SHAKE Cost-Based Key Derivation– SCB 1.1

Revision 1.1a, November 24, 2024

John G. Underhill – john.underhill@protonmail.com

SCB is a cost-based key derivation function, one that can increase the memory usage and computational complexity of the underlying hash function. Suitable for password hashing, key generation, and in cases where brute-force attacks on a derived key must be strongly mitigated.

1. Introduction

In the realm of cryptographic systems, Key Derivation Functions (KDFs) are essential for transforming entropy sources, such as passwords or cryptographic keys, into secure keys suitable for various cryptographic operations. The SHAKE Cost-Based (SCB) leverages the SHAKE extendable-output functions (XOFs) from the SHA-3 family to enhance security through memory-hard and computationally intensive processes. This paper provides a comprehensive analysis of the SCB implementation, detailing its protocol, mathematical underpinnings, security mechanisms, internal functionalities, and overall efficacy in resisting contemporary and emerging attack vectors.

2. Protocol Description

The SCB KDF is designed to generate pseudo-random bytes from a given seed and an optional personalization string, employing the SHA3 SHAKE-256 or SHAKE-512 extended output functions, based on the input seed length (64 bytes uses SHAKE-512, a 32 byte seed selects SHAKE-256). The protocol incorporates adjustable **memory cost** (`memcost`) and **CPU cost** (`cpucost`) parameters to adjust computational and memory hardness, thereby deterring adversaries from executing efficient brute-force or parallelized attacks.

2.1 Operational Workflow

2.1.1 Initialization (`qsc_scb_initialize`):

Seed Absorption: Depending on the seed length (32 bytes for SHAKE-256 or 64 bytes for SHAKE-512), the seed is absorbed into the SHAKE state along with an optional personalization string.

Cost Parameter Configuration: The `cpucost` and `memc` parameters are set, dictating the number of iterations and the memory usage during key derivation.

Key Generation: A derived key (`ckey`) is generated from the SHAKE state, serving as the foundational material for subsequent operations.

2.1.2 Key Generation (`qsc_scb_generate`):

Memory Buffer Allocation: Allocates a buffer proportional to the `memc` parameter, which determines the memory hardness.

Iterative Key Derivation:

For each iteration defined by `cpucost`:

- **Hash Update:** The current key is reabsorbed into the SHAKE state.

- **Memory Filling:** The buffer is populated using the `scb_fill_memory` function, which enforces a scattering pattern to induce cache misses.
- **Key Finalization:** The SHAKE state is finalized to produce an updated key.

Output Extraction: Generates the desired pseudo-random output bytes from the finalized SHAKE state.

2.1.3 State Updating (`qsc_scb_update`):

Key Material Integration: Absorbs additional seed material into the SHAKE state, allowing dynamic updates to the key material without reinitialization.

2.1.4 Disposal (`qsc_scb_dispose`):

State Clearing: Securely erases sensitive key material and resets state parameters to prevent residual data leakage.

The SCB protocol combines cryptographic hashing with memory-hard operations, ensuring that key derivation remains both secure and resistant to various attack methodologies.

3. Mathematical Description

The SCB KDF's robustness is underpinned by its mathematical framework, which enforces memory and computational hardness through deliberate memory access patterns and iterative hashing processes. This section details the mathematical constructs that ensure SCB's resistance to attacks such as brute-force, dictionary, and rainbow table attacks.

3.1 Memory Hardness via Scattering Pattern

At the core of SCB's security is the **scattering pattern**, which dictates memory access sequences to maximize cache misses and induce cache thrashing. The scattering pattern ensures that each memory access is spaced sufficiently apart to exceed the capacity of the CPU's cache hierarchy, targeting the L1 and L2 caches.

Definitions

- **Cache Line Size (`CL`):** 64 bytes.
- **L2 Cache Size (`L2`):** (default) 256 KiB (262,144 bytes).
- **Memory Access Spacing (`S`):** (default) 256 KiB.

Cache Line and Access Calculations

Each cache line occupies 64 bytes. The number of cache lines that fit within the L2 cache is:

$$\text{Cache Lines in } L2 = \frac{L2}{CL} = \frac{262{,}144}{64} = 4{,}096 \text{ cache lines}$$

To induce cache misses, SCB ensures that each memory access is spaced 256 KiB apart, corresponding to 4,096 cache lines. This spacing guarantees that consecutive accesses target distinct cache lines beyond the typical L2 cache's capacity, enforcing a high cache miss rate.

The L2 cache size is configurable by modifying the QSC_SCB_L2CACHE_DEFAULT_SIZE constant. The default is L2 cache size is 256 KiB (1024 x 1024 x 256), but this size can be changed if targeting a larger cache size. The size of the L2 cache paremeter must be a doubling from 128 KiB; 128KiB, 256 KiB, 512 KiB, 1024 KiB (1 MiB), 2048 KiB (2 MiB). If the cache size is changed, the memory cost must increased so that the memory cost is at least 4 times the size of the L2 cache size; ex. at 256 Kib L2 cache target, the memory cost should be at least 1 MiB the minimum setting of 1, at 512 KiB a memory cost minimum of 2 MiB or 2, at 1024 MiB a memory cost of 4.

3.2 Lane Multiplier and Cache Line Allocation

The function `scb_scatter_index_dynamic` computes the **lane multiplier** (`lmul`) and the **cache lines per lane** (`ccnt`) based on the total number of cache lines (`count`) in the buffer. These parameters determine the scattering pattern's structure.

3.2.1 Formulas

$$lmul = \frac{\text{count x } CL}{L2} = \frac{\text{count x } 64}{262{,}144}$$

$$ccnt = \frac{\text{count}}{lmul}$$

Where:

- **`count`:** Total number of cache lines in the buffer.
- **`CL`:** Cache line size (64 bytes).
- **`L2`:** L2 cache size (256 KiB).

3.2.2 Index Assignment

The scattering pattern assigns indices in an interleaved fashion across multiple lanes to ensure that consecutive indices correspond to memory addresses spaced 256 KiB apart. Mathematically, the index assignment is given by:

$$\text{indice}[lmul \times i + j] = i + (j \times ccnt)$$

Where:

- **`i`:** Iterates over cache lines per lane (`0 \leq i < ccnt`).
- **`j`:** Iterates over lanes (`0 \leq j < lmul`).
- **`indice[k]`:** The memory index assigned to the `k^{th}` cache line in the buffer.

This interleaving ensures that memory accesses are uniformly distributed across the buffer, maximizing cache misses and enhancing memory hardness.

3.3 Memory Filling and Hash State Integration

The `scb_fill_memory` function populates the memory buffer using the scattering pattern and integrates iteration counts and memory indices into the hash state to maintain cryptographic linkage.

3.3.1 Process Overview

1. **SHAKE Initialization:** Initializes a local SHAKE state (`kstate`) with the derived key (`ckey`).
2. **Scatter Index Allocation:** Allocates and initializes an array of scatter indices (`indice`) using `scb_scatter_index_dynamic`.
3. **Memory Population Loop:**

   For each cache line index (`i`):

   - **SHAKE Squeeze:** Generates a pseudo-random block (`kblk`) from the SHAKE state.
   - **Memory Copy:** Copies `kblk` to the buffer at the offset determined by `indice[i]`.
   - **Hash State Update:** Updates the main hash state (`hstate`) with the iteration count (`i`) and the memory index (`indice[i]`).
   - **Periodic Buffer Hashing:** At intervals equivalent to the L2 cache size, hashes the entire buffer to reinforce memory hardness.

4. **Resource Cleanup:** Disposes of the local SHAKE state and frees the allocated scatter indices array.

3.3.2 Mathematical Representation

For each iteration `i` and lane `j`:

$$\text{indice}[lmul \times i + j] = \text{i} + (j \times ccnt)$$

$$\text{buffer}[\text{indice}[i] \times CL] = \text{SHAKE}(kstate)$$

$$hstate = \text{SHA3\_Update}(hstate, i, \text{indice}[i])$$

$$\text{if } (i + 1) \bmod \frac{L2}{C1} = 0 \text{ then hstate} = \text{SHA3\_Update}(\text{hstate, buffer})$$

This iterative process ensures that each memory access is cryptographically linked, maintaining the integrity and unpredictability of the derived keys.

4. Security Analysis

The SCB KDF's architecture is designed to withstand a variety of cryptographic attacks by enforcing both computational and memory hardness. This section provides an in-depth analysis of SCB's security mechanisms, highlighting its resilience against specific attack vectors and the theoretical foundations that underpin its robustness.

4.1 Memory and Computational Hardness

Memory Hardness refers to the requirement that a significant amount of memory resources is necessary to compute the function, thereby deterring attackers from efficiently performing brute-force or parallelized attacks. SCB achieves memory hardness through its scattering pattern and iterative hashing processes.

4.1.1 Cache Miss Induction

By enforcing a fixed memory access spacing of (default) 256 KiB, SCB ensures that each memory access exceeds the L2 cache's capacity, inducing a near 100% cache miss rate. This strategy compels the CPU to fetch data from higher latency memory tiers, such as the L3 cache or main memory, thereby increasing the time and energy required for key derivation.

4.1.2 Computational Cost

The **CPU cost parameter** (`cpucost`) determines the number of iterations in the key derivation loop. Each iteration involves SHAKE state updates and memory filling operations, collectively escalating the computational burden. This iterative process ensures that key derivation remains computationally intensive, thereby impeding rapid key generation attempts.

4.2 Resistance to Specific Attack Vectors

4.2.1 Brute-Force Attacks:

- **Challenge:** Systematically trying all possible keys until the correct one is found.
- **SCB's Defense:** The high computational and memory costs imposed by SCB exponentially increase the time required to test each key, rendering brute-force attacks infeasible within practical timeframes.

4.2.2 Dictionary Attacks:

- **Challenge:** Utilizing precomputed lists of likely keys derived from common passwords or seed materials.

- **SCB's Defense:** Memory-hardness ensures that generating and storing comprehensive dictionaries would require exorbitant memory resources, making such attacks impractical.

4.2.3 Rainbow Table Attacks:

- **Challenge:** Employing precomputed chains of keys to reverse-engineer the original seed or password.
- **SCB's Defense:** The iterative and memory-intensive nature of SCB disrupts the feasibility of creating effective rainbow tables, as each table entry would necessitate substantial memory resources and computational effort.

4.2.4 Side-Channel Attacks:

- **Challenge:** Exploiting information leakage through timing discrepancies, power consumption patterns, or electromagnetic emanations to infer key material.
- **SCB's Defense:** The deterministic scattering pattern and uniform memory access intervals obscure access patterns, minimizing timing discrepancies and reducing information leakage through side channels.

4.2.5 Parallelized Hardware Attacks:

- **Challenge:** Leveraging the parallel processing capabilities of GPUs, FPGAs, or ASICs to accelerate key derivation.
- **SCB's Defense:** Each write of a cache line to a memory location, writes the cache position index and loop iterator to the key hash, and the entire buffer is written to the hash at each L2 sized interval (default) 256 KiB, sequential operations that make any significant parallelization impossible.

4.3 Cryptographic Linkage and State Integrity

SCB ensures a cryptographic linkage between iterations by incorporating both the iteration count and memory index into the hash state (`hstate`). This linkage maintains the integrity of the key derivation process, preventing attackers from decoupling computational and memory costs or inferring intermediate states.

4.3.1 Mathematical Linkage

$$\text{hstate} = \text{SHA3\_Update}(\text{hstate}, i, \text{indice}[i])$$

Where:

- **`i`:** Iteration count.
- **`indice[i]`:** Memory index for the current cache line.

This mathematical linkage ensures that each iteration's state is dependent on both the computational and memory aspects, reinforcing the memory-hard property and preventing state compromise.

4.4 Integration with Keccak Functions

SCB leverages the **Keccak** functions (SHAKE-256 and SHAKE-512) due to their proven cryptographic strength and flexibility as extendable-output functions (XOFs). Keccak's sponge construction, comprising absorption and squeezing phases, facilitates the secure generation and manipulation of pseudo-random data essential for SCB's operations.

4.4.1 Sponge Construction

1. **Absorption Phase:**
   Input data (seed, personalization string) is XORed into the Keccak state and permuted using the Keccak permutation function.

2. **Squeezing Phase:**
   Output is generated by applying the permutation function and extracting portions of the state.

SCB utilizes the absorption phase during initialization and the squeezing phase during memory filling and key finalization, ensuring that each pseudo-random block is cryptographically linked to the current state.

5. Internal Functions

The SCB KDF is comprised of several internal functions that collaboratively enforce its security and performance characteristics. This section provides a detailed analysis of these functions, describing their roles, methodologies, and mathematical foundations.

`scb_scatter_index_dynamic`

5.1 Purpose

The `scb_scatter_index_dynamic` function is pivotal in establishing the **scattering pattern** that underpins SCB's memory-hardness. It calculates and assigns indices to the memory buffer in a manner that ensures each access is spaced (default) 256 KiB apart, thereby maximizing cache misses.

5.2 Operational Mathematics

Given:

- **`count`:** Total number of cache lines in the buffer.
- **`QSC_MEMUTILS_CACHE_LINE_SIZE (CL)`:** 64 bytes.

- **`QSC_SCB_L2CACHE_DEFAULT_SIZE (L2):`** 256 KiB (262,144 bytes).

The function computes:

$$lmul = \frac{\text{count x } CL}{L2}$$

For each cache line index `i` and lane `j`, the index assignment is:

$$ccnt = \frac{\text{count}}{lmul}$$

This formula ensures that consecutive entries in the `indice` array correspond to memory addresses spaced 256 KiB apart, thereby enforcing cache miss induction.

`scb_fill_memory`

Purpose

The `scb_fill_memory` function populates a memory buffer with pseudo-random data derived from the SHAKE state, adhering to the scattering pattern established by `scb_scatter_index_dynamic`. This function embodies the memory-hard operation central to SCB's security.

Operational Mathematics

Given:

- **`buflen`:** Length of the memory buffer in bytes.
- **`CL`:** Cache line size (64 bytes).
- **`lcnt`:** Number of cache lines in the buffer, calculated as:

  $$lcnt = \frac{\text{buflen}}{CL}$$

The function performs the following steps for each cache line index `i`:

1. **SHAKE Squeeze:**

   $$kblk = \text{SHAKE\_Squeeze}(kstate, 1)$$

2. **Memory Offset Calculation:**

   $$oft = \text{indice}[i] \times CL$$

3. **Memory Copy:**

$$\text{buffer}[oft] = kblk$$

4. **Hash State Update:**

$$hstate = \text{SHA3\_Update}(hstate, \text{iteration\_count} = i)$$

$$\text{hstate} = \text{SHA3\_Update}(hstate, \text{memory\_index} = \text{indice}[i])$$

5. **Periodic Buffer Hashing:**

$$\text{if } (i+1) \bmod (\frac{L2}{CL}) = 0 \text{ then } hstate = \text{SHA3\_Update}(hstate, \text{buffer})$$

This process ensures that memory accesses are scattered to induce cache misses and that each iteration's computational and memory actions are cryptographically linked.

Public API Functions

The SCB KDF exposes a set of public API functions defined in the `scb.h` header file. These functions facilitate the initialization, generation, updating, and disposal of the SCB state, providing a robust interface for integrating SCB into cryptographic applications.

`qsc_scb_dispose`

- **Purpose:** Securely disposes of the SCB state by clearing sensitive key material and resetting cost parameters.
- **Parameters:**
  - **`ctx`:** Pointer to the SCB state structure.
- **Behavior:** Clears the `ckey` buffer, sets `cpuc`, `memc`, and `klen` to zero, and resets the SHAKE rate to `qsc_keccak_rate_none`.

`qsc_scb_initialize`

- **Purpose:** Initializes the SCB state with a seed, optional personalization string, and memory and CPU cost parameters.
- **Parameters:**
  - **`ctx`:** Pointer to the SCB state structure.
  - **`seed`:** Pointer to the seed data (32 bytes for SHAKE-256, 64 bytes for SHAKE-512).
  - **`seedlen`:** Length of the seed data.
  - **`info`:** Pointer to the optional personalization string.
  - **`infolen`:** Length of the personalization string.

- **`cpucost`:** Number of CPU cost iterations (minimum 1, maximum 1000).
- **`memcost`:** Memory cost in MiB (minimum 1, maximum 10,000).

- **Behavior:**

  - Determines the appropriate SHAKE function based on `seedlen`.
  - Clears and sets the `ckey` buffer.
  - Absorbs the seed and personalization string into the SHAKE state to generate the initial key.

`qsc_scb_generate`

Purpose: Generates pseudo-random bytes based on the SCB state.

- **Parameters:**

  - **`ctx`:** Pointer to the SCB state structure.
  - **`output`:** Pointer to the output buffer where pseudo-random bytes will be written.
  - **`otplen`:** Number of bytes to generate.

- **Behavior:**

  - Allocates a memory buffer based on `memc`.
  - Iteratively updates the SHAKE state and fills the memory buffer using the scattering pattern.
  - Finalizes the SHAKE state with the derived key.
  - Extracts the desired pseudo-random bytes from the finalized SHAKE state.

`qsc_scb_update`

- **Purpose:** Updates the SCB state with new seed material.
- **Parameters:**

  - **`ctx`:** Pointer to the SCB state structure.
  - **`seed`:** Pointer to the new seed data.
  - **`seedlen`:** Length of the new seed data.

- **Behavior:**

  - Absorbs the new seed material into the SHAKE state, integrating it with the existing key material.

Summary of Public API Functions

Function	Purpose	Parameters

qsc_scb_dispose	Securely disposes of the SCB state	`qsc_scb_state* ctx`
qsc_scb_initialize	Initializes the SCB state with seed and costs	`qsc_scb_state* ctx, const uint8_t* seed, size_t seedlen, const uint8_t* info, size_t infolen, size_t cpucost, size_t memcost`
qsc_scb_generate	Generates pseudo-random bytes	`qsc_scb_state* ctx, uint8_t* output, size_t otplen`
qsc_scb_update	Updates the SCB state with new seed material	`qsc_scb_state* ctx, const uint8_t* seed, size_t seedlen`

These functions collectively provide a secure and efficient interface for key derivation, ensuring that SCB can be seamlessly integrated into cryptographic workflows while maintaining stringent security standards.

Cache Theory and Impact Analysis

Understanding the interaction between SCB's memory access patterns and the CPU's cache hierarchy is critical to appreciating SCB's security implications. This section delves into cache theory, elucidating how SCB's design induces cache misses and thrashing to enforce memory hardness.

CPU Cache Hierarchy

Modern CPUs employ a multi-level cache hierarchy to mitigate the latency gap between the fast CPU cores and the slower main memory (RAM). The primary cache levels are:

- **L1 Cache:**
    - **Size:** 16 KiB - 64 KiB per core.
    - **Latency:** Approximately 1 CPU cycle.
    - **Structure:** Typically split into separate instruction (L1i) and data (L1d) caches.
- **L2 Cache:**
    - **Size:** 128 KiB - 1,024 KiB per core.
    - **Latency:** Approximately 3-10 CPU cycles.
    - **Structure:** Often unified, serving both instructions and data.
- **L3 Cache:**
    - **Size:** 2,048 KB - 65,536 KiB shared among all cores.

- **Latency:** Approximately 10-40 CPU cycles.
- **Structure:** Unified and shared across multiple cores.

Cache Misses and Thrashing

A **cache miss** occurs when the CPU attempts to access data not present in the cache, necessitating a fetch from lower cache levels or main memory. **Cache thrashing** refers to the frequent eviction and loading of cache lines, leading to sustained high cache miss rates and degraded performance.

SCB's Induced Cache Misses

SCB enforces a memory access pattern that ensures each access is spaced 256 KiB apart. Given that the typical L2 cache size is 256 KiB, this spacing guarantees that each new access evicts previously loaded cache lines from the L2 cache, thereby inducing a cache miss.

$$\text{Number of Cache Lines per L2 Cache} = \frac{L2}{CL} = \frac{262{,}144}{64} = 4{,}096 \text{ cache lines}$$

Each memory access targets a distinct cache line outside the L2 cache's retention capacity, maintaining a near 100% cache miss rate. This strategic spacing forces the CPU to engage with higher latency memory tiers, thereby increasing the computational and memory access overhead required for key derivation.

Mathematical Impact on Cache Performance

The cache miss rate (`CMR`) can be modeled as:

$$\text{CMR} = 1 - \left(1 - \frac{CL \times \text{Accesses}}{L2}\right)^{\text{Accesses}}$$

Given the fixed spacing of 256 KiB, the `CMR` approaches 1 (or 100%), ensuring maximal cache miss induction. This high cache miss rate significantly elevates the time and energy required for key derivation, enhancing SCB's resistance against resource-intensive attacks.

Impact on Memory Bandwidth and Performance

The enforced cache miss rate translates to increased memory bandwidth consumption and higher latency in memory access operations. This impact is twofold:

1. **Legitimate Operations:**

   While SCB imposes substantial memory and computational costs, legitimate cryptographic operations benefit from the high entropy and unpredictability of derived keys.

2. **Adversarial Attacks:**

   Attackers aiming to execute brute-force or parallelized attacks are constrained by the elevated memory bandwidth and computational requirements, rendering such attacks impractical within feasible timeframes.

Cache Thrashing and Security Enhancement

Cache thrashing, resulting from SCB's scattering pattern, serves as a deterrent against attacks by ensuring that each memory access necessitates a fetch from higher latency memory tiers. This mechanism:

- **Increases Resource Consumption:** Amplifies the CPU and memory resources required for key derivation, escalating the cost of attacks.
- **Reduces Attack Efficiency:** Limits the effectiveness of parallelized and hardware-accelerated attacks by enforcing uniform memory access patterns that are resistant to optimization.

Conclusion

The SHAKE Cost-Based (SCB) Key Derivation Function represents a sophisticated integration of cryptographic robustness and memory-hard principles, delivering a secure and efficient mechanism for deriving cryptographic keys. By meticulously orchestrating memory access patterns to induce cache misses and leveraging the cryptographic strength of SHAKE-256 and SHAKE-512 functions, SCB achieves a high degree of security and resilience against a broad spectrum of attack vectors, including brute-force, dictionary, rainbow table, and side-channel attacks.

SCB's internal mechanisms, notably `scb_scatter_index_dynamic` and `scb_fill_memory`, are expertly designed to enforce memory and computational hardness, ensuring that key derivation remains both resource-intensive and secure. The adaptive configuration of cost parameters (`cpucost` and `memcost`) allows SCB to scale its resistance in alignment with evolving adversarial capabilities, maintaining its efficacy in a dynamic threat landscape.

Furthermore, SCB's foundation on the well-vetted SHAKE functions from the SHA-3 family ensures that the core pseudo-random generation is cryptographically secure, providing a reliable basis for the additional memory-hard mechanisms employed by SCB. The deliberate scattering pattern and iterative hashing processes collectively fortify SCB's defense mechanisms, rendering it a formidable tool in the arsenal of quantum-resistant cryptographic solutions.

In summary, SCB stands as a testament to the intricate interplay between cryptographic ingenuity and system-level optimization, delivering a KDF that is both secure and efficient. As computational paradigms continue to advance, SCB's design principles offer a robust blueprint

for future developments in cryptographic key derivation methodologies, ensuring sustained resilience against emerging threats and maintaining the integrity of cryptographic operations.