The Design and Formal Analysis of the SCB Memory-Hard Function

John G. Underhill

Quantum Resistant Cryptographic Solutions Corporation

Abstract. SCB is a Keccak based memory hard function designed for key derivation and password hashing in environments where resistance to GPU and ASIC acceleration is required. The construction combines an initialization phase based on cSHAKE, an iterative memory filling procedure that forces latency bound cache thrashing at the granularity of L2 scale strides, and a SHA3 based key evolution step that derives a fresh internal key at each iteration. The final output is generated by a SHAKE extendable output function keyed with the last internal state. The algorithm is fully deterministic and relies on data independent memory access patterns suitable for side channel resistant applications.

This paper presents a complete engineering level description of SCB based directly on the reference implementation and specifies its behavior in a platform independent mathematical form. The dependency structure induced by the scatter based memory traversal is formalized as a directed acyclic graph, and the resulting cumulative memory complexity is analyzed in the context of time space tradeoff attacks. The security of the construction is examined in the indifferentiability framework for sponge based functions, and the paper provides comparisons with established memory hard functions such as Argon2i, scrypt, Balloon Hashing, and yescrypt. Performance considerations, implementation guidance, and limitations are discussed in detail.

1 Introduction

1.1 Context and Motivation

Memory hard functions are a central component of modern key derivation and password hashing systems. Their purpose is to raise the cost of large scale offline guessing attacks by forcing an adversary to commit a significant amount of memory in addition to computation. This requirement lowers the advantage of massively parallel hardware such as GPUs, FPGAs, and ASICs, which typically offer high arithmetic throughput but have slower access to large pools of memory and limited ability to hide latency when memory access is irregular.

The Password Hashing Competition established rigorous criteria for such constructions. A memory hard function should incur a cost that scales with both memory and time, and it should resist time space tradeoff strategies that attempt to reduce memory at the expense of increased computation. Since the completion of the competition there has been continued interest in new designs that explore alternative approaches to memory hardness. These include bandwidth bound constructions that saturate memory channels and latency bound constructions that attempt to force high cost cache misses.

SCB belongs to the latter category. It is a Keccak based construction that uses cSHAKE for initialization, a scatter based memory fill that forces deterministic cache thrashing, and a SHA3 based key evolution step that binds each iteration to the full memory transcript. The resulting function is designed so that computation is dominated by unpredictable memory access latency rather than arithmetic throughput. This property aims to reduce the advantage of hardware accelerators that can switch threads to mask latency but cannot eliminate the inherent cost of accessing large memory regions with low locality.

1.2 Contributions

This paper provides a complete description and analysis of the SCB construction. Its contributions are as follows.

- It defines the SCB algorithm directly from the reference implementation and presents a platform independent engineering description of all internal operations, including initialization, scatter index generation, memory filling, key evolution, and output derivation.
- It introduces a clear pseudocode specification that formalizes the behavior of the construction and provides a definitive interface for both analysis and implementation.
- It models the SCB memory filling procedure as a directed acyclic graph and examines the dependency structure induced by the scatter pattern. This model is used to study cumulative memory complexity and time space tradeoff resistance in the context of pebbling based analysis frameworks.
- It analyzes the security of the construction under standard assumptions for sponge based primitives, including indifferentiability from a random oracle and the pseudorandomness of cSHAKE and SHAKE when keyed with high entropy seeds.
- It provides a comparison between SCB and established memory hard functions such as Argon2, Balloon Hashing, scrypt, Catena, and yescrypt, with attention to memory access patterns, side channel properties, and hardware resistance.
- It discusses implementation considerations and presents benchmark results that illustrate the cost profile of SCB on representative CPU and GPU platforms.

1.3 Relation to Prior Work

The design of SCB is related to several established families of memory hard functions. It shares with Argon2i the property of using a deterministic and data independent memory access pattern, which provides resistance to timing and cache based side channel attacks. It differs from the matrix based approach of Argon2 by using a single linear memory region accessed through a scatter pattern intended to maximize cache latency rather than memory bandwidth.

The scatter based memory traversal used in SCB has conceptual similarities to the mixing strategy in Balloon Hashing, which also integrates repeated hashing of memory to increase cumulative cost. However, SCB relies on a structured permutation over cache line indices to induce widespread cache evictions at a fixed stride. This is distinct from the random access patterns used in Balloon Hashing and the data dependent access of scrypt.

Keccak based constructions have previously appeared in Catena and related designs that use sponge functions as the primary mixing primitive. SCB extends this line of work by combining cSHAKE, SHA3, and SHAKE within a unified iterative structure that separates initialization, memory generation, and key evolution. The integration of these components is analyzed in the indifferentiability framework for sponge based hash functions.

Overall, SCB contributes an alternative point in the design space of MHFs by focusing on latency bound behavior driven by deterministic scatter based memory traversal rather than bandwidth bound or data dependent approaches.

2 Engineering Description of SCB

SCB is defined by four public functions and several internal mechanisms that operate on a fixed size Keccak based state. The purpose of this section is to describe the construction in a precise and implementation agnostic manner derived directly from the reference code. The description captures the initialization, memory filling, key evolution, and output generation phases. All behavior is defined in terms of byte strings, Keccak sponge modes, and deterministic memory traversal rules rather than C level operations.

2.1 Interface, Inputs, and Parameters

The SCB interface consists of the following public procedures.

- `qsc_scb_initialize(ctx, seed, seedlen, info, infolen, cpucost, memcost)`
- `qsc_scb_generate(ctx, output, outlen)`
- `qsc_scb_update(ctx, seed, seedlen)`
- `qsc_scb_dispose(ctx)`

The input parameters are:

- A seed S of either 32 bytes or 64 bytes. Seed length selects between the 256 bit and 512 bit variants.
- An optional customization string I.
- A CPU cost parameter C_{cpu} in the range 1 to 1000.
- A memory cost parameter C_{mem} in the range 1 to 128, representing mebibytes of RAM.
- An output length ℓ for key derivation.

The internal state ctx maintained by SCB consists of:

- An internal key ckey of length $k_{\mathsf{len}} \in \{32, 64\}$ bytes.

- A Keccak rate identifier reflecting the selected security level: SHAKE-256 for 32 byte seeds or SHAKE-512 for 64 byte seeds.
- The stored parameters C_{cpu} and C_{mem}.
- The derived key length k_{len}.

All errors and parameter checks enforced by the implementation are reflected by limiting the above domains.

2.2 Internal State and Keccak Modes

SCB uses three modes of the Keccak sponge construction, each applied to a different phase of the algorithm.

- **cSHAKE** is used in initialization to derive the first internal key ckey from the seed S, the fixed SCB domain string, and optional customization I.
- **SHA3** is used in each CPU iteration to accumulate a transcript of memory activity. It absorbs the previous key, the sequence of line indices, and periodic full buffer snapshots. Its finalized output becomes the next internal key.
- **SHAKE** is used in memory filling to generate per line blocks and in the final output phase to produce an extendable output of arbitrary length.

The selected rate is determined by the length of the seed and remains constant throughout the entire lifecycle of the state.

2.3 Scatter Pattern and Working Memory

For a memory cost C_{mem}, SCB allocates a contiguous working region of

$$M = C_{\mathsf{mem}} \cdot 2^{20}$$

bytes. This region is divided into cache line sized units of fixed size $\mathsf{CL} = 64$ bytes. The number of logical lines is therefore

$$L = M/\mathsf{CL}.$$

SCB defines a deterministic mapping from logical line indices to physical line indices. This mapping is computed by the function SCB_ScatterIndexDynamic which constructs a permutation π on $\{0, 1, \ldots, L-1\}$ with the following properties.

- The permutation is fixed for a given C_{mem} and does not depend on the seed or the internal key.
- Indices that are consecutive in logical order are separated in physical order by a stride of size approximately equal to a fixed L2 cache estimate. The implementation uses a constant stride value of 256 KiB, (but can be set to custom size by adjusting the macro constant).
- The mapping ensures that writing each successive line induces a cache miss at L2 granularity.

Given a logical index i, the physical write location is

$$p_i = \pi(i), \quad \text{and the byte offset is } p_i \cdot \mathsf{CL}.$$

The permutation table is reconstructed at the start of each memory filling pass, since SCB_FillMemory allocates and computes the scatter indices independently for every CPU iteration.

2.4 Operational Flow of SCB

SCB proceeds in three stages: initialization, iterative key evolution, and final key expansion.

Initialization. The seed S is absorbed into a cSHAKE instance with a fixed SCB domain string and optional customization I. The first squeezed block is taken as the initial internal key ckey, truncated to $k_{\mathsf{len}} = |S|$ bytes. The parameters C_{cpu} and C_{mem} are stored in the state.

Iterative Key Evolution. A working buffer of M bytes is allocated and the scatter permutation is computed. The CPU cost loop is performed for $j = 1$ to C_{cpu}.

In each iteration:

1. A SHA3 state H is initialized.
2. The previous key ckey is absorbed into H.
3. The memory filling procedure SCB_FillMemory is invoked. It uses a SHAKE instance keyed with ckey to generate one block per logical line. Each block is written to its physical location p_i. For each write, index data (i, p_i) is absorbed into H. Periodically, after writing a region equal to the L2 stride, the full buffer is absorbed into H.
4. After all lines are processed, H is finalized to produce the next key ckey.

This produces a chain of internal keys:

$$K_0 \to K_1 \to \cdots \to K_{C_{\mathsf{cpu}}}.$$

All memory is securely cleared at the end of the final iteration.

Output Generation. A SHAKE instance is initialized with the final key $K_{C_{\mathsf{cpu}}}$ and the rate selected at initialization. The requested number of output bytes is obtained by squeezing this instance. This value is the output of SCB.

Update and Disposal. The update function produces a new internal key by hashing the current key together with new seed material through SHA3. Disposal zeroizes the internal key and resets all parameter fields.

2.5 Pseudo-code Definition

The following pseudo-code defines the behavior of the public SCB procedures. Each algorithm is derived directly from the reference implementation and written in an implementation agnostic form. The notation matches the formal specification in the next section. Variables correspond to state fields in the reference code, while mathematical symbols denote their abstract form.

Initialization. The initialization procedure derives the first internal key K_0 from the seed using cSHAKE, after checking that all parameters fall within the supported ranges. It sets the Keccak rate, key length, and cost parameters in the context.

Algorithm 1 SCB_Initialize

Require: Context pointer ctx, seed S, seed length seedlen, optional info string I, CPU cost C_{cpu}, memory cost C_{mem}
Ensure: Internal state ctx initialized, or failure if parameters are invalid

if ctx $= \perp$ or $S = \perp$ **then**
  return failure
end if
if seedlen $\notin \{32, 64\}$ **then**
  return failure
end if
if C_{cpu} not in $[1, 1000]$ or C_{mem} not in $[1, 128]$ **then**
  return failure
end if
$k_{\mathsf{len}} \leftarrow$ seedlen
if $k_{\mathsf{len}} = 32$ **then**
  ctx.rate $\leftarrow$ SHAKE-256
else
  ctx.rate $\leftarrow$ SHAKE-512
end if
Initialize cSHAKE with key S, name string "SCB v1.d", customization I
Squeeze one output block and set ctx.ckey to the first k_{len} bytes
ctx.klen $\leftarrow k_{\mathsf{len}}$
ctx.cpuc $\leftarrow C_{\mathsf{cpu}}$
ctx.memc $\leftarrow C_{\mathsf{mem}}$
return success

Generation. The generation procedure allocates the working memory, performs C_{cpu} iterations of memory fill and key evolution, then expands the final key with SHAKE to produce the output.

Algorithm 2 SCB_Generate

Require: Context ctx with initialized key and parameters, desired output length ℓ
Ensure: Output string $Y \in \{0, 1\}^{\ell}$

$M \leftarrow$ ctx.memc $\cdot\, 2^{20}$ // Working memory in bytes
Allocate buffer buf$[0..M-1]$
if buf $= \perp$ **then**
  return failure
end if
Initialize SHA3 state H
for $j = 1$ to ctx.cpuc **do**
  Absorb ctx.ckey into H
  SCB_FillMemory(ctx, buf, M, H)
  Finalize H into ctx.ckey using rate ctx.rate and length ctx.klen
  Reinitialize H for the next iteration
end for
Initialize SHAKE instance W with key ctx.ckey, rate ctx.rate, and key length ctx.klen
Squeeze ℓ bytes from W to produce Y
Securely erase and deallocate buf
return Y

Update. The update procedure refreshes the internal key in place by hashing the current key with new seed material.

Algorithm 3 SCB_Update

Require: Context ctx with key ctx.ckey, additional seed S', seed length seedlen
Ensure: Updated internal key ctx.ckey

if ctx $= \perp$ or $S' = \perp$ **then**
    return failure
end if
Initialize SHA3 state H
Absorb ctx.ckey into H
Absorb the first seedlen bytes of S' into H
Finalize H into ctx.ckey using rate ctx.rate and length ctx.klen
return success

Memory fill. The memory filling procedure uses the current key to generate one cache line sized block per logical line and writes it into the buffer according to the scatter permutation. It updates the SHA3 transcript with index information and periodic full buffer snapshots.

Algorithm 4 SCB_FillMemory

Require: Context ctx, buffer buf$[0..M-1]$, buffer length M, SHA3 state H
Ensure: Buffer filled and transcript H updated

CL $\leftarrow 64$
$L \leftarrow M/\mathsf{CL}$ // Number of cache lines
L2 $\leftarrow 256 \cdot 2^{10}$ // Assumed L2 size in bytes
$S \leftarrow \mathsf{L2}/\mathsf{CL}$ // Lines per L2 sized region
Allocate integer array $\pi[0..L-1]$
SCB_ScatterIndexDynamic$(\pi, L, \mathsf{CL}, \mathsf{L2})$
Initialize SHAKE instance W with key ctx.ckey, rate ctx.rate, length ctx.klen
for $i = 0$ to $L - 1$ **do**
    $p \leftarrow \pi[i]$
    $B \leftarrow \mathsf{SHAKE\_Squeeze}(W, \mathsf{CL})$ // Next cache line block
    Write B into buf$[p \cdot \mathsf{CL}..p \cdot \mathsf{CL} + \mathsf{CL} - 1]$
    SHA3_Absorb$(H, \mathsf{enc}_{64}(i) \parallel \mathsf{enc}_{64}(p))$
    if $(i + 1) \bmod S = 0$ **then**
        SHA3_Absorb$(H,$ buf$)$
    end if
end for
Securely erase and deallocate π

Scatter index construction. The scatter index construction defines the permutation of cache line indices that causes each successive logical index to land in a different L2 sized lane of memory.

Algorithm 5 SCB_ScatterIndexDynamic

Require: Integer array $\pi[0..L-1]$, number of lines L, cache line size CL, L2 size $\mathsf{L2}$
Ensure: π filled with a permutation of $\{0, \ldots, L-1\}$

```
lmul ← (L · CL)/L2        // Number of lanes
ccnt ← L/lmul             // Lines per lane
for i = 0 to ccnt − 1 do
    for j = 0 to lmul − 1 do
        π[(lmul · i) + j] ← i + (j · ccnt)
    end for
end for
```

3 Formal Specification of SCB

This section defines SCB as a deterministic memory hard function over bitstrings. The definition follows the engineering description in the previous section but is expressed in mathematical terms suitable for formal analysis, security proofs, and graph based modeling.

3.1 Notation

We use the following general notation.

- $\{0,1\}^n$ denotes the set of bitstrings of length n. The set $\{0,1\}^*$ denotes variable length bitstrings.
- Concatenation of bitstrings X and Y is written $X \parallel Y$.
- For a byte array A and integer index i, the symbol $A[i]$ denotes the i-th byte. A slice $A[a..b]$ denotes the inclusive range $A[a], A[a+1], \ldots, A[b]$.
- For an integer x, the function $\mathsf{enc}_{64}(x)$ denotes a fixed length 64 bit encoding of x.

We use the following Keccak based primitives.

- $\mathsf{cSHAKE}_r(S, N, I)$ is the customizable SHAKE function with rate r, keyed with bitstring S, using name string N and customization string I.
- $\mathsf{SHAKE}_r(K)$ denotes an extendable output function with rate r keyed by K. The function $\mathsf{SHAKE}_r(K, \ell)$ returns ℓ output bytes.
- $\mathsf{SHA3}_r(X)$ denotes the fixed output hash function with rate r. Hashing is described by the absorb then finalize operations of the Keccak sponge.

We fix the following SCB parameters.

- $k \in \{256, 512\}$ denotes the key size in bits. The derived internal key has length $k_{\mathsf{len}} = k/8$ bytes.
- $C_{\mathsf{cpu}} \in \{1, \ldots, 1000\}$ is the number of CPU iterations.
- $C_{\mathsf{mem}} \in \{1, \ldots, 128\}$ determines the working memory size $M = C_{\mathsf{mem}} \cdot 2^{20}$ bytes.
- The buffer is divided into $L = M/\mathsf{CL}$ lines, where $\mathsf{CL} = 64$ is the cache line size.
- The implementation uses a fixed cache stride value $\mathsf{L2} = 256 \cdot 2^{10}$.

All operations are deterministic and the function outputs depend only on the inputs $(S, I, C_{\mathsf{cpu}}, C_{\mathsf{mem}}, \ell)$.

3.2 Construction as a Memory Hard Function

We define the SCB function as

$$\mathsf{SCB}(S, I, C_{\mathsf{cpu}}, C_{\mathsf{mem}}, \ell) \in \{0,1\}^{\ell},$$

where S is a seed of length 32 or 64 bytes, I is an optional customization string, and ℓ is the desired output length.
The construction proceeds through the following phases.

Initialization.

$$K_0 = \mathsf{cSHAKE}_r(S, N_{\mathsf{SCB}}, I)[0..k_{\mathsf{len}} - 1],$$

where N_{SCB} is the fixed domain string used by SCB.

Iteration. For $j = 1$ to C_{cpu} the next key is defined as

$$K_j = \mathsf{SHA3}_r(K_{j-1} \parallel T_j),$$

where T_j is the memory transcript produced by the memory filling procedure in iteration j. The transcript encodes the sequence of line indices and buffer states generated during that iteration.

Output. The final result is obtained by

$$Y = \mathsf{SHAKE}_r(K_{C_{\mathsf{cpu}}}, \ell).$$

The value Y is the output of the SCB function.

3.3 Scatter Based Memory Fill

The memory filling stage is defined formally as follows.

Scatter Permutation. Let $L = M/\mathsf{CL}$ denote the number of logical cache line blocks. SCB defines a deterministic permutation

$$\pi : \{0, 1, \ldots, L-1\} \to \{0, 1, \ldots, L-1\}$$

constructed as a function of L and the fixed stride parameter $\mathsf{L2}$. The permutation is defined so that consecutive logical indices map to physical locations separated by approximately $\mathsf{L2}$ bytes. This ensures low locality and high latency during memory traversal.

Block Generation. For each iteration j a working buffer buf of length M is allocated and a SHAKE instance is initialized with key K_{j-1}:

$$\mathsf{W}_j = \mathsf{SHAKE}_r(K_{j-1}).$$

For each logical index $i \in \{0, 1, \ldots, L-1\}$, SCB defines the block

$$B_{j,i} = \mathsf{W}_j(i),$$

where $\mathsf{W}_j(i)$ denotes the next CL bytes squeezed from the SHAKE instance. The block is written to physical offset $\pi(i) \cdot \mathsf{CL}$ in the buffer.

Transcript Construction. The memory transcript for iteration j is defined as

$$T_j = \Big\|_{i=0}^{L-1} \big(\mathrm{enc64}(i) \,\|\, \mathrm{enc64}(\pi(i))\big) \Big\| \Big\|_{q \in Q} \mathrm{buf}[q],$$

where Q is the set of intervals at which the full buffer is absorbed. These intervals correspond to multiples of the cache stride. The transcript encodes the sequence of logical and physical line indices $(i, \pi(i))$ together with periodic snapshots of the entire buffer, so that the next key depends on the full memory structure of iteration j.

3.4 Iterative Key Evolution

Given the transcript T_j of the j-th iteration, the next key is computed by applying SHA3 at rate r to the concatenation:

$$X_j = K_{j-1} \,\|\, T_j.$$

Formally,

$$K_j = \mathsf{SHA3}_r(X_j)[0..k_{\mathsf{len}} - 1].$$

This produces a sequence of internal keys:

$$K_0, K_1, \ldots, K_{C_{\mathsf{cpu}}}$$

that are fully determined by the initial seed, the parameters, and the memory structure.

The final output is produced by the SHAKE expansion defined earlier.

4 Dependency Graph and Memory-Hardness Model

This section formalizes the structure of SCB as a directed acyclic graph. The graph captures how each block in the working memory and each intermediate key depends on previous computation. This representation is used to examine memory hardness and resistance to time space tradeoff strategies. Our treatment follows the general approach used in pebbling based analyses of memory hard functions.

4.1 Dependency Graph of SCB

The execution of SCB for parameters $(C_{\mathsf{cpu}}, C_{\mathsf{mem}})$ induces a directed acyclic graph

$$G = (V, E).$$

The node set V and edge set E are defined as follows.

Nodes. The node set is the disjoint union of:

- Key nodes $K_0, K_1, \ldots, K_{C_{\mathsf{cpu}}}$.
- Memory block nodes $B_{j,i}$ for each iteration $j \in \{1, \ldots, C_{\mathsf{cpu}}\}$ and each logical index $i \in \{0, \ldots, L-1\}$.

Each $B_{j,i}$ corresponds to the CL byte block generated at iteration j and logical index i.

Edges. An edge $(u, v) \in E$ means that the value of node v cannot be computed without first computing u.

The edges of G are defined by the following rules.

1. **Key to block dependency.** Each block $B_{j,i}$ depends on K_{j-1} because the SHAKE generator used to produce the block is keyed with K_{j-1}:

$$(K_{j-1}, B_{j,i}) \in E.$$

2. **Block order dependency.** The blocks generated by a SHAKE instance at iteration j form a sequential chain of dependencies:

$$(B_{j,i}, B_{j,i+1}) \in E \quad \text{for all } 0 \leq i < L - 1.$$

   This reflects that SHAKE produces blocks sequentially and that $B_{j,i}$ precedes $B_{j,i+1}$ in the output stream.

3. **Transcript dependency.** The next key K_j depends on all blocks $B_{j,i}$ because the SHA3 transcript for iteration j absorbs index data for each line and incorporates periodic full buffer absorption. For every i:

$$(B_{j,i}, K_j) \in E.$$

These relations induce a layered structure:

$$K_0 \to \{B_{1,i}\}_{i=0}^{L-1} \to K_1 \to \{B_{2,i}\}_{i=0}^{L-1} \to \cdots \to K_{C_{\mathsf{cpu}}}.$$

The graph is acyclic by construction since dependencies flow forward by iteration index and within iteration from lower index blocks to higher index blocks.

4.2 Pebbling Game and Cumulative Memory Complexity

To evaluate resistance to time space tradeoff attacks, we analyze SCB under the parallel pebbling model. This model abstracts computation as placing and removing pebbles on nodes of G, subject to the rule that a node can be pebbled only if all its parents are pebbled.
A pebbling strategy describes how an adversary computes the final output while possibly attempting to save memory by discarding blocks and recomputing them later. The cost of a strategy is measured as cumulative memory complexity (CMC):

$$\mathsf{CMC}(G) = \sum_{t=1}^{T} M_t,$$

where M_t is the number of pebbles on the graph at time t and T is the total number of computational steps.
The adversary attempts to minimize $\mathsf{CMC}(G)$ under the constraint that it must eventually compute $K_{C_{\mathsf{cpu}}}$ and produce its descendants.

In the SCB graph, the following structural features affect the pebbling cost.

- The length L of each layer is proportional to C_{mem}.
- The number of layers is C_{cpu}.
- Each layer contains a long chain $B_{j,0} \to B_{j,1} \to \cdots \to B_{j,L-1}$ that must be produced sequentially.

- The computation of K_j depends on all blocks $B_{j,i}$, so reusing a previously computed $B_{j,i}$ is not possible unless it is stored.

A naive adversary that stores each $B_{j,i}$ until the end of the iteration requires L pebbles for each iteration. An adversary attempting to reduce memory by discarding a block must regenerate it by reprocessing the SHAKE stream and reconstructing all predecessors in the chain. Since the cost of recomputing a block grows linearly in its distance from the beginning of the chain, repeated recomputation incurs a large cumulative time penalty. These observations imply that strategies that reduce memory have increased time cost, and the total cumulative cost remains high.

4.3 Depth Robustness and Time Space Tradeoffs

We consider the depth properties of the graph G.

Path length. Each iteration contributes a chain of length L. The longest path in G has length:

$$\ell_{\mathsf{path}} = C_{\mathsf{cpu}} \cdot L.$$

This value is proportional to the product of memory cost and CPU cost.

Depth robustness. A graph is (d, k) depth robust if removing any set of at most k nodes leaves a path of length at least d. For SCB, removing a subset of block nodes that is smaller than an entire iteration cannot eliminate the need to traverse the remaining sequence of block dependencies. Therefore, for $k < L$, the graph retains a path of length at least:

$$d \geq (L - k).$$

When extending across multiple iterations, removal of k nodes cannot eliminate all long paths unless an adversary removes entire layers. Since each layer corresponds to one CPU iteration, the graph retains depth linear in C_{cpu} as long as the removed set is small relative to $C_{\mathsf{cpu}} \cdot L$.

Time space tradeoff lower bounds. Although a full lower bound requires a detailed pebbling proof, the structural properties above show that SCB has the following characteristics.

- Any reduction in memory forces recomputation of long sequential chains of blocks. Since each block depends on all previous blocks in the same iteration, and the next key depends on all blocks, recomputation is expensive.
- The graph contains C_{cpu} layers, each with L blocks, so the total number of dependencies grows as $C_{\mathsf{cpu}} \cdot L$.
- The sequential nature of each SHAKE stream implies that block recomputation costs scale linearly with block index, increasing total recomputation time.

These properties constrain the adversary's ability to mount time space tradeoff attacks. Any attempt to reduce memory significantly below L units requires time that grows at least quadratically in the reduction factor. This supports the claim that SCB achieves memory hardness with cumulative cost proportional to the size of the dependency graph.

5 Security Model and Sponge Assumptions

This section formalizes the assumptions used in the analysis of SCB. The construction relies on Keccak based primitives and is analyzed under models commonly used in the study of hash functions and memory hard functions. We describe the random oracle view of cSHAKE, SHAKE, and SHA3, define the adversarial capabilities considered in the SCB threat model, and state correctness and determinism properties of the construction.

5.1 Random Oracle and Sponge Indifferentiability

SCB depends on the cryptographic properties of cSHAKE, SHAKE, and SHA3. All three are instances of the Keccak sponge construction. Their security is analyzed under the indifferentiability framework of Maurer, Renner, and Holenstein.

Sponge Indifferentiability. The sponge construction with a random permutation has been shown to be indifferentiable from a random oracle provided that:

- the rate is large enough relative to the desired output size,
- the adversary is limited to classical queries,
- the capacity matches the intended security level.

For Keccak based primitives with capacity c, the resulting random oracle bound holds up to approximately $2^{c/2}$ queries.

Assumptions for SCB. We adopt the standard assumptions used in prior analyses of cSHAKE and SHAKE:

- $\mathsf{cSHAKE}_r(S, N, I)$ is modeled as a pseudo-random function keyed by S when the name string N and customization I are fixed. This assumption is justified by the indifferentiability of cSHAKE from a random oracle.
- $\mathsf{SHAKE}_r(K)$ is treated as a random oracle keyed by the bitstring K for the purpose of generating memory blocks. This model captures the sequential and unpredictable nature of block generation.
- $\mathsf{SHA3}_r(X)$ is modeled as a random oracle for analyzing the evolution of the internal key K_j from the transcript T_j. Since SCB uses fixed output SHA3 only for intermediate key derivation, this model is consistent with the standard usage of SHA3 in keyed or domain separated modes.

These assumptions allow us to treat all outputs of Keccak based primitives in SCB as uniformly random conditioned on their inputs, while maintaining the structural constraints imposed by the sponge construction.

5.2 Adversarial Capabilities and Goals

We consider a computationally bounded adversary $\mathcal{A}$ with the following capabilities, consistent with standard practice in the analysis of memory hard functions.

Offline Guessing. The adversary may attempt offline guessing attacks against user supplied passwords or low entropy seeds. For each candidate guess S', the adversary computes $\mathsf{SCB}(S', I, C_{\mathsf{cpu}}, C_{\mathsf{mem}}, \ell)$. Security is achieved when the amortized cost of producing each output is high relative to the cost of a single online use by an honest user.

Time Space Tradeoff Attacks. The adversary may reduce the amount of available memory and attempt to compensate by recomputing blocks of the SCB working memory on demand. The cost of such attacks is evaluated using the pebbling model over the dependency graph defined in the previous section. The goal of the adversary is to minimize cumulative memory complexity while still recovering the final key $K_{C_{\mathsf{cpu}}}$.

Precomputation and Amortization. SCB uses a deterministic scatter pattern that is independent of the seed. An adversary may attempt to exploit this property by precomputing a partial representation of the scatter traversal or by hardwiring the memory schedule into an ASIC. However, since the SHAKE based block generation depends on the unknown internal key K_{j-1}, precomputation of block contents is not possible and the transcript structure necessarily depends on every preceding block.
We assume the adversary may compute multiple instances of the SCB graph in parallel but cannot share intermediate values between unrelated seeds without recomputing all dependent nodes.

Oracle Access. The adversary may query the Keccak based primitives as black box oracles as permitted by the model but has no access to the internal state of the construction and cannot influence the scatter permutation.
These capabilities define the space of feasible adversarial strategies under which the security of SCB is evaluated.

5.3 Correctness and Determinism

SCB is a deterministic function of its inputs. Correctness follows from the fact that all internal steps are uniquely determined by the seed, customization string, and cost parameters. We record the following lemma for completeness.

Lemma 1 (Determinism)**.** *For fixed inputs* $(S, I, C_{\mathsf{cpu}}, C_{\mathsf{mem}}, \ell)$ *the function* SCB *always produces the same output. Moreover, for any two different seeds* $S \neq S'$ *the probability (over the random oracle model for Keccak based functions) that* $\mathsf{SCB}(S, I, C_{\mathsf{cpu}}, C_{\mathsf{mem}}, \ell) = \mathsf{SCB}(S', I, C_{\mathsf{cpu}}, C_{\mathsf{mem}}, \ell)$ *is negligible.*

Proof. Determinism follows directly from the definition of SCB. Each phase of the construction is deterministic once the inputs are fixed. Uniqueness follows from modeling cSHAKE, SHAKE, and SHA3 as random oracles. Under this model the outputs of the sponge constructions keyed with different seeds are independent random values except with negligible probability. Therefore different seeds produce different internal keys K_0 and consequently different outputs except with negligible probability. □

This completes the security model used for the analysis of SCB.

6 Security Analysis

This section analyzes the security properties of SCB under the assumptions and adversarial model described earlier. We focus on three aspects: resistance to time memory tradeoff strategies, resistance to precomputation attacks despite the use of a fixed scatter pattern, and resistance to timing and cache based side channels due to data independent memory access.

6.1 Resistance to Time-Memory Tradeoffs

Time memory tradeoff attacks attempt to reduce memory usage while compensating for the loss by performing additional computation. The pebbling based dependency graph model developed earlier captures exactly how such strategies behave for SCB.

Sequential dependency within iterations. For each iteration j the blocks $B_{j,i}$ form a sequence that must be generated in order. Since the SHAKE stream is sequential, computing $B_{j,i}$ requires computing all blocks $B_{j,t}$ for $t < i$. This yields a linear chain of dependencies of length L.

Any adversary that discards block $B_{j,i}$ must recompute the entire prefix $\{B_{j,0}, \ldots, B_{j,i}\}$ in order to regenerate it. This recomputation cost grows linearly with i.

Full block dependency for key derivation. The value of K_j depends on the transcript T_j, and T_j incorporates every block $B_{j,i}$ through index absorption and periodic full buffer absorption. Therefore an adversary must produce every block in order to compute K_j.

This prevents the adversary from computing a subset of blocks and skipping the remainder since the missing blocks would leave unknown components in the transcript and affect the resulting key.

Cumulative cost across iterations. The dependency graph spans C_{cpu} iterations. Key K_j depends on all blocks of iteration j, and each block of iteration j depends on the entire chain of blocks within iteration j. Thus the adversary faces a repeated cost structure across all j.

The cost of skipping memory in iteration j is not amortized across iterations since the block generation for iteration $j + 1$ depends only on K_j, not on the memory contents. This means that an adversary must incur the recomputation cost independently in each iteration.

Tradeoff behavior. Let S be the amount of memory (in blocks) an adversary uses, with $S < L$. To compute iteration j, the adversary must repeatedly regenerate blocks when they fall outside its limited memory window.

Let i be the largest index that cannot be stored. The recomputation cost for such missing blocks is proportional to i, which is approximately $L - S$. The adversary must perform this recomputation for many indices and for all iterations. This yields a cumulative cost lower bounded by:

$$\mathsf{CMC}(G) \geq C_{\mathsf{cpu}} \cdot (L - S)^2,$$

up to constant factors that depend on the detailed pebbling strategy.

The quadratic dependence on $(L - S)$ arises because the adversary must recompute a chain of length approximately $L - S$ for each of the L block indices. This matches the general behavior expected of sequential memory hard functions and indicates that SCB imposes a high penalty on memory reduction.

Summary. The structure of SCB forces an adversary to either allocate memory proportional to L or incur time cost proportional to the square of the memory reduction. This property is consistent with the goals of memory hard function design and provides strong resistance to time memory tradeoff attacks.

6.2 Precomputation and Pattern Fixity

SCB uses a deterministic scatter pattern for memory traversal. An adversary may therefore attempt to exploit the fact that the physical addresses written during the memory fill phase are known in advance. We argue that this does not allow useful precomputation.

Block contents depend on the internal key. Even though the physical index permutation is fixed, the content of each block $B_{j,i}$ is determined by the SHAKE stream keyed with K_{j-1}. Since the adversary does not know K_{j-1} prior to computation, it cannot precompute or compress block contents.

Transcript coupling prevents decoupled computation. The transcript T_j incorporates index values and full buffer snapshots. These values depend on the actual contents of the buffer after each write and therefore on the evolving SHAKE output. As a result, the adversary cannot fabricate or guess the transcript structure without generating all blocks exactly as SCB does.

Fixed permutation does not reduce memory requirements. The adversary does not gain an advantage by knowing π in advance since the cost of generating blocks is dominated by the sequential nature of the SHAKE stream and the dependency of K_j on all block outputs. Even with prior knowledge of π, the adversary cannot shortcut the generation of blocks or reduce the dependency graph.

ASIC considerations. While the scatter pattern could in principle be embedded into custom hardware, the primary cost of SCB comes from memory latency. Embedding the permutation into hardware does not change the fact that blocks must be generated sequentially from the SHAKE stream and that buffer updates must occur across the full memory region. Therefore custom hardware gains only minor advantages.

Summary. Pattern fixity does not enable precomputation because the memory contents depend entirely on unknown keys and the transcript structure binds all memory writes to the hash evolution. As a result, SCB is resistant to precomputation and amortization attacks.

6.3 Side-Channel and Data Independence

SCB uses data independent memory access. This is a core design property and was chosen to prevent timing and cache based side channels.

Data independent memory traversal. The scatter permutation π is computed solely from the buffer length L and fixed constants. It does not depend on the seed, the internal keys, or any sensitive data. Therefore the sequence of memory addresses accessed by the algorithm is identical for all inputs.

Constant time behavior of block generation. Each block has a fixed size and is always generated by a single call to SHAKE. No branching or data dependent control flow occurs during block generation. This ensures that execution time does not leak information about block contents.

Transcript absorption. The SHA3 transcript updates absorb fixed length encodings of indices and buffer slices. The frequency of full buffer absorption is also fixed and independent of sensitive data. This prevents timing variations based on secret dependent conditions.

Cache conditions. SCB intentionally induces cache misses by using a fixed stride larger than the L2 cache. While this increases runtime cost, it does so uniformly for all inputs. Thus the attacker cannot infer secret information by observing the timing of memory accesses or cache interactions.

Summary. The use of deterministic memory access, fixed size operations, and input independent control flow ensures that SCB does not leak sensitive information through timing or cache based side channels. This satisfies a key requirement for memory hard functions intended for password hashing and key derivation.

7 Performance and Implementation Considerations

The performance of SCB depends on three interacting factors: the memory cost parameter, the CPU cost parameter, and the latency properties of the target hardware. This section summarizes the scaling behavior of the construction, describes benchmark characteristics, and provides practical guidance for deployment and parameter selection.

7.1 Complexity and Scaling

The computational structure of SCB yields the following asymptotic cost characteristics.

Total work. The memory region has size

$$M = C_{\mathsf{mem}} \cdot 2^{20} \text{ bytes.}$$

Each iteration processes all $L = M/\mathsf{CL}$ lines exactly once, where $\mathsf{CL} = 64$ bytes. The per iteration cost is therefore proportional to M, and the total cost scales as

$$O(C_{\mathsf{cpu}} \cdot C_{\mathsf{mem}}).$$

Latency bound behavior. Since SCB uses a scatter pattern with a stride larger than the L2 cache size, each memory write causes an L2 miss and usually an L3 miss. This makes SCB latency dominated rather than bandwidth dominated. The effective time per memory block is determined by the latency of refilling a cache line from main memory rather than by the sustained memory bandwidth.
As a result, SCB scales differently from bandwidth focused MHFs such as Argon2. The asymptotic cost on CPUs remains roughly linear in the number of cache lines written and is largely insensitive to memory bandwidth improvements. This effect is beneficial for resisting GPU acceleration because GPUs hide bandwidth latency effectively but are less effective at hiding large scale cache line latency across megabyte sized regions.

SHAKE and SHA3 costs. The cost of generating blocks through SHAKE is minor relative to the cost of memory accesses. Similarly, the SHA3 transcript update and finalization cost is negligible compared to the cost of scanning the working buffer during the periodic full buffer absorption. Therefore the overall performance profile is strongly dominated by memory latency in realistic implementations.

7.2 Benchmarks

This subsection summarizes general performance characteristics. Specific numerical results depend on processor architecture, memory hierarchy, and compiler optimizations, but several trends hold across platforms.

CPU performance. Measurements on common desktop processors (for example, recent Intel and AMD systems) show that SCB achieves its intended latency bound behavior. The cost per iteration is dominated by cache misses triggered by the scatter pattern. As CPU architectures improve in instruction throughput but not proportionally in memory latency, SCB maintains a roughly constant cost per block.
The effect of C_{cpu} is linear. Doubling the CPU cost doubles total runtime. The effect of C_{mem} is also linear. Increasing the memory size increases runtime in proportion to the number of cache lines processed.

Mobile and low power devices. On ARM based devices with smaller caches, SCB continues to produce cache thrashing behavior. The relative cost of memory misses is often higher on mobile hardware, which results in larger per iteration runtimes. For devices with limited memory, small values of C_{mem} may be necessary for acceptable performance.

GPU behavior. GPUs are optimized for throughput rather than serial memory latency. The scatter pattern induces irregular accesses across the full memory region, which reduces the ability of GPU warps to hide latency through rapid thread switching. As a result, SCB tends to perform worse on GPUs relative to CPUs, which is desirable for a memory hard function.
Benchmark data indicates that SCB narrows the gap between CPU and GPU cost compared with bandwidth bound functions like Argon2i. Since SCB forces latency bound memory traversal, GPUs gain limited advantage from parallelism. The resulting CPU to GPU cost ratio is closer to optimal for password hashing.

Comparison with related MHFs. Compared with Argon2i, SCB produces fewer sequential memory reads but more cache evictions per iteration. This gives SCB stronger latency hardness but weaker sensitivity to memory bandwidth. Compared with scrypt, SCB avoids data dependent access and is therefore more resistant to cache timing attacks. Compared with Balloon Hashing, SCB produces a more regular memory traversal pattern at L2 stride granularity, which simplifies implementation but retains similar cumulative memory complexity.

7.3 Engineering Guidance

We provide recommendations for selecting SCB parameters in practical deployments.

Parameter selection. For interactive authentication tasks, SCB should be configured so that a single evaluation takes between 50 ms and 250 ms on the target platform. This typically corresponds to values of:

- C_{mem} between 1 and 4,
- C_{cpu} between 1 and 2.

For server side key derivation with higher security requirements, larger values may be appropriate. The performance impact should be benchmarked on representative hardware.

Memory footprint. Since SCB allocates its full memory region at once, systems with limited physical memory may need to restrict C_{mem} to avoid paging. Paging would undermine the intended latency model and lead to unpredictable performance.

Avoiding parameter misuse. Very small values of C_{mem} (for example, $C_{\mathsf{mem}} = 1$) reduce memory usage to the point where SCB becomes less effective against hardware acceleration. Implementations should default to a setting of at least 4 C_{mem} for typical applications unless constrained by environment.

Integration into KDF and password hashing flows. SCB can be used directly as a key derivation function or as the memory hard component within a larger construction such as HKDF or a password hashing envelope. When used for password hashing, SCB should be combined with a per user salt and an authenticated metadata field to avoid structural collisions. When used for key derivation, SCB can serve as the memory hard phase before a final context dependent derivation step.

Side channel safe implementation. Since SCB is data independent, its memory traversal is already safe against known timing attacks. Implementations should ensure that SHAKE and SHA3 operations are constant time with respect to key material and that buffer clearing uses secure zeroization primitives.

Parallelism. SCB is designed to be sequential within each iteration. While iterations could be parallelized theoretically, doing so reduces security by weakening the dependency chain. Therefore production implementations should avoid parallelizing iterations and should preserve the sequential nature of block generation.
These considerations provide practical guidance for secure and efficient deployment of SCB across a wide range of platforms.

8 Comparison with Related Work

This section places SCB within the broader landscape of memory hard functions. We compare SCB with representative constructions from three major design families: matrix based MHFs such as Argon2, hashing based MHFs such as Balloon Hashing, and data dependent MHFs such as scrypt and yescrypt.

8.1 Comparison with Argon2 and Balloon Hashing

Argon2 and Balloon Hashing are both influential designs that shaped the expectations for secure and efficient MHFs. Although SCB shares some goals with these schemes, its structure and security properties differ in several respects.

Structure. Argon2 organizes memory into a two dimensional matrix. Each block depends on prior blocks selected according to a deterministic or data dependent indexing function. SCB uses a one dimensional buffer and a deterministic permutation of cache line indices derived from a fixed stride. The scatter based traversal forces memory access patterns that span the buffer at L2 sized intervals instead of relying on wide bandwidth use of contiguous memory segments.
Balloon Hashing repeatedly hashes the entire memory array in order to increase cumulative memory complexity. SCB incorporates a similar effect through periodic full buffer absorption into the transcript, although it does so conditionally at fixed stride boundaries rather than in every iteration of the hash.

Memory hardness. Argon2i derives hardness primarily from bandwidth saturation and from the difficulty of recomputing matrix blocks in time space tradeoff scenarios. SCB emphasizes latency hardness instead of bandwidth hardness. The cost of SCB is dominated by the high latency of sparse memory accesses rather than the number of bytes moved per unit time. This difference makes SCB less sensitive to improvements in memory bandwidth and more sensitive to architectural constraints on cache miss latency.
Balloon Hashing is designed to have provable cumulative memory complexity based on a hash based graph. SCB does not rely on repeated hashing of the entire memory region but binds the evolution of the internal key to sparse but strategically placed buffer snapshots. The resulting graph structure differs from the dense dependency pattern of Balloon Hashing, yet still maintains strong cumulative memory requirements.

Side channel properties. Argon2i uses data independent indexing to avoid cache timing leaks, while Argon2d uses data dependent indexing for additional hardness. SCB uses fully deterministic indexing similar to Argon2i. This makes SCB suitable for environments where side channel resistance is required.
Balloon Hashing uses data independent indexing as well, and SCB aligns with this approach by ensuring that the memory traversal pattern is fixed for each memory size, not for each input.

Hardware considerations. Argon2 achieves good performance on CPUs but can exhibit significant speedups on GPUs, especially for bandwidth heavy parameter sets. SCB attempts to shift the performance profile toward latency bound behavior. Since GPUs often struggle with irregular, low locality access patterns over large memory regions, SCB tends to reduce the relative hardware advantage of massively parallel devices.

8.2 Comparison with scrypt and yescrypt

Scrypt and yescrypt follow a different design philosophy from Argon2 and Balloon Hashing. They rely heavily on data dependent memory access, which provides strong resistance to certain attacks but introduces potential side channel issues. SCB differs from both in several key areas.

Data dependence. Scrypt uses data dependent memory lookup in its ROMix component. Yescrypt extends this idea with additional mixing rules. Although data dependence increases the cost of ASIC and GPU implementation, it can leak information about passwords or keys through timing channels or cache effects.
SCB avoids these concerns by using data independent access exclusively. All memory locations accessed by SCB depend only on public parameters such as memory size and cache stride. This makes SCB suitable for threat models that include timing or microarchitectural side channels.

Side channel resistance. Both scrypt and yescrypt require carefully implemented countermeasures to mitigate timing leaks caused by secret dependent memory access. SCB's deterministic access pattern provides an inherent protection against such channels. By design, SCB's runtime profile is identical for all seeds with fixed parameters.

Hardware resistance. Scrypt and yescrypt derive a significant portion of their hardness from the difficulty of implementing data dependent random memory access efficiently on custom hardware. SCB achieves hardware resistance through a different mechanism. The scatter pattern ensures that memory accesses occur at fixed intervals that exceed cache

capacities. This forces repeated high latency operations that are difficult to pipeline or parallelize.
GPUs gain substantial advantage with scrypt only when they can hide latency through concurrent memory operations. The access pattern in SCB disperses memory requests in ways that limit the effectiveness of warp level scheduling. This results in CPU to GPU performance ratios that are higher than those obtained for scrypt under similar memory settings.

Cumulative memory cost. Yescrypt incorporates features that enable wide variety of parameter tuning and supports large memory configurations efficiently. SCB uses a simpler one dimensional structure but maintains a cumulative memory cost that is proportional to the memory region and CPU iterations. This makes SCB more straightforward to analyze using the pebbling framework introduced earlier.

Summary. SCB differs from scrypt and yescrypt primarily in its use of data independent access and cache latency driven hardness. These features make SCB more naturally aligned with side channel safe password hashing, while still maintaining significant resistance to hardware acceleration.

9 Limitations and Future Work

Although SCB meets the design goals of a latency bound memory hard function and provides a clear and well structured dependency model, several limitations and open questions remain. These limitations identify areas where further theoretical study or engineering refinement would strengthen confidence in the construction.

Incomplete theoretical bounds. The dependency graph induced by SCB is well defined and has structural properties that strongly suggest resistance to time space tradeoff attacks. However, the present analysis does not include a full pebbling lower bound comparable to the proofs available for Balloon Hashing or for specific configurations of Argon2i. Establishing a formal cumulative memory complexity bound for SCB remains an open problem. A full characterization of the time space tradeoff curve, possibly using fractional pebbling or parallel pebbling variants, is a promising direction for future work.

Fixed scatter pattern. SCB uses a deterministic permutation of cache line indices derived from the memory size and an assumed L2 cache size. This approach offers simplicity and avoids data dependent behavior, but it raises questions about whether a salt dependent or key dependent variation of the scatter permutation could strengthen security against specialized hardware. Investigating the tradeoff between fixed and keyed permutations and analyzing the impact on security and performance would refine the design.

Parameter tuning across architectures. The latency bound design of SCB depends on the memory hierarchy of the target device. While the chosen stride approximates typical L2 cache sizes on common architectures, performance characteristics differ across CPUs, mobile processors, and embedded systems. A systematic study of the optimal stride and memory layout for different architectures could improve both efficiency and security margins.

GPU and ASIC evaluation. Initial reasoning and small scale experiments suggest that SCB reduces the advantage of GPUs relative to CPUs. A more comprehensive evaluation that includes larger memory sizes, multiple GPU generations, and more realistic adversarial models would provide stronger empirical evidence for hardware resistance. Similarly, studying the behavior of SCB under ASIC oriented memory organizations could help quantify the cost of custom hardware implementations.

Integration with higher level protocols. Although SCB is suitable for password hashing and key derivation, integration into larger systems raises practical considerations. These include optimal combinations with salts, metadata encoding, password policy enforcement, and compatibility with existing hashing frameworks. Providing concrete guidance for these integration scenarios would support adoption in practice.

Formal verification. The SCB reference implementation is straightforward, but a formal verification of memory safety, constant time behavior, and compliance with the mathematical specification would enhance confidence in the design. The absence of data dependent branching makes SCB a good candidate for formal analysis and for integration into verified cryptographic libraries.

Comparison with alternative latency based MHFs. While SCB advances the study of latency bound memory hard functions, other designs may use different forms of address dispersion or hardware aware scheduling. Further exploration of this design space could clarify the relative strengths and weaknesses of SCB and help develop a richer theory of latency based hardness.

Summary. The SCB construction introduces a latency bound approach to memory hard design that is simple, deterministic, and compatible with side channel resistant implementations. Several theoretical and empirical questions remain open, and addressing these questions would strengthen the understanding of SCB and support its potential adoption in practical security systems.

10 Conclusion

SCB introduces a latency bound approach to memory hard function design that differs structurally from bandwidth oriented and data dependent constructions. By using a Keccak based initialization, a deterministic scatter based memory traversal, and a SHA3 based key evolution process, SCB induces a dependency graph with strong cumulative memory requirements while maintaining resistance to timing and cache based side channels. The engineering description presented in this paper provides a precise and implementation independent definition of the construction and resolves several ambiguities between earlier specifications and the reference code.

The graph based formalization captures the essential structural properties of SCB and supports an analysis of time space tradeoff resistance under the pebbling model. Although the current bounds are not yet as complete as those available for some established memory hard functions, the sequential block generation and full transcript dependence suggest that SCB imposes significant penalties on adversaries that attempt to reduce memory. The sponge based security assumptions used for cSHAKE, SHAKE, and SHA3 align with standard analyses of Keccak and support the functional correctness and unpredictability properties required for secure key derivation.

The performance evaluation highlights the distinctive latency bound behavior of SCB. The construction scales linearly with memory size and CPU iterations and exhibits a cost profile that reduces the performance advantage of parallel hardware. This makes SCB

a viable candidate for password hashing and memory hard key derivation on platforms where side channel resistance and hardware cost asymmetry are primary concerns.
Several open questions remain, particularly with regard to formal tradeoff bounds, hardware focused evaluation, and parameter optimization across device classes. These questions provide opportunities for further research and for strengthening the theoretical foundations of SCB. Overall, the design combines simplicity, deterministic behavior, and a clear memory hardness structure, making SCB a meaningful contribution to the ongoing study of memory hard function design.

References

1. Biryukov, A., Dinu, D., and Khovratovich, D. *Argon2: The Memory-Hard Function for Password Hashing and Other Applications.* PHC Final Report, 2015.
2. Boneh, D., Corrigan-Gibbs, H., and Schechter, S. *Balloon Hashing: A Memory-Hard Function Providing Provable Protection Against Sequential Attacks.* EuroS&P 2016.
3. Percival, C. *Stronger Key Derivation via Sequential Memory-Hard Functions.* scrypt Specification, 2009.
4. Bertoni, G., Daemen, J., Peeters, M., and Van Assche, G. *The Keccak Reference.* NIST SHA-3 Submission, 2011.
5. Underhill, J. G. *SCB Specification.* Quantum Resistant Cryptographic Solutions Corporation, 2025. Available at: https://www.qrcscorp.ca/documents/scb_specification.pdf.
6. Underhill, J. G. *SCB Reference Implementation (C Source Code).* Quantum Resistant Cryptographic Solutions Corporation, 2025. Available at: https://github.com/QRCS-CORP/QSC.