA-512	64	
SHA3-512		

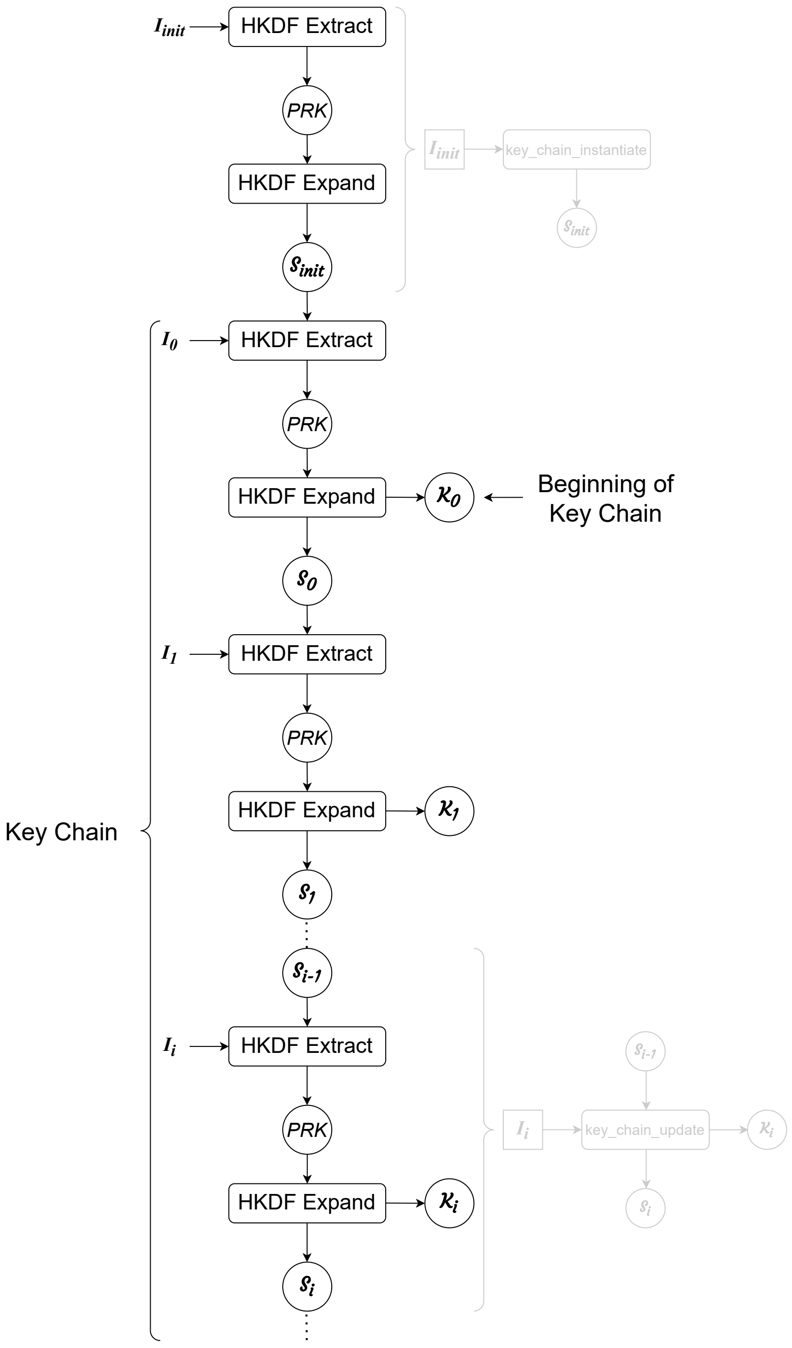

Figure 4.1: HKDF-Based Key Chain

The maxout Value in HKDF: When using the HKDF to form the key chain, as we consider part of the output material KM to be the state $\mathcal{S}$ of the key chain, we must choose a certain state size while bearing in mind the (fixed) *maxout* value of the HKDF as well. This *maxout* value denotes the maximum length $l \leftarrow |\mathcal{S}| + |\mathcal{K}|$ of output that can be generated from the HKDF in a single invocation of the PRF, i.e., in a single HKDF Expand call and this l is primarily dependent on the choice of the underlying hash function. We have chosen the hash functions mentioned in Table 4.2 to use for the HKDF [13], [14], [30]. The parameter which one must be cautious about is the size $|\mathcal{K}|$ of the output key $\mathcal{K}$ as the other parameters like block size b, digest size d etc. are inherently dependent on the hash function itself over which a user has no control and for our key chain construction we choose to fix the state size $|\mathcal{S}|$ depending on the hash function. Although we fix the key chain's state size to be $|\mathcal{S}|$ bytes, it is still modifiable, but changes in the state size $|\mathcal{S}|$ will also lead to changes in the limit on the size $|\mathcal{K}|$ of the output key $\mathcal{K}$ to ensure that the length $l \leftarrow |\mathcal{S}| + |\mathcal{K}|$ still conforms to the bounds of the *maxout* value of the different hash functions.

Table 4.2: Rules for Parameter Size in Bytes for HKDF-Based Key Chain [13], [14], [30]

Hash Function	Size of Each Block (b)	Size of Hash Digest (d)	maxout ($255 \times d$)	Output State Size $\|\mathcal{S}\|$	Random Output Size $\|\mathcal{K}\|$
SHA-256	64	32	8,160	32	$\leq$ 8128
SHA3-256	136				
SHA-512	128	64	16,320	64	$\leq$ 16,256
SHA3-512	72				

4.3 Key Chain Based on XDRBG

With respect to the standard interface of the key chain in Figure 1.1, the XDRBG operations in Algorithm 2 and our vision of the XDRBG-based key chain in Figure 4.2:

1. The parameter I_{init} of n bytes denotes the seed SD_{init} required for the XDRBG INSTANTIATE call and all the other input parameters, i.e., I_i where $i \in \{0, 1, 2...\}$ of m bytes denote the seed SD_{rsd} required for the RESEED call with respect to Algorithm 2. The length of SD_{init} and SD_{rsd} for different XOFs is mentioned in Table 4.3.

2. For the key chain instantiation, we use the *key_chain_instantiate*(I_{init}) method where I_{init} denotes the seed SD_{init} and it generates an initial state $\mathcal{S}_{init}$ by invoking an INSTANTIATE call to the XDRBG. This serves as the initial (and ideally it is expected to be a secure) state of the cryptographic key chain itself and afterwards the key chain starts. We wish to explicitly mention that we do not use the optional α parameter for the INSTANTIATE call.

3. After the establishment of $\mathcal{S}_{init}$, the *key_chain_update*($I_i, \mathcal{S}_{i-1}$) method is used which involves the following steps:
    a) At first it invokes the RESEED call to the XDRBG which takes an input parameter, i.e., I_i and the current state of the key chain (say) $\mathcal{S}_{i-1}$ as inputs which are SD_{rsd} and $\mathcal{S}'$ respectively (according to Algorithm 2) and produces $\mathcal{S}_{rsd}$ as the output which is the reseeded state of the XDRBG and

it serves as the current state of the key chain as well. We wish to explicitly mention that we do not use the optional α parameter for the RESEED call.

b) Then, it invokes a GENERATE call to the XDRBG which takes the current state of the key chain, i.e., $\mathcal{S}_{rsd}$ and the length l (excluding the optional α parameter) as inputs and produces the total output T comprising of:

   i. $\mathcal{K}_i$ of size l bytes which represents the random output $\mathcal{K}$ suitable to be used for cryptographic purposes.

   ii. $\mathcal{S}_i$ of size $|\mathcal{S}|$ bytes (in Figure 4.2) which is the same as the new state $\mathcal{S}$ of the XDRBG (according to Algorithm 2) and this serves to be the (new) state of the key chain as well. Following the analogy mentioned as the *persistent storage requirement* in section 3.1, this parameter $\mathcal{S}_i$ will be persistently stored because it was generated as part of the total output T from the XDRBG GENERATE call from which the corresponding key $\mathcal{K}_i$ was also generated. This state $\mathcal{S}_i$ will act as the starting point for the generation of the next cryptographic key $\mathcal{K}_{i+1}$ in the key chain because $\mathcal{S}_i$ will be used as an input to the next (RESEED) call to the XDRBG.

4. Over time, the key chain using XDRBG is formed by the chronological repetition of all the steps from the preceding point (item 3).

It is mentioned in [7] that according to [31] and [32] respectively, a sponge-based XOF with a capacity of c bits can provide an approximate classical security of $c/2$ bits and an approximate quantum security of $c/3$ bits. According to [27], DRBG's have a security level denoted as $\lambda \in \{128, 192, 256\}$ bits. So, for a DRBG based on (say) SHAKE-128 XOF, i.e. a SHAKE-128 XDRBG where SHAKE-128 has a capacity c of 32 bytes (256 bits), it can provide classical security of $32/2 = 16$ bytes ($256/2 = 128$ bits). As we mentioned in subsection 3.2.2 regarding the requirement of minimum entropy of H_{init} and H_{rsd} bits for the seeds SD_{init} and SD_{rsd} respectively, it can also be defined in terms of the security level λ as follows [7]:

1. The seed SD_{init} for instantiation must have a minimum entropy of $H_{init} \geq 3\lambda/2$ bits.
2. The seed SD_{rsd} for reseeding must have a minimum entropy of $H_{rsd} \geq \lambda$ bits.

Table 4.3: Input Parameter Size in Bytes for XDRBG-Based Key Chain

XOF	SD_{init} (I_{init} of n bytes)	SD_{rsd} (I_i of m bytes)
SHAKE-128	24	
SHAKE-256	48	
ASCON-XOF	24	

For our key chain construction, we consider the same amount of minimum entropy for the input parameter I_{init} (which is the seed SD_{init}) and the input parameters I_i where $i \in \{0, 1, 2...\}$ (which is the seed SD_{rsd}), i.e., $n = m$ which also means that $H_{init} = H_{rsd}$. This is chosen in a manner such that it indeed fulfils the two conditions mentioned above, and the specific values of these parameters for different XOFs are mentioned in Table 4.3.

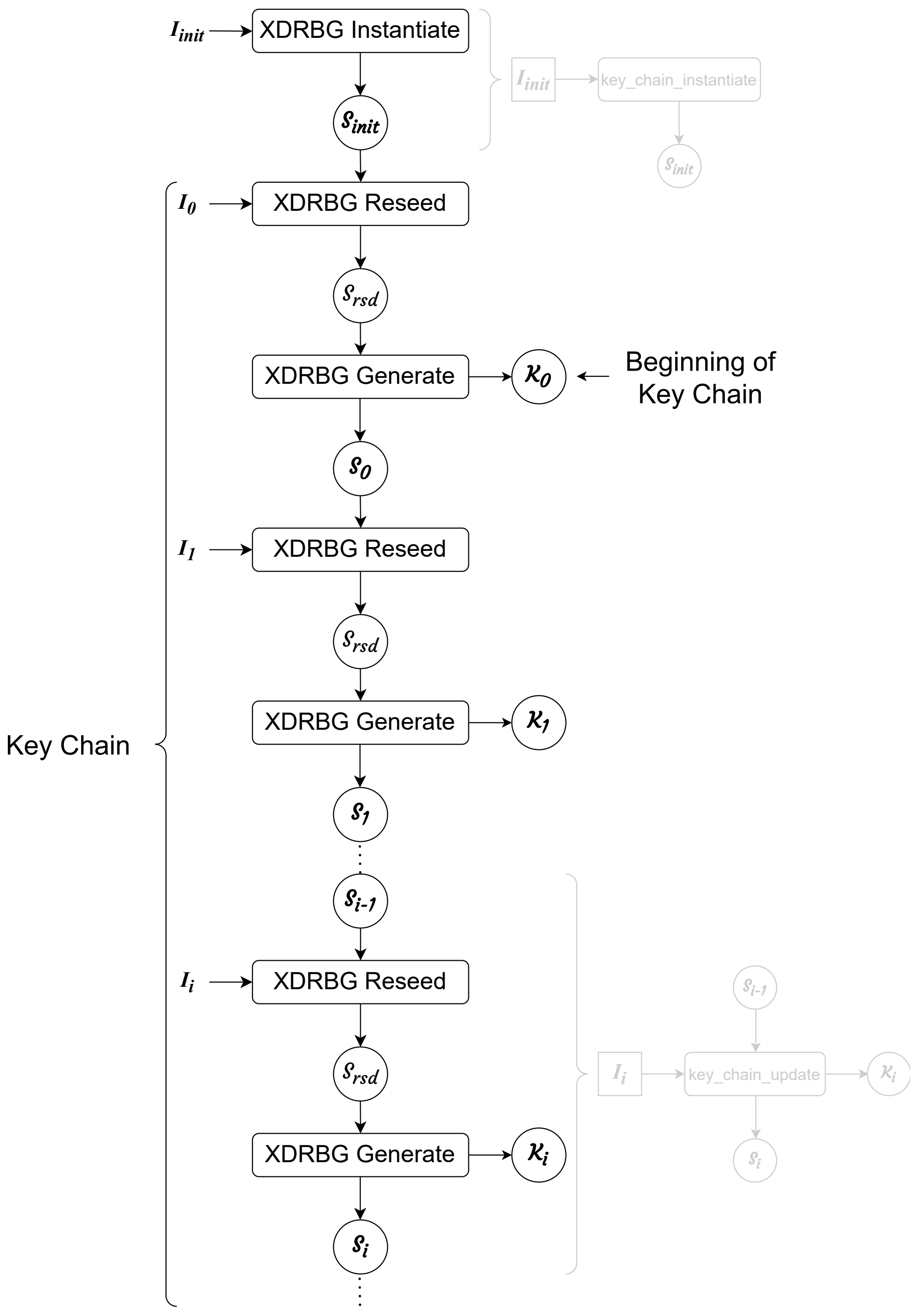

Figure 4.2: XDRBG-Based Key Chain

The maxout Value in XDRBG: A *maxout* value is associated with each GENERATE call to ensure that the desired length l of the random output $\mathcal{K}$ is such that it does not take a single GENERATE call too much time to produce that output $\mathcal{K}$ of the length l. This is to ensure that there is not an XDRBG state compromise within that duration. The maxout value enforces a limit on the total output $T \leftarrow l + |\mathcal{S}|$ (as mentioned in Algorithm 2) that can be generated from a single XDRBG GENERATE call. We have chosen three different XOFs namely SHAKE-128, SHAKE-256 and ASCON-XOF to implement the XDRBG which must abide by the parameters mentioned in Table 4.4 [7].

Table 4.4: Rules For Parameters Size in Bytes for XDRBG-Based Key Chain [7]

Name of XDRBG	XDRBG Based On	Rate (r)	Capacity (c)	H_{init}	H_{rsd}	maxout (T)	Output State Size $\|\mathcal{S}\|$	Random Output Size l	Approximate Security Level	
									Classical	Quantum
XDRBG-128	SHAKE-128	168	32	≥ 24	≥ 16	304	32	≤ 272	16	8
XDRBG-256	SHAKE-256	136	64	≥ 48	≥ 32	344	64	≤ 280	32	16
XDRBG-L128	ASCON-XOF	8	32	≥ 24	≥ 16	256	32	≤ 224	16	8

Clarification for the Design Decision of the XDRBG-Based Key Chain

Here, we wish to explicitly provide clarification for any confusion regarding the resemblance and consistency of the XDRBG-based key chain in Figure 4.2 and the standard interface of the key chain in Figure 1.1. With respect to the Figure 1.1, it is evident that each $key\_chain\_update(I_i, \mathcal{S}_{i-1})$ call after the establishment of the initial state $\mathcal{S}_{init}$ takes the current state of the key chain (say) $\mathcal{S}_{i-1}$ and an input parameter I_i as inputs to generate a random output $\mathcal{K}_i$ which can be used for cryptographic purposes and a (new) state $\mathcal{S}_i$ of the key chain which will be persistently stored in a storage. Now, this state $\mathcal{S}_i$ will act as input to the next $key\_chain\_update(I_i, \mathcal{S}_{i-1})$ call in addition to the input parameter I_{i+1}. But, for the XDRBG-based key chain, we show that for each call to $key\_chain\_update(I_i, \mathcal{S}_{i-1})$ it performs an XDRBG RESEED at first and then an XDRBG GENERATE. This is because, it is the only way to fulfil the interface compliance requirement from section 3.1 for a key chain based on XDRBG. The state of the key chain generated from the RESEED call denoted as $\mathcal{S}_{rsd}$ in Figure 4.2 is not persistently stored, but is used as the input to the forthcoming GENERATE call which takes this (currently) reseeded state of the key chain as input to generate a random output $\mathcal{K}_i$ and a (new) XDRBG state $\mathcal{S}_i$. This combination of exactly one RESEED before one GENERATE reflects one $key\_chain\_update(I_i, \mathcal{S}_{i-1})$ call while simultaneously ensuring that, before each GENERATE call, the state of the key chain is replenished with a certain amount of minimum entropy (which in this case is SD_{rsd} of H_{rsd} bits).

It is likely to come off a bit weird at first glance to Figure 4.2 that there is a RESEED call in addition to an INSTANTIATE call in the beginning. We acknowledge that the $\mathcal{S}_{init}$ is already derived using a seed SD_{init} with a minimum entropy of H_{init} bits and this must fulfil the requirement of starting the key chain after establishing an initial state of the key chain with a certain amount of minimum entropy. So, one might think of performing the GENERATE call right after the establishment of the $\mathcal{S}_{init}$, but in order to be compliant with the interface compliance requirement from section 3.1, we obtain the input parameter I_0 and perform the XDRBG RESEED after the INSTANTIATE call as well at the beginning of our key chain construction, and we naturally end up doing exactly one RESEED before each GENERATE throughout the key chain.

4.4 Key Chain Based on PRG

With respect to the standard interface of the key chain in Figure 1.1, the PRG operations in Algorithm 3 and our vision of the PRG-based key chain in Figure 4.3:

1. As we mentioned in subsection 3.2.3, the initial PRG state $\mathcal{S} \leftarrow \{0\}^{\lambda}$ is a bit string

of all zeroes of length equal to the security parameter λ. Thus, considering this state as the initial state $\mathcal{S}_{init}$ of the key chain does not seem to be a wise decision from the security perspective, as it is certain that if we use this $\mathcal{S}_{init}$ (which is the bit string of all zeroes) as an input to a NEXT call, the output generated from this NEXT call will definitely be predictable for an adversary $\mathcal{A}$.

2. For the key chain instantiation, we use the $key\_chain\_instantiate(I_{init})$ method to generate an initial state $\mathcal{S}_{init}$. This invokes a REFRESH call to the PRG with the state $\mathcal{S} \leftarrow \{0\}^{\lambda}$ and I_{init} which are $\mathcal{S}'$ and X respectively (according to Algorithm 3) to generate a (refreshed) PRG state denoted as $\mathcal{S}_{init}$ in Figure 4.3. This serves as the initial (and ideally it is expected to be a secure) state of the cryptographic key chain itself and afterwards the key chain starts.

3. For our PRG-based key chain construction in Figure 4.3 and the Algorithm 3, we consider the parameter I_{init} (of n bytes) and all the other input parameters $I_i, i \in \{0, 1, 2...\}$ (of m bytes) to be the parameter $X \leftarrow \text{EXTRACT}(x)$. In our construction, the length of the parameter X is mentioned in Table 4.5 for $\lambda = \{16, 24, 32\}$ bytes ({128, 192, 256} bits). As an important note, we wish to mention that the length of X must always be equal to the choice of λ according to the EXTRACT function defined in subsection 3.2.3 which means that $n = m = \lambda$.

4. After the establishment of $\mathcal{S}_{init}$, the $key\_chain\_update(I_i, \mathcal{S}_{i-1})$ method is used which invokes a single call to the PRG and it involves the following steps:
   a) At first it invokes the REFRESH call to the PRG which takes an input parameter, i.e., I_i and the current state of the key chain (say) $\mathcal{S}_{i-1}$ as inputs which are X and $\mathcal{S}'$ respectively (according to Algorithm 3) and produces $\mathcal{S}_{rd}$ as the output which is the refreshed state of the PRG and it serves as the current state of the key chain as well.
   b) Then, it invokes a NEXT call to the PRG which takes the current state of the key chain, i.e., $\mathcal{S}_{rd}$ as the input and produces the total output T comprising of:
      i. $\mathcal{K}_i$ of size λ bytes which represents the random output $\mathcal{K}$ suitable to be used for cryptographic purposes.
      ii. $\mathcal{S}_i$ of size λ bytes (in Figure 4.3) which is the same as the new state $\mathcal{S}$ of the PRG (according to Algorithm 3) and this serves to be the (new) state of the key chain as well. Following the analogy mentioned as the *persistent storage requirement* in section 3.1, this parameter $\mathcal{S}_i$ will be persistently stored because it was generated as part of the total output T from the PRG NEXT call from which the corresponding key $\mathcal{K}_i$ was also generated. This state $\mathcal{S}_i$ will act as the starting point for the generation of the next cryptographic key $\mathcal{K}_{i+1}$ in the key chain because $\mathcal{S}_i$ will be used as an input to the next (REFRESH) call to the PRG.

5. Over time, the key chain using PRG is formed by the chronological repetition of all the steps from the preceding point (item 4).

We have not explicitly provided a clarification for the design decision of the PRG-based key chain, because, for the PRG-based key chain, a similar clarification like the one we mentioned above for the XDRBG-based key chain also intuitively holds.

Table 4.5: Input Parameter Size in Bytes for PRG-Based Key Chain

Security Parameter	X (I_{init} of n bytes and I_i of m bytes)
$\lambda = 16$	16
$\lambda = 24$	24
$\lambda = 32$	32

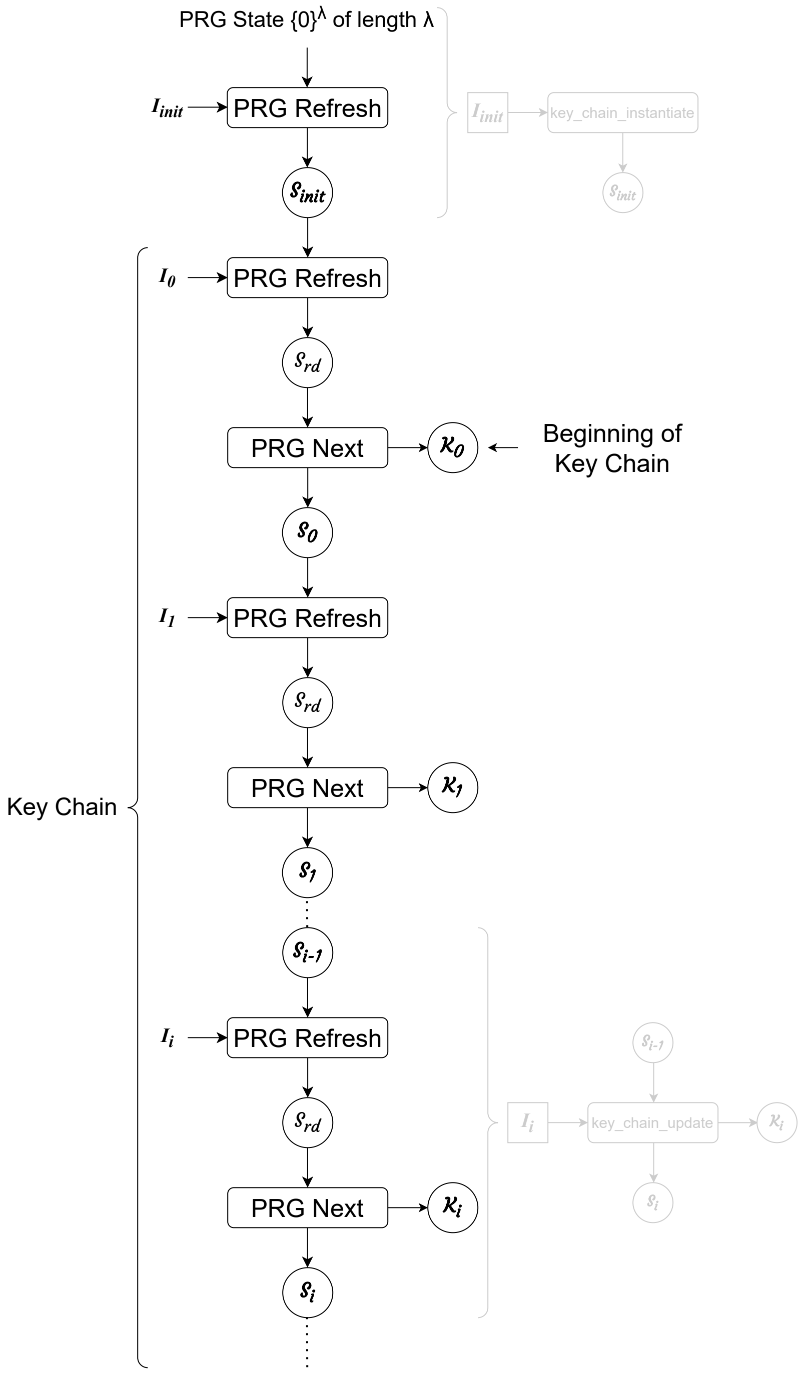

Figure 4.3: PRG-Based Key Chain

Clarification for the Security Parameter λ Being Only Equal to the Acceptable Key Sizes for AES Counter Mode

We mentioned the block size, (input) key size and the number of rounds for AES in Table 2.1 and that we use AES Counter Mode as the generator function G is also mentioned in subsection 3.2.3. So, the exactly same parameter bounds with respect to the (input) key size is applicable for G as well, i.e., the input key to G must be of length 16 or 24 or 32 bytes (128 or 192 or 256 bits respectively). Considering a scenario where the security parameter $\lambda = 48$ bytes (384 bits), then the initial PRG state being a bit string of all zeroes (in Figure 4.3) will be of length 48 bytes (384 bits). Now, when the REFRESH call is made to the PRG, the generator function G will take $(\mathcal{S}' \oplus X)$ as the input which is basically the input key to the AES block cipher in counter mode. As this state $\mathcal{S}'$ is a bit string of all zeroes of length 48 bytes (384 bits) and we know from subsubsection 3.2.3 that X will also be of length equal to λ, then the output value resulting from the computation $\mathcal{S}' \oplus X$ will also be of length 48 bytes (384 bits) which does not conform to an acceptable length of the input key to AES block cipher in counter mode according to Table 2.1. Thus, when using AES block cipher in Counter Mode as the generator function G, we must always choose the security parameter λ to be any of the value from $\{16, 24, 32\}$ bytes (or $\{128, 192, 256\}$ bits respectively).

Chapter 5

Evaluation

In this chapter, initially, we provide an overview of the system setup where we have conducted the performance evaluation, and then we proceed onto the details of the performance evaluation of the three cryptographic key chains (mentioned in section 4.2, section 4.3 and section 4.4) extensively. Then we also discuss regarding fulfillment of the security requirements (from section 3.1) by the different key chains along with an evaluation of the entropy loss.

5.1 Performance Evaluation

We have chosen to perform the whole evaluation of the cryptographic key chain based on HKDF [6], XDRBG [7] and PRG [8] by implementing it in Python 3.12.3 on a Dell Inspiron 15 3511 laptop having Windows 11 64-bit environment, 11$^{\text{th}}$ Gen Intel(R) Core(TM) i5-1135G7 @ 2.40 GHz processor, 8.00 GB DDR4 RAM (7.74 GB usable). We constantly mention throughout this report regarding storing the state of the cryptographic key chain in a persistent storage, which in our case is an SQLite Database having SQLite version 3.35.5 existing on a storage device with the following specifications:

- Model Number: WDC WD10SPZX-75Z10T3
- Media Type: Fixed Hard Disk
- Capacity: 931.51 GB with 2 Partitions
    1. Partition #0 Size: 491.33 GB (The database is stored in this partition.)
    2. Partition #1 Size: 440.06 GB
- Bytes per Sector: 512
- Total Cylinders: 121,601
- Total Sectors: 1,953,520,065
- Total Tracks: 31,008,255

We have implemented the XDRBG [7] and the PRG [8], whereas we have used the open source implementation of the HKDF [30], ASCON-XOF [33] and Circulant [34]. We have used the standard python library *hashlib* for all the hash functions, SHAKE-128 and SHAKE-256 XOF, but we have explicitly installed the *PyCryptodome* module for AES block cipher in counter mode. We have made some modifications to the open source implementation of HKDF [30] to fit our needs, but these changes do not alter the fundamental logic of how HKDF inherently works in any manner (which is indeed obtained from their implementation). The changes are as follows:

1. We have changed most of the variable names.

2. We have taken an object-oriented approach as opposed to their procedural approach.
3. Additionally, we have provided extensive comments which also align with the texts mentioned in this work while simultaneously providing proper *python docstrings* for the different methods and proper type annotations for better code readability.

For our evaluation, we compare the performance of the two cryptographic primitives for generating a key chain only if they provide the same level of classical security. By classical security, we mean that the cryptographic primitive provides a certain level of security against adversaries having access to only classical computers and not quantum computers. The performance evaluation mentioned in Table 5.1 represents the average execution time in seconds for generating 100 key chains using each cryptographic primitive, where each key chain comprises of 100 cryptographic keys $\mathcal{K}_i$. In addition to the average execution times, we also provide the standard deviation of the execution times and the respective confidence interval of the average execution time with a 95% confidence level. We wish to explicitly mention that the values mentioned in Table 5.1 are rounded off to four decimal places and the average execution times are only for the $key\_chain\_update(I_i, \mathcal{S}_{i-1})$ call as this is the one which generates the output key $\mathcal{K}_i$, so the timings mentioned in Table 5.1 do not account for time required for the $key\_chain\_instantiate(I_{init})$ call. For further insights into the key chain instantiation using $key\_chain\_instantiate(I_{init})$, we have provided the average execution times (over 100 iterations) in seconds in Table 5.4 and the values are only represented up to 6 digits after the decimal place.

We mentioned in section 4.1, for our key chain construction we use Circulant (from subsection 2.4.2) to generate the input parameters, namely I_{init} and $I_i, i \in \{0, 1, 2, \dots\}$. We also mentioned in the previous paragraph that in each key chain there are 100 cryptographic keys, i.e., for each key chain the $key\_chain\_update(I_i, \mathcal{S}_{i-1})$ method is invoked 100 times. So, invocation of $key\_chain\_update(I_i, \mathcal{S}_{i-1})$ 100 times means that we need 100 input parameters $I_i, i \in \{0, 1, 2, \dots, 99\}$ for generating one key chain. As we generate 100 such key chains (using each cryptographic primitive), this means that the $key\_chain\_update(I_i, \mathcal{S}_{i-1})$ method will be invoked $100 \times 100 = 10,000$ times, which signifies that we need 10,000 input parameters $I_i, i \in \{0, 1, 2, \dots, 9999\}$ altogether for generating 100 key chains (using each cryptographic primitive). The data provided in Table 5.2 represents only the time (averaged over 100) needed for extracting the input parameters $I_i, i \in \{0, 1, 2, \dots, 9999\}$ using Circulant required for the $key\_chain\_update(I_i, \mathcal{S}_{i-1})$ method for generating 100 key chains (using each cryptographic primitive). We wish to explicitly mention that these values are rounded off to four decimal places. Although these timings should not differ by a lot (and ideally it should not differ at all) irrespective of whether we are persistently storing the state $\mathcal{S}_i$ of the chain, but we have still calculated these timings for both, i.e., when we are persistently storing and when we are not. Additionally, we also provide the average execution times in seconds required for the all the other computations (which are mostly the operations of the cryptographic primitives and some non-cryptographic operations) in Table 5.3 for generating 100 key chains. We do so by calculating the difference between the average execution time for generating the key chains (from Table 5.1) and the average execution time needed for extracting the input parameters (from Table 5.2) for each individual cryptographic primitive separately for both, i.e., when we are persistently storing and when we are not.

Table 5.1: Average Execution Time in Seconds to Generate 100 Key Chains Along With the Standard Deviation in Execution Times and Confidence Interval With 95% Confidence Level

Cryptographic Primitive		Without Persistent Storage			With Persistent Storage		
		Average Execution Time	Standard Deviation	Confidence Interval	Average Execution Time	Standard Deviation	Confidence Interval
HKDF	SHA-256	0.0340	0.0059	(0.0328, 0.0352)	12.7648	0.6034	(12.6465, 12.8830)
	SHA3-256	0.0330	0.0050	(0.0320, 0.0340)	13.0969	0.5877	(12.9817, 13.2121)
	SHA-512	0.0652	0.0062	(0.0640, 0.0664)	11.9055	0.4941	(11.8087, 12.0024)
	SHA3-512	0.0659	0.0064	(0.0647, 0.0672)	11.4135	0.5073	(11.3141, 11.5129)
XDRBG	SHAKE-128	0.0221	0.0075	(0.0206, 0.0236)	11.2929	0.5116	(11.1927, 11.3932)
	SHAKE-256	0.0440	0.0067	(0.0427, 0.0453)	12.6062	0.7317	(12.4628, 12.7496)
	ASCON-XOF	0.1522	0.0094	(0.1504, 0.1540)	12.7126	0.5648	(12.6019, 12.8233)
PRG	$\lambda = 16$	0.0236	0.0089	(0.0218, 0.0254)	11.4259	0.3954	(11.3484, 11.5034)
	$\lambda = 24$	0.0272	0.0074	(0.0257, 0.0286)	11.2193	0.4755	(11.1261, 11.3125)
	$\lambda = 32$	0.0385	0.0080	(0.0370, 0.0401)	10.9645	0.2123	(10.9229, 11.0061)

Table 5.2: Average Time in Seconds to Extract Input Parameters from Circulant for 100 Key Chains

Cryptographic Primitive		Without Persistent Storage	With Persistent Storage
HKDF	SHA-256	0.0327	0.0970
	SHA3-256	0.0315	0.1026
	SHA-512	0.0633	0.1966
	SHA3-512	0.0650	0.1822
XDRBG	SHAKE-128	0.0217	0.0569
	SHAKE-256	0.0435	0.1368
	ASCON-XOF	0.0225	0.0689
PRG	$\lambda = 16$	0.0190	0.0512
	$\lambda = 24$	0.0242	0.0732
	$\lambda = 32$	0.0361	0.1125

Table 5.3: Average Time in Seconds For the Cryptographic and the Non-Cryptographic Operations to Generate 100 Key Chains

Cryptographic Primitive		Without Persistent Storage	With Persistent Storage
HKDF	SHA-256	0.0013	12.6678
	SHA3-256	0.0015	12.9943
	SHA-512	0.0019	11.7089
	SHA3-512	0.0009	11.2313
XDRBG	SHAKE-128	0.0004	11.2360
	SHAKE-256	0.0005	12.4694
	ASCON-XOF	0.1297	12.6437
PRG	$\lambda = 16$	0.0046	11.3747
	$\lambda = 24$	0.0030	11.1461
	$\lambda = 32$	0.0024	10.8520

5.1.1 Interpretation of Performance Evaluation Without Persistent Storage

The *without persistent storage* column in the Table 5.1 depicts the performance evaluation when we are not persistently storing the state $\mathcal{S}_i$ of the key chain (from item 2(b)ii of section 4.2, item 3(b)ii of section 4.3 and item 4(b)ii of section 4.4 for HKDF, XDRBG and PRG respectively). As we mentioned earlier that we compare the performance of the cryptographic primitives for generating a key chain only if they provide the same level of classical security, we state the comparisons below along with our interpretation of the performance results for the key chain constructions.

Comparison of Cryptographic Primitives Providing 16 Bytes of Classical Security

We compare the SHAKE-128, ASCON-XOF based XDRBG and PRG with the security parameter $\lambda = 16$ as they provide 16 bytes (128 bits) of classical security. It is also explicitly mentioned in Table 4.4 for SHAKE-128 and ASCON-XOF, but for the PRG with security parameter $\lambda = 16$, the column *key length in bits* in Table 2.1 also refers to the security provided by the AES variant. In this case, for the XDRBG and the PRG-based key chain, we have also considered the size of the output key $\mathcal{K}_i$ (from item 3(b)i of section 4.3 and item 4(b)i of section 4.4 respectively) to be 16 bytes (128 bits). If we observe the average execution times of these three cryptographic primitives in Table 5.1, it is evident that the XDRBG-based key chain with SHAKE-128 XOF is the most suitable cryptographic primitive, as it takes the lowest average execution time of 0.0221 seconds.

The reason behind the XDRBG key chain based on the SHAKE-128 XOF being the most efficient, is because of its rate r (mentioned in Table 4.4). We mentioned in section 2.1 that the rate r of a sponge function basically determines its efficiency. Although there is no explicit terminology regarding the rate of AES block cipher (in counter mode), but we know from Table 2.1 that the AES block cipher can process an input block of only and exactly 128 bits (16 bytes) which can be considered equivalent to the rate of AES block cipher (based on our understanding). Then, based on the data mentioned in Table 2.1 (in column block size) and Table 4.4 (in column rate), it is apparent that the SHAKE-128 XOF has the highest rate r of 168 bytes (1344 bits) amongst the three primitives which naturally makes it the most efficient cryptographic primitive when generating the key chain where the output keys $\mathcal{K}_i$ provide 16 bytes (128 bits) of classical security. Another example of the sheer high rate of SHAKE-128 XOF can also be referred from the fact that, the average execution times mentioned in Table 5.1 also accounts for the time needed to extract 10,000 input parameters (from Circulant) for generating 100 key chains using the different cryptographic primitives. In Table 5.2, SHAKE-128 XOF takes 0.0217 seconds for extracting (10,000) input parameters of size 24 bytes (mentioned in Table 4.3), which is more than the time needed for PRG with security parameter $\lambda = 16$, i.e., 0.0190 seconds for extracting (10,000) input parameters of size 16 bytes (mentioned in Table 4.5). Irrespective of this, the XDRBG key chain based on the SHAKE-128 XOF showed the most efficient performance, which is purely because of such a high rate of the cryptographic primitive, but we can still say that the performance of PRG with security parameter $\lambda = 16$ is somewhat comparable to that of XDRBG key chain. We do not comment much about the ASCON-XOF because it takes the most amount of time, thereby not being an efficient choice for generating the key chains. This is reasonable because, it has the lowest rate amongst the three cryptographic primitives while simultaneously having to extract (10,000) input parameters of size 24 bytes (mentioned in Table 4.3).

Comparison of Cryptographic Primitives Providing 32 Bytes of Classical Security

Here we compare the SHAKE-256 based XDRBG, and SHA-256, SHA3-256 based HKDF and PRG with the security parameter $\lambda = 32$ as they provide 32 bytes (256 bits) of classical security. It is mentioned in Table 4.4 for SHAKE-256 and in Table 2.1 for the PRG with security parameter $\lambda = 32$. For the HKDF with SHA-256 and SHA3-256 as the hash functions, the digest size d (mentioned in Table 4.2) also refers to the security provided by the cryptographic primitive. In this case, for the HKDF, XDRBG and the PRG-based key chain, we have considered the size of the output key $\mathcal{K}_i$ (from item 2(b)i of section 4.2, item 3(b)i of section 4.3 and item 4(b)i of section 4.4 respectively) to be 32 bytes (256 bits). If we directly observe the average execution times in Table 5.1, undoubtedly the HKDF key chain based on SHA3-256 has the least execution time of 0.0330 seconds, thus being the best choice for generating the key chains. For a hash function (be it from the SHA-2 [13] or SHA-3 [14] family), the block size b of the hash function determines its efficiency, which can also be considered as the rate r of the hash function (like we mentioned in subsection 2.2.1). Now, if we observe the rate of SHAKE-256 in Table 4.4, SHA-256 and SHA3-256 in Table 4.2 (in column block size) and for PRG with security parameter $\lambda = 32$ in Table 2.1 (in column block size), naturally the rate of PRG is the lowest which is 16 bytes (also mentioned in the previous paragraph). As is evident from Table 4.2 and Table 4.4, the rate of SHA3-256 and SHAKE-256 is the same, i.e., 136 bytes (which is the highest amongst these four cryptographic primitives). But, the reason that SHA3-256 performs better than SHAKE-256 is because, for the XDRBG key chain based on the SHAKE-256 XOF we indeed extract 10,000 input parameters (from Circulant) which must be of size 48 bytes (according to Table 4.3) which takes 0.0435 seconds, whereas for the HKDF key chain based on SHA3-256, we extract 10,000 input parameters (from Circulant) of size 32 bytes (according to Table 4.1) which only takes 0.0315 seconds (as mentioned in Table 5.2). This is why the HKDF key chain based on SHA3-256 has the least average execution time amongst these 4 cryptographic primitives where the output key $\mathcal{K}_i$ provides 32 bytes (256 bits) of classical security.

Comparison of Cryptographic Primitives Providing 64 Bytes of Classical Security

Similarly, we also compare SHA-512 and SHA3-512 based HKDF as they (are the only ones who) provide 64 bytes (512 bits) of classical security which is based on the digest size d from Table 4.2. In this case, for the HKDF-based key chain, we have considered the size of the output key $\mathcal{K}_i$ (from item 2(b)i of section 4.2) to be 64 bytes (512 bits). Based on the data from Table 5.1, amongst these two cryptographic primitives, the HKDF key chain based on SHA-512 stands out to be the better choice as it takes a comparatively lower average execution time. Here, the size of the input parameters is 64 bytes (according to Table 4.1), so the difference in the average execution time is primarily because of the rate of SHA-512, which is comparatively higher than that of SHA3-512 as mentioned in Table 4.2.

The Only Cryptographic Primitive Providing 24 Bytes of Classical Security

We do not provide any comparison for the PRG with security parameter $\lambda = 24$ because there are no other cryptographic primitives (amongst the ones discussed in this work) which provide 24 bytes of classical security. The size of the output key $\mathcal{K}_i$ (from item 4(b)i of section 4.4) with this cryptographic primitive would be 24 bytes and indeed it is possible to generate 24 bytes of output from the HKDF and the XDRBG, but it would never provide exactly 24 bytes of classical security.

Additional Note for the Performance Evaluation of SHA3-256-Based HKDF and SHAKE-256-Based XDRBG Key Chain

Earlier, we provided a comparison between the different cryptographic primitives providing 32 bytes of classical security where we observed that the HKDF key chain based on the SHA3-256 hash function performed the best. Irrespective of SHA3-256 and SHAKE-256 XOF having the same rate (136 bytes), the performance of the XDRBG key chain based on SHAKE-256 was not at par with the performance of the HKDF key chain based on SHA3-256 because of the usage of Circulant for extracting 10,000 input parameters. But, there is more to this scenario than meets the eye. If we observe the timings mentioned in Table 5.3 for SHAKE-256 and SHA3-256, the average execution time required for (mostly cryptographic and some non-cryptographic) operations of the HKDF key chain based on SHA3-256 is three times more than the time needed by the XDRBG key chain based on SHAKE-256. This indicates that, as a cryptographic primitive, the XDRBG key chain based on SHAKE-256 XOF is more efficient than the HKDF key chain based on SHA3-256 if the usage of Circulant is set aside. This is just a hint for the reader to enable them to make an informed decision regarding exclusively deciding on a cryptographic primitive, setting aside the randomness extractor used for the input parameter extraction.

Table 5.4: Average Time in Seconds for Key Chain Instantiation

Cryptographic Primitive		Average Execution Time
HKDF	SHA-256	0.000010
	SHA3-256	0.000011
	SHA-512	0.000010
	SHA3-512	0.000011
XDRBG	SHAKE-128	0.000002
	SHAKE-256	
	ASCON-XOF	0.000520
PRG	16	0.000012
	24	
	32	

5.1.2 Interpretation of Performance Evaluation With Persistent Storage

The *with persistent storage* column in Table 5.1 depicts the performance evaluation while we are persistently storing the state $\mathcal{S}_i$ of the key chain (as mentioned in item 2(b)ii of section 4.2, item 3(b)ii of section 4.3 and item 4(b)ii of section 4.4 for the HKDF, XDRBG and PRG respectively). We choose not to explicitly segregate based on the (classical) security levels and argue regarding any one particular cryptographic primitive being the best choice for generating the key chain, because, the overall performance (when persistently storing) is limited by the choice of the storage device. This acts as a bottleneck and the overhead of storing the state $\mathcal{S}_i$ (of the key chain) in the storage device (which in our case is the database) should be roughly the same for all the different cryptographic primitives provided the state size $\mathcal{S}_i$ (of the key chain) generated by each of the cryptographic primitive does not differ by a lot. This data primarily reflects that how the (performance and) usage of such a key chain will look like in the real world.

5.2 Security Discussions

In this section, we consider the key chain constructions based on the different cryptographic primitives and provide a reasoning about the fulfilment of the *security requirements a-c* from section 3.1. We discuss the *security requirement d* in section 5.3.

5.2.1 For the HKDF-Based Key Chain

Although it is not directly and explicitly mentioned in [6] about the properties (also referred as the security requirements from section 3.1) that an HKDF fulfils, but, it is explicitly mentioned in [6] that the idea of the extract-then-expand KDF is taken from the PRG architecture mentioned in [8] where it uses a generator function G taking a parameter X of length λ (which is obtained from a randomness extractor) and generates an output of size 2λ (which can be considered as expanding part). We know that the Double Ratchet algorithm [5] uses HKDF-based key chains, and it is explicitly mentioned in [5] that such a key chain fulfils forward secrecy, backward secrecy and resilience (exactly the ones mentioned in section 3.1), which leads us to conclude that the HKDF-based key chain indeed fulfils the *security requirement a, b and c.*

5.2.2 For the XDRBG-Based Key Chain

We know from subsection 3.2.2 that XDRBG [7] fulfils forward and backward secrecy, so it is natural that the XDRBG-based key chain also fulfils the same, i.e., we can directly conclude that the XDRBG-based key chain will fulfil *security requirement a and b.* With respect to the resilience requirement in section 3.1, nothing is explicitly mentioned in [7] regarding the XDRBG being able to fulfil it or not. But, based on our (limited) understanding of the security game in *section 4* of [7], we believe that XDRBG should fulfil the resilience requirement as well. This is because, fulfilment of backward secrecy ensures that even after a state compromise, if there is a RESEED (or INSTANTIATE) call (even if the input is partially controlled by $\mathcal{A}$ according to *section 4* of [7]) the adversary $\mathcal{A}$ will be completely oblivious to the newly generated XDRBG state. Now, if this state is used as an input to the XDRBG GENERATE call, the generated output $\mathcal{K}$ should also seem completely indistinguishable from random to $\mathcal{A}$. Based on this reasoning, we can say that the XDRBG-based key chain also fulfils *security requirement c.*

5.2.3 For the PRG-Based Key Chain

We know from subsection 3.2.3 that PRG [8] fulfils forward secrecy, backward secrecy and resilience, so it is natural that the PRG-based key chain also fulfils the same, i.e., the PRG-based key chain will fulfil *security requirement a, b and c.*

5.3 Evaluation of Entropy Loss

According to [28], in information theory, entropy refers to the degree of unpredictability or randomness of a particular data. If a particular piece of data (say a bit string $\{0,1\}^*$ of random length) does not have sufficient amount of entropy, then this particular value is considered to be predictable. As entropy is a vital component of cryptographic primitives such as XDRBG, PRG etc., lack of the same leaves a cryptosystem unsecure where such a cryptographic primitive is used. This means that, without a good source of entropy, cryptographic primitives can become vulnerable to attacks with respect to predictability of the generated output [28].

In the implementation of a PRG or an XDRBG, bugs in the code or an inherently flawed design may lead to *entropy loss.* As these cryptographic primitives are extremely difficult to design, implement and debug, ideally it would be delightful if there were a way to detect and *quantify* said entropy loss, but we have not yet come across any academic work who aim to quantify the entropy loss. But, the authors of [35] provided a novel static analysis technique in the form of a tool named *Entroposcope* (to which we had no access) specifically for quality assurance of any cryptographic Pseudorandom Number Generator (PRNG). This tool is used for *detecting* (and not quantifying) any entropy loss occurred during the usage of the PRNG, i.e, starting from seeding the PRNG to generating the pseudorandom stream of output. According to [35], entropy loss occurs in PRNGs in either of the two scenarios:

1. The entropy contained in the seed with which the PRNG is seeded is not fully utilized to generate the pseudorandom output stream.
2. Two different seeds lead to the generation of the same pseudorandom output stream.

5.3.1 The Analysis Methodology

The authors of [35] also consider a PRNG to be built of certain cryptographic and non-cryptographic parts, like we mentioned in subsection 3.2.3 regarding a PRG built with a (cryptographic) generator function G which is the AES block cipher in counter mode and the (non-cryptographic) randomness extractor defined as the EXTRACT function. A PRNG function defined by [35] is as follows:

$$g : \{0,1\}^m \to \{0,1\}^n; n \geq m$$

which resembles the generator function G defined in subsection 3.2.3. According to [35], claiming that no seed entropy is lost is the same as establishing that the aforementioned PRNG function g is *injective*, i.e., for the same PRNG when seeded with two different seeds, it must not lead to the generation of the same pseudorandom output. This can also be described as [35]:

$$\begin{gathered}
g(\text{seed}_1) \to \text{output}_1;\ \text{seed}_1 \in \{0,1\}^m \text{ and output}_1 \in \{0,1\}^n \\
g(\text{seed}_2) \to \text{output}_2;\ \text{seed}_2 \in \{0,1\}^m \text{ and output}_2 \in \{0,1\}^n \\
\text{where} \begin{cases} \text{output}_1 = \text{output}_2 & \text{if seed}_1 = \text{seed}_2, \\ \text{output}_1 \neq \text{output}_2 & \text{if seed}_1 \neq \text{seed}_2 \end{cases}
\end{gathered}$$

In [35], the authors focus on the non-cryptographic part of a PRNG for the detection of entropy loss, as opposed to the cryptographic part. This is because, the authors assume that the cryptographic part does not contribute to any entropy loss as they are (generally) some standardized primitives, e.g., the usage of SHA-1 hash function as the function g (in [35]) or AES block cipher in counter mode as the (cryptographic) generator function G in the PRG architecture from [8] mentioned in subsection 3.2.3. However, the authors emphasize that the non-cryptographic part (like the EXTRACT function) is not standardized and is prone to errors. The fact that the non-cryptographic part (like the EXTRACT function) is not standardized is also evident from subsection 3.2.3 where we mentioned that the authors of [8] did not specify a mandate regarding the choice of this EXTRACT function needed for a PRG following the architecture from [8].

5.3.2 The Two Idealizations

Based on our understanding, here we describe about the two idealizations that the authors of [35] consider for the cryptographic part, i.e., for the function g in order to successfully do the detection of entropy loss. In [35], the authors have provided an example of the idealizations with the OpenSSL PRNG which uses three static invocations of SHA-1 hash function, which is the cryptographic part. But, here we (try to) correlate and provide the example of these idealizations with the PRG architecture from [8] mentioned in Algorithm 3 for ease of understanding. We also wish to redirect a reader to *section 4.1, 6.3, and 6.4* of [35] should they choose to go through the examples of the actual idealizations for the OpenSSL PRNG.

We have observed (in Algorithm 3) that there are two operations, namely NEXT and REFRESH to interact with the PRG following the architecture from [8]. And, in each invocation of the PRG NEXT call and the PRG REFRESH call, we see the usage of the (cryptographic) generator function G which is the AES block cipher in counter mode and for the PRG REFRESH call we additionally see the usage of the (non-cryptographic) EXTRACT function of whose output, i.e., X acts as the seed (which is the source of additional entropy) for the PRG REFRESH call.

The Unsound Idealization

In this type of idealization, the (cryptographic) generator function G is depicted to act like a simple *projection* instead of a complex function.

- Purpose: This approach allows a user to easily identify possible issues with the PRG as it helps in tracking the information flow through the PRG, i.e., from seeding to generating the pseudorandom output stream.
- Workflow: With respect to the PRG REFRESH call, it simplifies the complex (cryptographic) generator function G by considering only part of the input, i.e., the part which is responsible for providing additional entropy X (obtained from the non-cryptographic EXTRACT function) while discarding the rest of the input, i.e., $\mathcal{S}'$. It will allow a user to identify defects quickly by visually observing occurrences of the same concrete bit pattern from the seed to the output by using Entroposcope's counterexample visualizer [35]. Although this method is easier to interpret for the user, it is not guaranteed to always replicate the correct behaviour of the original complex function had it used both the inputs, i.e., X and $\mathcal{S}'$ thereby leading to incorrect conclusions at times.

The Sound Idealization

In this type of idealization, the (cryptographic) generator function G is expected to be injective (as we mentioned earlier about the function g) [35].

- Purpose: This approach primarily aims to ensure that the entropy loss analysis results are sound and correct.
- Workflow: The assumption of the injectivity axiom is stricter and theoretically safer. This is because, in this idealization, all the inputs to the function G in the PRG REFRESH call are considered instead of considering only part of the input. If two (distinct) seeds do not lead to the same output from the generator function

G, then we can conclude with certainty that there is no entropy loss. In the sound idealization, the analysis is also performed using Entroposcope's counterexample visualizer [35].

5.3.3 Applying the Idealizations to the Cryptographic Primitives

In this subsection, we describe that how we have applied the two idealizations for the cryptographic primitives, namely, HKDF, XDRBG, and PRG. We have only applied the idealization for HKDF, XDRBG, and PRG instead of the $key\_chain\_update(I_i, \mathcal{S}_{i-1})$ method, because, there is no post-processing of the output generated from the cryptographic primitives, i.e., HKDF, XDRBG, and PRG in our key chain constructions. So, as long as these cryptographic primitives are injective, i.e., they do not exhibit any entropy loss, this means that there will not be any entropy loss when using the $key\_chain\_update(I_i, \mathcal{S}_{i-1})$ method as well to generate the output key $\mathcal{K}_i$.

The Unsound Idealizations

Here, we consider Algorithm 2, and Algorithm 3 and demonstrate the changes that we have made to realize the unsound idealization for the XDRBG, and PRG respectively.

The Unsound Idealization for XDRBG: The Algorithm 4 defines the modified XDRBG RESEED operation to align with the unsound idealization method where the cryptographic part is the xof.

Algorithm 4 Modified XDRBG RESEED Operation

```
1: function RESEED($SD_{rsd}$)                          ▷ Removed $\mathcal{S}'$ and $\alpha$
2:     $\mathcal{S} \leftarrow$ xof(ENCODE($SD_{rsd}, n = 1$), $|\mathcal{S}|$)
3:     return $\mathcal{S}$
4: end function
```

We know from subsection 3.2.2 that the source of additional entropy during XDRBG RESEED is the SD_{rsd} with a minimum entropy of at least H_{rsd} bits. We have not considered performing the unsound idealization for the XDRBG INSTANTIATE because it only takes one input parameter, i.e., SD_{init} which is expected to be the source of (providing a minimum) entropy of at least H_{init} bits, and we have an optional α parameter as well. Secondly, we have also not provided the unsound idealization for XDRBG GENERATE because it takes the current XDRBG state $\mathcal{S}'$ and an optional α parameter, but neither of them act as a source of entropy. But, we indeed provide the sound idealization for XDRBG INSTANTIATE and GENERATE. To detect any entropy loss, we have checked whether the RESEED(SD_{rsd}) call is injective or not:

$$\mathcal{S}_1 \leftarrow \text{RESEED}(SD_{rsd1}) \neq \mathcal{S}_2 \leftarrow \text{RESEED}(SD_{rsd2}); SD_{rsd1} \neq SD_{rsd2}$$
$$\mathcal{S}_1 \leftarrow \text{RESEED}(SD_{rsd1}) = \mathcal{S}_2 \leftarrow \text{RESEED}(SD_{rsd2}); SD_{rsd1} = SD_{rsd2}$$

The Unsound Idealization for PRG: The modified PRG REFRESH operation is defined in Algorithm 5 to align with the unsound idealization method, with the cryptographic part being the AES block cipher in counter mode as the generator function G. We have not explicitly provided the unsound idealization for the PRG NEXT call, because, the (cryptographic) generator function G in the PRG NEXT call takes only

Algorithm 5 Modified PRG REFRESH Operation

```
1: function REFRESH($x$)                                    ▷ Removed $\mathcal{S}'$
2:     $X \leftarrow$ EXTRACT($x$)
3:     $\mathcal{S} \leftarrow$ G($X$)
4:     return $\mathcal{S}$          ▷ Only the first $\lambda$ bits out of the generated $2\lambda$ bits
5: end function
```

one input parameter, i.e., the current state of the PRG $\mathcal{S}'$ but it does not act as an additional source of entropy. But, we indeed provide the sound idealization for PRG NEXT. To detect any entropy loss, we have checked whether the REFRESH(x) call is injective or not:

$$\mathcal{S}_1 \leftarrow \mathrm{G}(X_1) \neq \mathcal{S}_2 \leftarrow \mathrm{G}(X_2); X_1 \neq X_2$$
$$\mathcal{S}_1 \leftarrow \mathrm{G}(X_1) = \mathcal{S}_2 \leftarrow \mathrm{G}(X_2); X_1 = X_2$$

No Unsound Idealization for HKDF: With respect to Algorithm 1 for the HKDF, HMAC is the cryptographic part. We have not provided the unsound idealization for the HKDF extract (XTR) because it takes only one input parameter *SKM* which acts as the source of entropy and the salt *XTS* is the optional parameter. Similarly, for the HKDF expand (PRF) call, it takes the pseudorandom key *PRK* and an optional *info* parameter, but neither of them act as a source of entropy. But, we indeed provide the sound idealization for both the HKDF extract and expand call.

The Sound Idealizations

Here, we have considered the operations of HKDF, XDRBG, and PRG from Algorithm 1, Algorithm 2, and Algorithm 3 respectively exactly as they are to conduct the sound idealization.

The Sound Idealization for HKDF: Excluding the optional salt *XTS* and the info parameter *info* altogether, to detect the entropy loss, the following injectivity axioms must hold:

$$\text{For XTR}:$$
$$PRK_1 \leftarrow \mathrm{XTR}(SKM_1) \neq PRK_2 \leftarrow \mathrm{XTR}(SKM_2); SKM_1 \neq SKM_2$$
$$PRK_1 \leftarrow \mathrm{XTR}(SKM_1) = PRK_2 \leftarrow \mathrm{XTR}(SKM_2); SKM_1 = SKM_2$$
$$\text{For same } l \; in \text{ PRF}:$$
$$KM_1 \leftarrow \mathrm{PRF}(PRK_1, l) \neq KM_2 \leftarrow \mathrm{PRF}(PRK_2, l); PRK_1 \neq PRK_2$$
$$KM_1 \leftarrow \mathrm{PRF}(PRK_1, l) = KM_2 \leftarrow \mathrm{PRF}(PRK_2, l); PRK_1 = PRK_2$$

The Sound Idealization for XDRBG: Excluding the optional α parameter altogether, to detect the entropy loss, the following injectivity axioms must hold:

$$\text{For INSTANTIATE :}$$
$$\mathcal{S}_1 \leftarrow \text{INSTANTIATE}(SD_{init1}) \neq \mathcal{S}_2 \leftarrow \text{INSTANTIATE}(SD_{init2}); SD_{init1} \neq SD_{init2}$$
$$\mathcal{S}_1 \leftarrow \text{INSTANTIATE}(SD_{init1}) = \mathcal{S}_2 \leftarrow \text{INSTANTIATE}(SD_{init2}); SD_{init1} = SD_{init2}$$
$$\text{For same } \mathcal{S}' \textit{ in } \text{RESEED :}$$
$$\mathcal{S}_1 \leftarrow \text{RESEED}(\mathcal{S}', SD_{rsd1}) \neq \mathcal{S}_2 \leftarrow \text{RESEED}(\mathcal{S}', SD_{rsd2}); SD_{rsd1} \neq SD_{rsd2}$$
$$\mathcal{S}_1 \leftarrow \text{RESEED}(\mathcal{S}', SD_{rsd1}) = \mathcal{S}_2 \leftarrow \text{RESEED}(\mathcal{S}', SD_{rsd2}); SD_{rsd1} = SD_{rsd2}$$
$$\text{For same } l \textit{ in } \text{GENERATE :}$$
$$(\mathcal{S}'_{op_1}, \mathcal{K}_1) \leftarrow \text{GENERATE}(\mathcal{S}'_{inp_1}, l) \neq (\mathcal{S}'_{op_2}, \mathcal{K}_2) \leftarrow \text{GENERATE}(\mathcal{S}'_{inp_2}, l); \mathcal{S}'_{inp_1} \neq \mathcal{S}'_{inp_2}$$
$$(\mathcal{S}'_{op_1}, \mathcal{K}_1) \leftarrow \text{GENERATE}(\mathcal{S}'_{inp_1}, l) = (\mathcal{S}'_{op_2}, \mathcal{K}_2) \leftarrow \text{GENERATE}(\mathcal{S}'_{inp_2}, l); \mathcal{S}'_{inp_1} = \mathcal{S}'_{inp_2}$$

The Sound Idealization for PRG: To detect the entropy loss, the following injectivity axioms must hold:

$$\text{For same } \mathcal{S}' \textit{ in } \text{REFRESH :}$$
$$\mathcal{S}_1 \leftarrow \text{G}(\mathcal{S}' \oplus X_1) \neq \mathcal{S}_2 \leftarrow \text{G}(\mathcal{S}' \oplus X_2); X_1 \neq X_2$$
$$\mathcal{S}_1 \leftarrow \text{G}(\mathcal{S}' \oplus X_1) = \mathcal{S}_2 \leftarrow \text{G}(\mathcal{S}' \oplus X_2); X_1 = X_2$$
$$\text{For NEXT :}$$
$$(\mathcal{S}'_{op_1}, \mathcal{K}_1) \leftarrow \text{NEXT}(\mathcal{S}'_{inp_1}) \neq (\mathcal{S}'_{op_2}, \mathcal{K}_2) \leftarrow \text{NEXT}(\mathcal{S}'_{inp_2}); \mathcal{S}'_{inp_1} \neq \mathcal{S}'_{inp_2}$$
$$(\mathcal{S}'_{op_1}, \mathcal{K}_1) \leftarrow \text{NEXT}(\mathcal{S}'_{inp_1}) = (\mathcal{S}'_{op_2}, \mathcal{K}_2) \leftarrow \text{NEXT}(\mathcal{S}'_{inp_2}); \mathcal{S}'_{inp_1} = \mathcal{S}'_{inp_2}$$

5.3.4 Idealization Results

In this subsection, we describe in detail that how we have checked the injectivity between the different inputs and corresponding outputs based on the sound and unsound idealizations applied to the cryptographic primitives from subsection 5.3.3. In order to depict this, we would like to provide a generalization as mentioned in Algorithm 6. This generalization fundamentally states that, we generate n number of inputs referred as *INPs* using GENERATE_INPUTS. Then, using those inputs *INPs* as the input parameter to any operation of a cryptographic primitive, namely the HKDF, XDRBG and PRG, we generate the respective outputs *OPs*. To check the injectivity, we use the CHECK_DUPLICATES method where we define an empty map named *input_to_output_map* at first, which will store key-value pairs of output op_i as the key and input inp_i as the value. Then, we take those inputs *INPs* and their corresponding outputs *OPs* and simultaneously iterate through them and basically check whether an output op_i is already present as a key in the *input_to_output_map* or not. If this op_i is present as a key in the *input_to_output_map*, then an error message is shown which depicts that, we have indeed encountered a scenario where two inputs have lead to the same output op_i, i.e., in any of the previous iterations, an input $ip_j, 0 \leq j < i$ has already been mapped to this output op_i. We wish to explicitly clarify that, we have used the output op_i as the key in the map and not the input inp_i because, our intention is to check whether any two inputs lead to the same output or not as opposed to just storing the input output pairs in the *input_to_output_map*.

The Unsound Idealization for XDRBG: With respect to Algorithm 4 and Algorithm 6, we have checked the injectivity between the seeds and the reseeded XDRBG states for

Algorithm 6 The Injectivity Checking Generalization

```
1: function GENERATE_INPUTS($n$)        ▷ $n$ is the number of the inputs to be generated
2:     $INPs \leftarrow []$
3:     for $i$ in range ($n$) do
4:         $inp_i \leftarrow generate\_random\_input\_parameters()$
5:         $INPs$.append($inp_i$)
6:     end for
7:     return $INPs$                       ▷ A list of $n$ inputs
8: end function
9: function CHECK_DUPLICATES($INPs$, $OPs$)
10:    input_to_output_map $\leftarrow \{\}$
11:    for $inp_i, op_i$ in zip($INPs$, $OPs$) do      ▷ $i \in \{0, 1, \dots, n-1\}$
12:        if $op_i$ in input_to_output_map then
13:            return An error message and abort checking further
14:        end if
15:        input_to_output_map[$op_i$] = $inp_i$
16:    end for
17:    return Nothing                ▷ Indeed ensures that there are no duplicates
18: end function
```

SHAKE-128, SHAKE-256 and ASCON-XOF which is mentioned in Algorithm 7 as the unsound idealization of XDRBG.

Algorithm 7 Unsound Idealization of XDRBG

```
1: function TEST_ROUTINE_FOR_XDRBG()
2:     $INPs \leftarrow$ GENERATE_INPUTS($n = 2^{21}$)     ▷ $INPs = SD_{rsd_i \in \{0,1,\dots,2^{21}-1\}}$
3:     $OPs \leftarrow []$
4:     for $inp_i$ in $INPs$ do
5:         $\mathcal{S}_i \leftarrow$ RESEED($inp_i$)
6:         $OPs$.append($\mathcal{S}_i$)             ▷ $OPs = \mathcal{S}_{i \in \{0,1,\dots,2^{21}-1\}}$
7:     end for
8:     CHECK_DUPLICATES($INPs$, $OPs$)
9: end function
```

The Unsound Idealization for PRG: With respect to Algorithm 5 and Algorithm 6, we have checked the injectivity between the seeds and the refreshed PRG states while considering the security parameter $\lambda \in \{16, 24, 32\}$, similar to what we did for the XDRBG in Algorithm 7. We have generated 2^{21} seeds for refreshing, i.e, X_i where $i \in \{0, 1, \dots, 2^{21} - 1\}$ using $INPs \leftarrow$ GENERATE_INPUTS($n = 2^{21}$). With these seeds as the (only) input parameter to the generator function G in the PRG REFRESH call, we generated 2^{21} refreshed PRG states $\mathcal{S}_i$ which are basically the random outputs from G, i.e, the AES block cipher in counter mode.

The Sound Idealization for HKDF: For the sound idealization of HKDF (mentioned in subsubsection 5.3.3) and Algorithm 6, we have checked the injectivity of the different inputs and corresponding outputs as mentioned in Algorithm 8.

Algorithm 8 Sound Idealization of HKDF

```
function TEST_ROUTINE_FOR_HKDF()
    $INPs \leftarrow$ GENERATE_INPUTS($n = 2^{21}$)        ▷ $INPs = SKM_{i\in\{0,1,\ldots,2^{21}-1\}}$
    $OPs \leftarrow []$
    for $inp_i$ in $INPs$ do
        $PRK_i \leftarrow$ XTR($inp_i$)
        $OPs$.append($PRK_i$)        ▷ $OPs = PRK_{i\in\{0,1,\ldots,2^{21}-1\}}$
    end for
    CHECK_DUPLICATES($INPs$, $OPs$)
    $OPs \leftarrow []$
    $l = 32$        ▷ Fixed length $l = 32$ bytes
    for $inp_i$ in $INPs$ do        ▷ Here $INPs = PRK_{i\in\{0,1,\ldots,2^{21}-1\}}$
        $KM_i \leftarrow$ PRF($PRK_i, l$)
        $OPs$.append($KM_i$)        ▷ $OPs = KM_{i\in\{0,1,\ldots,2^{21}-1\}}$
    end for
    CHECK_DUPLICATES($INPs$, $OPs$)  ▷ $INPs = PRK_{i\in\{0,1,\ldots,2^{21}-1\}}$, $OPs = KM_{i\in\{0,1,\ldots,2^{21}-1\}}$
end function
```

The Sound Idealization for XDRBG: For the sound idealization of XDRBG (from subsubsection 5.3.3) and Algorithm 6, we have checked the injectivity of the different inputs and corresponding outputs as mentioned in Algorithm 9.

Algorithm 9 Sound Idealization of XDRBG

```
function TEST_ROUTINE_FOR_XDRBG()
    $INPs \leftarrow$ GENERATE_INPUTS($n = 2^{21}$)        ▷ $INPs = SD_{init_i \in\{0,1,\ldots,2^{21}-1\}}$
    $OPs \leftarrow []$
    for $inp_i$ in $INPs$ do
        $\mathcal{S}_i \leftarrow$ INSTANTIATE($inp_i$)
        $OPs$.append($\mathcal{S}_i$)        ▷ $OPs = \mathcal{S}_{i\in\{0,1,\ldots,2^{21}-1\}}$
    end for
    CHECK_DUPLICATES($INPs$, $OPs$)
    $OPs \leftarrow []$
    $\mathcal{S}' \leftarrow$ INSTANTIATE($SD_{init}$)        ▷ Fixed current XDRBG state $\mathcal{S}'$
    for $inp_i$ in $INPs$ do        ▷ Here $INPs = SD_{rsd_i \in\{0,1,\ldots,2^{21}-1\}}$
        $\mathcal{S}_i \leftarrow$ RESEED($\mathcal{S}', inp_i$)
        $OPs$.append($\mathcal{S}_i$)        ▷ $OPs = \mathcal{S}_{i\in\{0,1,\ldots,2^{21}-1\}}$
    end for
    CHECK_DUPLICATES($INPs$, $OPs$)    ▷ $INPs = SD_{rsd_i \in\{0,1,\ldots,2^{21}-1\}}$, $OPs = \mathcal{S}_{i\in\{0,1,\ldots,2^{21}-1\}}$
    $INPs \leftarrow []$, $OPs \leftarrow []$
    $l = 32$        ▷ Fixed length $l = 32$ bytes
    $\mathcal{S}'_{inp_0} \leftarrow$ INSTANTIATE($SD_{init}$)
    for i in range ($n$) do        ▷ $n = 2^{21}$
        $INPs$.append($\mathcal{S}'_{inp_i}$)        ▷ $INPs = \mathcal{S}_{inp_{i\in\{0,1,\ldots,2^{21}-1\}}}$
        $(\mathcal{S}'_{op_i}, \mathcal{K}_i) \leftarrow$ GENERATE($\mathcal{S}'_{inp_i}, l$)
        $\mathcal{S}'_{inp_{i+1}} = \mathcal{S}'_{op_i}$
        $OPs$.append($\mathcal{K}_i$)        ▷ $OPs = \mathcal{K}_{i\in\{0,1,\ldots,2^{21}-1\}}$
    end for
    CHECK_DUPLICATES($INPs$, $OPs$)
end function
```

The Sound Idealization for PRG: For the sound idealization of PRG (from subsubsection 5.3.3) and Algorithm 6, we have checked the injectivity of the different inputs and corresponding outputs as mentioned in Algorithm 10.

Algorithm 10 Sound Idealization of PRG

```
1: function TEST_ROUTINE_FOR_PRG()
2:     $INPs \leftarrow$ GENERATE_INPUTS($n = 2^{21}$)          ▷ $INPs = X_{i \in \{0,1,\ldots,2^{21}-1\}}$
3:     $OPs \leftarrow []$
4:     $\mathcal{S}' \leftarrow \{0\}^\lambda$          ▷ Fixed current PRG state $\mathcal{S}'$
5:     for $inp_i$ in $INPs$ do
6:         $\mathcal{S}_i \leftarrow \mathrm{G}(\mathcal{S}' \oplus inp_i)$          ▷ For the PRG REFRESH call
7:         $OPs$.append($\mathcal{S}_i$)          ▷ $OPs = \mathcal{S}_{i \in \{0,1,\ldots,2^{21}-1\}}$
8:     end for
9:     CHECK_DUPLICATES($INPs$, $OPs$)
10:    $INPs \leftarrow []$, $OPs \leftarrow []$
11:    $\mathcal{S}'_{inp_0} \leftarrow \{0\}^\lambda$
12:    for i in range ($n$) do          ▷ $n = 2^{21}$
13:        $INPs$.append($\mathcal{S}'_{inp_i}$)          ▷ $INPs = \mathcal{S}_{inp_{i \in \{0,1,\ldots,2^{21}-1\}}}$
14:        $(\mathcal{S}'_{op_i}, \mathcal{K}_i) \leftarrow$ NEXT($\mathcal{S}'_{inp_i}$)
15:        $\mathcal{S}'_{inp_{i+1}} = \mathcal{S}'_{op_i}$
16:        $OPs$.append($\mathcal{K}_i$)          ▷ $OPs = \mathcal{K}_{i \in \{0,1,\ldots,2^{21}-1\}}$
17:    end for
18:    CHECK_DUPLICATES($INPs$, $OPs$)
19: end function
```

Final Outcome: Upon testing the different injectivity axioms for the different operations of the cryptographic primitives with 2^{21} input parameters, we have not (yet) encountered a scenario where two different inputs lead to the same (pseudo) random output. This indicates that indeed there is no entropy loss when using these cryptographic primitives, but we still ask a reader to be cautious when considering our idealizations.

Chapter 6

Conclusion

In this work we have explored and provided the design of cryptographic key chains based on three cryptographic primitives, namely, HKDF, XDRBG, and PRG. The key chains have been implemented and evaluated for performance both with and without persistently storing the state $\mathcal{S}_i$ of the key chain.

Now, we can assure that there are indeed other ways of generating key chains as opposed to only HKDF-based key chains like Double Ratchet algorithm [5]. This also proves that the key chain generation can be tailored based on specific requirements and availability of cryptographic primitive. We observed that, when we did not persistently store the state $\mathcal{S}_i$ of the key chain in the database, for the different (classical) security levels, i.e., 16 bytes, 32 bytes and 64 bytes, we have the XDRBG-based key chain with SHAKE-128 XOF, the HKDF-based key chain with SHA3-256 and the HKDF-based key chain with SHA-512 as the most efficient cryptographic primitives. But, for these three (classical) security levels, the PRG-based key chain turned out to be less efficient compared to the other cryptographic primitives. And in an overall sense, if a user wishes to aim for better performance, then one must ideally proceed with the SHAKE-128 XOF or PRG with security parameter $\lambda = 16$ which provides 16 bytes of security as they have somewhat comparable performance. If the performance is not a critical factor, then a user can choose to aim for a standard (but moderate) security of 32 bytes and proceed with the HKDF key chain based on SHA3-256. And in case one demands utmost security, then they can definitely choose to proceed with the HKDF key chain based on SHA-512.

When we persistently stored the state $\mathcal{S}_i$ of the key chain in the database, we observed that it leads to a considerable increase in the execution time of the key chain generation while simultaneously providing an insight as to how the key chain is actually going to perform in the real world. This also indicates that, for applications where performance is critical, minimizing or choosing an optimal frequency of database writes is imperative. Usage of HKDF as a cryptographic primitive is indeed suitable for deriving cryptographic keys when the input parameters are drawn from an imperfect source, but the introduction of XDRBG and PRG-based key chain constructions also diversifies the choice of cryptographic primitives for key chain generation.

Future Work: As part of future work, we believe it might be worth exploring new cryptographic primitives like the Multi-Factor Key Derivation Function (MFKDF) and the Stream Cipher-Based Key Derivation Function (SCKDF) respectively mentioned in item 2 and item 3 of Appendix A for generating the key chains.

Appendix A

Other Variety of KDFs

We mentioned in section 1.1 that there are other KDFs as well in addition to HKDF we describe about some types of such KDFs below:

1. Password-Based Key Derivation Function (PBKDF):
    a) As a part of the Public Key Cryptographic Standards (PKCS) of RSA laboratories, they proposed PBKDF [36] (specifically PBKDF v2.0). PBKDF2 employs usage of PRFs such as HMAC (subsection 2.2.1) where the secret input key K is a password or passphrase chosen by the user. According to [37], 1,300,000 iterations, 600,000 iterations, and 210,000 iterations are recommended if the hashing algorithm H in the HMAC is SHA1, SHA256 and SHA512 respectively. This procedure is also referred as key-stretching, and it acts as a preventive measure against password cracking.
    b) The author of [38] realized that indeed modifying the number of iterations in PBKDFs impact the computation time (from an adversary's point of view) when launching an attack, but with the development of semiconductor technologies and advancements in parallelized hardware implementations, the cost of a brute force attack has significantly dropped over time. So, *Scrypt* was proposed by [38] which is a PBKDF but it is a sequential memory-hard function. Such sequential memory-hard functions are aimed towards reducing an adversary's advantage when using custom-designed parallel circuit such as Application Specific Integrated Circuit (ASIC) or Graphical Processing unit (GPU). This is because, a sequential memory-hard function typically means that the function will require a colossal amount of memory to compute while simultaneously ensuring that it can only be correctly executed if done in a sequential manner instead of trying to quickly execute it by parallelizing the computation.
    c) Biryukov et al. (2016) [39] realized that attackers can still circumvent the security of Scrypt [38] because of its trivial time-memory trade-off, i.e., an adversary can use a less resource-intensive hardware implementation at the cost of taking more time to crack a password or the adversary will require an implementation with excessive memory requirements (which is expensive to parallelize) but will also be able to crack the password quickly. Additionally, the authors also highlight the complexity of Scrypt [38] and proposed the *Argon2* family of PBKDFs which comprises of three individual PBKDF namely, *Argon2i, Argon2d,* and *Argon2id* [39]. Argon2i is considered to be resistant to side-channel attacks, Argon2d is considered to be resistant to time-memory trade-off attacks, and Argon2id is the best of both worlds, and it has also been standardised by Internet Engineering Task Force (IETF) [40]. As an additional note, [41] explicitly mentions that in absence of Argon2id, Scrypt [38] is considered as the next best alternative as a PBKDF.

2. Multi-Factor Key Derivation Function (MFKDF): Nair and Song (2023) [42] realized an open issue with respect to usage of simple PBKDFs (as mentioned in the above item 1), i.e., lacking incorporation of secondary authentication factors in addition to a password with the key derivation process. This leaves the users vulnerable to password spraying and credential stuffing attacks as it creates a single point of failure, i.e., the password itself which can be easily guessed or broken by brute force. As a solution, they proposed MFKDF which not only provides security against the mentioned vulnerabilities but also ensures account recovery without having to rely on a trusted device (with no significant change to the user experience during the account recovery). As a part of the additional authentication factors of MFKDF, in addition to the user's password, they consider the usage of software tokens such as HMAC-based One-Time Password (HOTP), Time-based One-Time Password (TOTP), hardware tokens such as YubiKeys, and other out-of-band authentication factors like email.

3. Stream Cipher-Based Key Derivation Function (SCKDF): Wen et al. (2014) [43] observed that there are KDFs based on hash functions like HKDF and block ciphers like "KDF in Counter Mode" [44] which is a block cipher based KDF using Cipher-based Message Authentication Code (CMAC) as PRF with an underlying block cipher like an AES variant (from subsection 2.3.1). But, according to [43] these hash function based or block cipher based KDFs are not a suitable candidate for resource-constrained environments such as smartphones. As a solution, they proposed KDFs based on stream-ciphers (namely Trivium, Sosemanuk, and Rabbit) employing the usage of the extract-then-expand approach. The authors have also highlighted that the correct choice of the stream cipher is critical when considering the practical security and the performance of the SCKDF.

Bibliography

[1] P. W. Shor, “Algorithms for quantum computation: Discrete logarithms and factoring,” in *Proceedings 35th annual symposium on foundations of computer science*, Ieee, 1994, pp. 124–134.

[2] P. W. Shor, “Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer,” *SIAM Journal on Computing*, vol. 26, no. 5, pp. 1484–1509, 1997.

[3] Tommy Charles, Thalia Laing, *Anticipating the quantum threat to cryptography.* [Online]. Available: `https://threatresearch.ext.hp.com/anticipating-the-quantum-threat-to-cryptography/` (visited on 08/28/2024).

[4] L. K. Grover, “A fast quantum mechanical algorithm for database search,” in *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, 1996, pp. 212–219.

[5] Trevor Perrin, Moxie Marlinspike, *The double ratchet algorithm, signal messenger.* [Online]. Available: `https://kr-labs.com.ua/books/doubleratchet.pdf` (visited on 08/28/2024).

[6] H. Krawczyk, “Cryptographic extraction and key derivation: The hkdf scheme,” in *Annual Cryptology Conference*, Springer, 2010, pp. 631–648.

[7] J. Kelsey, S. Lucks, and S. Müller, “Xdrbg: A proposed deterministic random bit generator based on any xof,” *IACR Transactions on Symmetric Cryptology*, vol. 2024, no. 1, pp. 5–34, 2024.

[8] B. Barak and S. Halevi, “A model and architecture for pseudo-random generation with applications to/dev/random,” in *Proceedings of the 12th ACM conference on Computer and communications security*, 2005, pp. 203–212.

[9] M. J. Dworkin, “Sha-3 standard: Permutation-based hash and extendable-output functions,” 2015.

[10] H. Lipmaa, P. Rogaway, and D. Wagner, “Ctr-mode encryption,” in *First NIST Workshop on Modes of Operation*, Citeseer. MD, vol. 39, 2000.

[11] C. Foreman, R. Yeung, A. Edgington, and F. J. Curchod, “Cryptomite: A versatile and user-friendly library of randomness extractors,” *arXiv preprint arXiv:2402.09481*, 2024.

[12] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, “Sponge functions,” in *ECRYPT hash workshop*, vol. 2007, 2007.

[13] F. I. P. S. Publication 180-4, *Secure hash standard (shs).* [Online]. Available: `https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf` (visited on 08/28/2024).

[14] F. I. P. S. Publication 202, *Sha-3 standard: Permutation-based hash and extendable-output functions.* [Online]. Available: `https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf` (visited on 08/28/2024).

[15] T. Lange, *Sponge functions.* [Online]. Available: `https://hyperelliptic.org/tanja/teaching/crypto21/hash-5.pdf` (visited on 08/28/2024).

[16] C. Dobraunig, M. Eichlseder, F. Mendel, and M. Schläffer, "Ascon v1.2: Lightweight authenticated encryption and hashing," *J. Cryptol.*, vol. 34, no. 3, p. 33, 2021. DOI: 10.1007/S00145-021-09398-9. [Online]. Available: https://doi.org/10.1007/s00145-021-09398-9.

[17] Wikipedia Contributors, *Sponge function.* [Online]. Available: https://en.wikipedia.org/wiki/Sponge_function (visited on 08/28/2024).

[18] D. H. Krawczyk, M. Bellare, and R. Canetti, *HMAC: Keyed-Hashing for Message Authentication*, RFC 2104, Feb. 1997. DOI: 10.17487/RFC2104. [Online]. Available: https://www.rfc-editor.org/info/rfc2104.

[19] F. I. P. S. Publication 198-1, *The keyed-hash message authentication code(hmac).* [Online]. Available: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.198-1.pdf (visited on 08/28/2024).

[20] Hana Rhim, Milos Simic, *What are key derivation functions?* [Online]. Available: https://www.baeldung.com/cs/kdf-cryptography (visited on 08/28/2024).

[21] V. Rijmen and J. Daemen, "Advanced encryption standard," *Proceedings of federal information processing standards publications, national institute of standards and technology*, vol. 19, p. 22, 2001.

[22] F. I. P. S. Publication 197, *Advanced encryption standard (aes).* [Online]. Available: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197-upd1.pdf (visited on 08/28/2024).

[23] M. Dworkin, "Recommendation for block cipher modes of operation: Methods and techniques," *NIST Special Publication*, vol. 800, 38A, 2001.

[24] S. Wang, *The difference in five modes in the aes encryption algorithm.* [Online]. Available: https://www.highgo.ca/2019/08/08/the-difference-in-five-modes-in-the-aes-encryption-algorithm/ (visited on 08/28/2024).

[25] Wikipedia Contributors, *Probability mass function.* [Online]. Available: https://en.wikipedia.org/wiki/Probability_mass_function (visited on 08/28/2024).

[26] Statistics Review Website, *Probability: Joint, marginal and conditional probabilities.* [Online]. Available: https://sites.nicholas.duke.edu/statsreview/jmc/ (visited on 08/28/2024).

[27] E. Barker and J. Kelsey, "Recommendation for random number generation using deterministic random bit generators," *NIST Special Publication*, vol. 800, 90A, 2015.

[28] S. Ashwathnarayana, *Understanding entropy: The key to secure cryptography and randomness.* [Online]. Available: https://www.netdata.cloud/blog/understanding-entropy-the-key-to-secure-cryptography-and-randomness/#what-is-entropy (visited on 08/28/2024).

[29] Ben Lynn, *Pseudo-random number generators.* [Online]. Available: https://crypto.stanford.edu/pbc/notes/crypto/prng.html (visited on 08/28/2024).

[30] C. Casebeer, *Python-hkdf: Hmac-based extract-and-expand key derivation function (hkdf) implemented in python.* [Online]. Available: https://github.com/casebeer/python-hkdf (visited on 08/28/2024).

[31] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, "On the indifferentiability of the sponge construction," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, 2008, pp. 181–197.

[32] J. Czajkowski, "Quantum indifferentiability of sha-3," *Cryptology ePrint Archive*, 2021.

[33] M. Eichlseder, *Pyascon.* [Online]. Available: `https://github.com/meichlseder/pyascon` (visited on 08/28/2024).

[34] Quantinuum Ltd., *Cqcl/cryptomite: Python library of efficient and numerically-precise randomness extractors.* [Online]. Available: `https://github.com/CQCL/cryptomite` (visited on 08/28/2024).

[35] F. Dörre and V. Klebanov, "Practical detection of entropy loss in pseudo-random number generators," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 678–689.

[36] B. Kaliski, *PKCS #5: Password-Based Cryptography Specification Version 2.0*, RFC 2898, Sep. 2000. DOI: `10.17487/RFC2898`. [Online]. Available: `https://www.rfc-editor.org/info/rfc2898`.

[37] Open Web Application Security Project, *Pbkdf2: Password storage - owasp cheat sheet series.* [Online]. Available: `https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html#pbkdf2` (visited on 08/28/2024).

[38] C. Percival, *Stronger key derivation via sequential memory-hard functions.*

[39] A. Biryukov, D. Dinu, and D. Khovratovich, "Argon2: New generation of memory-hard functions for password hashing and other applications," in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, IEEE, 2016, pp. 292–302.

[40] A. Biryukov, D. Dinu, D. Khovratovich, and S. Josefsson, *Argon2 Memory-Hard Function for Password Hashing and Proof-of-Work Applications*, RFC 9106, Sep. 2021. DOI: `10.17487/RFC9106`. [Online]. Available: `https://www.rfc-editor.org/info/rfc9106`.

[41] Open Web Application Security Project, *Scrypt: Password storage - owasp cheat sheet series.* [Online]. Available: `https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html#scrypt` (visited on 08/28/2024).

[42] V. Nair and D. Song, "Multi-factor key derivation function (mfkdf) for fast, flexible, secure, & practical key management," in *Proceedings of the 32nd USENIX Conference on Security Symposium*, 2023, pp. 2097–2114.

[43] C. W. Chuah, E. Dawson, and L. Simpson, "Key derivation function: The sckdf scheme," in *Security and Privacy Protection in Information Processing Systems: 28th IFIP TC 11 International Conference, SEC 2013, Auckland, New Zealand, July 8-10, 2013. Proceedings 28*, Springer, 2013, pp. 125–138.

[44] NIST Publication SP 800-108 Revision 1 | CSRC, *Recommendation for key derivation using pseudorandom functions.* [Online]. Available: `https://csrc.nist.gov/News/2022/nist-publishes-sp-800-108-revision-1` (visited on 08/28/2024).

List of Abbreviations

AES	Advanced Encryption Standard
ASIC	Application Specific Integrated Circuit
CMAC	Cipher-based Message Authentication Code
DRBG	Deterministic Random Bit Generator
GPU	Graphical Processing unit
HKDF	HMAC-Based Key Derivation Function
HMAC	Hash-based Message Authentication Code
HOTP	HMAC-based One-Time Password
IETF	Internet Engineering Task Force
KDF	Key Derivation Function
KDFs	Key Derivation Functions
MFKDF	Multi-Factor Key Derivation Function
NIST	National Institute of Standards and Technology
PBKDF	Password-Based Key Derivation Function
PKCS	Public Key Cryptographic Standards
PRF	Pseudorandom Function
PRG	Pseudorandom Generator
PRNG	Pseudorandom Number Generator
SCKDF	Stream Cipher-Based Key Derivation Function
TOTP	Time-based One-Time Password
XDRBG	XOF-Based Deterministic Random Bit Generator
XOF	Extendable Output Function
XOFs	Extendable Output Functions