The Sgàil Cryptographic Hash Function

Peter Maxwell
peter@allicient.co.uk

January 2009 : v0.4

Abstract

Sgàil is a new hash algorithm based upon the Merkle-Damgård construction operating with a 2048-bit chaining state and a compression function that accepts 4096-bit message blocks. Sgàil can produce digest lengths of 224-bits, 256-bits, 384-bits, 512-bits, 768-bits, 1024-bits, 1536-bits and 2048-bits. Sgàil is both amenable to established analytical techniques to verify a lower security bound while incorporating additional features such as keyed diffusion to improve the security properties of the hash.

dedicated to Katharine, in gratitude for her compassion, support and insight

Contents

1 Preliminaries **1**
- 1.1 Pre-amble . . . 1
- 1.2 Generic Notation & Conventions . . . 1
  - 1.2.1 Definitions . . . 1
  - 1.2.2 Variable Representations . . . 2
  - 1.2.3 Variable & Constant Notation . . . 2
  - 1.2.4 Function Notation . . . 4
  - 1.2.5 Operators . . . 5
  - 1.2.6 Endianness . . . 5
- 1.3 Finite Fields in Sgàil . . . 5
  - 1.3.1 Finite Field Definitions . . . 5
  - 1.3.2 Finite Field Addition . . . 6
  - 1.3.3 Finite Field Multiplication . . . 6
  - 1.3.4 Finite Field Multiplicative Inverses . . . 6
- 1.4 Maximum Distance Separable Codes, Matrices & Cauchy Matrices 7
  - 1.4.1 Linear Codes, The Singleton Bound & MDS Matrices . . 7
  - 1.4.2 Cauchy Matrices as MDS Matrices . . . 7

2 Design Objectives & Constraints **8**
- 2.1 High-level Objectives . . . 8
- 2.2 First Pre-image Resistance . . . 8
- 2.3 Collision Resistance . . . 8
- 2.4 Second Pre-image Resistance . . . 8
- 2.5 Protection Against Length Extension & Message Input Block Manipulation . . . 8
- 2.6 Bijectivity . . . 9
- 2.7 Non-degeneracy . . . 9
- 2.8 Parallelisation Opportunities . . . 9
- 2.9 Randomimity . . . 10
- 2.10 Resistance to Standard Differential & Linear Cryptanalysis . . . 10
- 2.11 Local & Global Intra-block Properties . . . 10
- 2.12 Optimal & Non-linear Intra-round Diffusion . . . 11
- 2.13 Keyed Diffusion . . . 11
- 2.14 Large State Size . . . 11
- 2.15 Position Dependence . . . 11
- 2.16 Non-commutivity & Non-associativity . . . 11
- 2.17 Performance . . . 12

3 Algorithm **12**
- 3.1 Overview . . . 12
- 3.2 Substitution Box Construction & Minibox Generation . . . 13
- 3.3 Cauchy (MDS) Matrices . . . 14
  - 3.3.1 Round Constants . . . 14
- 3.4 General Structure . . . 14
  - 3.4.1 Word, Row, Quadrant & Byte Numbering . . . 14
- 3.5 Low-level Transform Layers . . . 15
  - 3.5.1 ε - The Permutation Matrix Generation Function . . . 15
  - 3.5.2 τ - The Permutation Transform . . . 16

3.5.3 ψ_8 - The 64-bit Adapted MDS Transform 17
3.5.4 ψ_{16} - The 128-bit Adapted MDS Transform 18
3.5.5 θ_8 - The 64-bit Adapted MDS Layer 18
3.5.6 θ_{16} - The 128-bit Adapted MDS Layer 19
3.5.7 ς - Two Word Pseudo-Hadamard Transform 19
3.5.8 ϕ_α - The First PHT Layer 19
3.5.9 ϕ_β - The Second PHT Layer 20
3.5.10 γ - The Quadrant Diffusion Transform 21
3.5.11 φ_0 - The First Quadrant Diffusion Layer 22
3.5.12 φ_1 - The Second Quadrant Diffusion Layer 22
3.5.13 φ_2 - The Third Quadrant Diffusion Layer 22
3.5.14 φ_3 - The Fourth Quadrant Diffusion Layer 23
3.5.15 μ - The Preliminary Key Processing Function 23
3.5.16 π_p - The Pair Block Principle Key Extraction 24
3.5.17 π_s - The Single Block Principle Key Extraction 25
3.5.18 κ_4 - The x4 Round Key Extract Function 25
3.5.19 κ_2 - The x2 Round Key Extract Function 26
3.5.20 κ_{pre} - The Pre-Whitening Round Key Extract Function . 28
3.5.21 κ_{post} - The Post-Whitening Round Key Extract Function 29
3.5.22 $\varrho_{224}, \varrho_{256}, \varrho_{384}, \varrho_{512}, \varrho_{768}, \varrho_{1024}, \varrho_{1536}, \varrho_{2048}$ - The Finalise Layer 29
3.6 High-level Transform Layers 30
3.6.1 $\Gamma_{pre}, \Gamma_{post}$ - The Pre and Post Permutation Function Groups 30
3.6.2 Υ - The Main Round Function Group 30
3.6.3 Ψ - The Compression Function 30
3.6.4 Σ - The Full Algorithm 32

4 Design Rationale **32**
4.1 High-level Priorities 32
4.2 Target Platform Priorities 32
4.3 Trade-offs 33
4.4 Tunable Parameters 34

5 Performance and Implementation **34**
5.1 General Performance 34
5.2 NIST Reference Platform 34
5.3 Hardware 35
5.4 Embedded 35
5.5 Parallelisation 35

6 Security Analysis **36**
6.1 Resistance to Standard Linear and Differential Cryptanalysis . . 36
6.2 Orthogonality 36
6.3 Complexity Argument 36

7 Appendices **36**
7.1 Constants 36
7.2 Cauchy Matrices 37
7.3 Round Constants 38

1 Preliminaries

1.1 Pre-amble

Sgàil [1] is essentially based on the Merkle-Damgård construction, built around a substitution-permutation network type cipher and the Davies-Mayer compression function. The algorithm operates on 4096-bit (512 byte) input blocks for all but possibly the final input block (which may be zero padded to 4096-bits or 2048-bits). The internal state is always 2048-bits (256 bytes) and the algorithm can produce digest lengths of 224-bits, 256-bits, 384-bits, 512-bits, 1024-bits, 1536-bits and 2048-bits.

The design of the SPN cipher at the heart of Sgàil takes a lot of inspiration from Whirlpool [Rijmen and Barreto(2001)]. However the design is divergent on a number of aspects. Firstly, Sgàil makes use of keyed diffusion, in that critical elements of the diffusion of the cipher are sensitively dependent on the key (input message blocks). Secondly, Sgàil incorporates a far more involved diffusion layer utilising exclusive-or, modular addition and rotation operations. While the diffusion layer in Whirlpool (and AES for that matter) is entirely linear over F_2, the diffusion layer in Sgàil has a small amount of non-linearity due to the implicit use of the multiplicative operation over F_2 arising from the modular addition. It is felt that an entirely linear diffusion layer over F_2 could be paving the way for easier algebraic attacks, and that the mixing of operations from different fields may defeat these types of attack while at the same time not changing the security analysis.

1.2 Generic Notation & Conventions

Some generic notation and conventions used throughout this document are now defined.

1.2.1 Definitions

The following general definitions that shall be used throughout this document:

- "state array" shall denote a 256 byte element array containing the state of the compression function or the intermediate chaining state;
- "state buffer" shall be synonymous with "state array";
- "state matrix" shall be synonymous with "state array" but to be considered as a 16 x 16 matrix of elements of F_{2^8};
- "input message block" shall be a 4096-bit (512 byte) or 2048-bit (256 byte) block of plain text message;
- "secret key" is a 256-bit (32 byte) user specified key for use in MAC constructions and the like, it is set to zero if used as drop in replacement for SHA family;

[1] Sgàil is a Scottish Gaelic word meaning shadow or veil

- "serial number" is a 64-bit user specified number which may be useful in certain instances, it is set to zero if used as a drop in replacement for SHA family;
- "block count high, block count low" is a 128-bit counter used for the current 2048-bit block of input which is stored as two 64-bit words (normally, the input block is considered as a 4096-bit block, however the counter counts on 2048-bit blocks);
- "final block bit count" this is always set to zero unless it is the last block, in which case it represents the actual number of bits in the message block before it is zero padded up to 2048-bits or 4096-bits;
- "preliminary key" is a 512-bit (64 byte) key which is independent of the data but depends on the block counter, serial, secret key and final block bit count;
- "principle key" is a 2048-bit master key derived from the input message block and preliminary key(s), it is used to generate the round keys;
- "extracted key" is a round key (which may be a pre or post whitening key) dervied from the principle key, preliminary key and round number using fast and simple operations;
- "permutation matrix" in the context of this document is a 256 element matrix containing a permutation of the numbers 0...255;
- "layer" in the context of this document means an operation that generally applies to the whole of the state matrix at once.

1.2.2 Variable Representations

In this document, a single variable may in general represent either a bit quantity, a byte quantity, a binary word of 64-bits in length, or a 16 x 16 matrix of bytes. A byte should be considered as an element of the field F_{2^8}, see sec 1.3 for more information on how F_{2^8} is used in Sgàil.

1.2.3 Variable & Constant Notation

There are some general conventions used for notation throughout this document:

- fields, groups or rings shall be denoted with caps, for example F_{2^8} or an arbitrary ring R.
- in general, sets shall be denoted with bold caps, for example $\mathcal{S}, \mathcal{T}, \mathcal{R}$;
- specifically the set of all n by n matrices with elements from F_{2^8} shall be denoted as $\mathcal{M}_{n\times n}$;
- the set of arrays of with n elements that contain as each of their elements exactly one of the sequence $(0, 1, \ldots, n-1)$ (or more succinctly, a permutation) shall be denoted as $\mathcal{R}_n$ (note: this is not the same as the mathematical definition of a permutation matrix, which is something completely different);

- the set of all 64 bit words shall be denoted as $\mathcal{W}_8$ and should be considered as the set of 8-tuples of F_{2^8}, $\mathcal{W}_8 = F_{2^8}^8$;
- matrices will be denoted in bold caps, for example we could quickly define a matrix $\mathbf{M} \in \mathcal{M}_{16,16}$;
- the state matrix for block index i shall be denoted as $\mathbf{S}_i$;
- the input message block for block index i shall be denoted as $\mathbf{D}_i$, usually $\mathbf{D}_i \in \mathcal{M}_{16,16}^2$ with $\mathbf{D}_{l,i}$ indicating the left 2048-bit half and $\mathbf{D}_{r,i}$ indicating the right 2048-bit half, however the last block may be $\mathbf{D}_i \in \mathcal{M}_{16,16}$ - the difference is usually silently ignored unless required to avoid any additional notation complexities;
- the principle key that is used for state matrix block i shall be denoted $\mathbf{K}_i$;
- an extracted key that is used for state matrix block i, and round j shall be denoted $\mathbf{E}_{i,j}$;
- an extracted key that is used for state matrix block i, and for either the pre or post permutation or whitening shall be denoted either $\mathbf{E}_{i,pre}$, $\mathbf{E}_{i,post}$;
- a preliminary key consists of eight 64-bits words and shall be written as $P = \langle \bar{p_0}, \bar{p_1}, \bar{p_2}, \bar{p_3}, \bar{p_4}, \bar{p_5}, \bar{p_6}, \bar{p_7} \rangle$;
- there are three preliminary keys defined: left, right and combined - P_l, P_r, P_c
- the matrix that is derived for permutating state matrix block i shall be denoted $\mathbf{R}_i \in \mathcal{R}_{256}$;
- the Cauchy or MDS matrices shall be denoted $\mathbf{C}_8$ for the 8×8 matrix, and $\mathbf{C}_{16}$ for the 16×16 size matrix;
- individual variables or elements will be donated with lower case letters, for example $f \in F$ may denote an element of some field F;
- variables representing byte elements, or more appropriately elements of F_{2^8}, shall be written with no accent, e.g. $a \in F_{2^8}$;
- variables representing word quantities shall be accented with a bar above, e.g. $\bar{w} \in \mathcal{W}_8$;
- variables representing bits shall be written as $\hat{a}, \hat{b}, \hat{c} \in F_2$;
- vectors shall be written in bold lower case letters, for example if we define a set of n-tuples of the field F as F^n, then $\mathbf{u} \in F^n$ would be a vector with elements of F of length n;
- when the element of a vector need to be written explicitly, angled brackets are used, e.g., $\mathbf{u} = \langle a, b, c, d, e, f \rangle$;
- later, the notion of a quadrant shall be used and denoted as $\mathbf{Q}_i, i \in 0, 1, 2, 3$ which represents a column vector of eight 64-bit words;
- each 2048-bit matrix is composed of four quadrants, the functions $\vartheta_0(\mathbf{M})$, $\vartheta_1(\mathbf{M})$, $\vartheta_2(\mathbf{M})$, $\vartheta_3(\mathbf{M})$ return the respective quadrants from the matrix $\mathbf{M}$;

- matrix elements can be referenced with subscripts, however the subscripts are **zero based**, for example $m_{i,j}$ could denote the $(i+1), (j+1)^{th}$ element of the matrix $\mathbf{M}$, or using square bracket notation, e.g. $\mathbf{M}[i][j]$ would also denote the $(i+1), (j+1)^{th}$ element;
- the i^{th} element of a vector may be denoted with square bracket notation, e.g. $\mathbf{u}[4]$ would denote the **fifth** element of the vector $\mathbf{u}$ (remember, it is zero based);
- matrices may be considered as a one-dimensional array and referenced using zero based square bracket notation, for example given a matrix $\mathbf{M}_{16,16}$ and an element of that matrix $\mathbf{M}[3][5]$, then it could be written as $\mathbf{M}[(3 * 16) + 5] = \mathbf{M}[53]$;
- where appropriate, the sub-bytes of a word may referenced using square bracket notation as if the word were an array of bytes, endianness is assumed to be preserved in the manner as described in sec 1.2.6;
- the following specific variables are defined, b_h is the 64-bit high word of the 128-bit block counter, b_l is the 64-bit low word with the block counter corresponding to the number of 2048-bit blocks processed (not 4096-bit blocks as there needs to be a preliminary key processed for each 2048-bit block);
- u is the 64-bit serial number;
- $\bar{t}_0, \bar{t}_1, \bar{t}_2, \bar{t}_3$ are the four 64-bit secret key words of $\mathbf{T}$, the 256-bit secret key.

Note : A reference to an element of a matrix, array or sub-byte of a word shall be **zero based** - which is not the way that normal notation is written. The reason this has been done is that matrix and array references are used heavily throughout this document and the C source code, so to preserve notation between the two it was thought more beneficial than using normal 'one' based notation here and 'zero' based in the source code.

1.2.4 Function Notation

We shall define all intra-round transforms, functions or layers with uncapitalised Greek letters, and inter-round transforms or functions with capitalised Greek letters. Sub-scripts will denote any additional details, relevant constants or in certain instances parameters (to aid readability) under which that function operates. For example:

- $\Psi_{\mathbf{K}} : \mathcal{M}_{16,16} \rightarrow \mathcal{M}_{16,16}$, may represent the state update under key $\mathbf{K}$;
- $\phi_\alpha : \mathcal{M}_{16,16} \rightarrow \mathcal{M}_{16,16}$, would usually represent some form of intra-round transformation, say a PHT layer.

Although ξ is used to represent the function invoked by a S-Box, the notation will be used as if it were synonymous with the S-Box itself.

1.2.5 Operators

The definition of some basic operators that shall be used in this document:

- $\oplus$ is to be construed as the standard bitwise "exclusive-or" operation, or additive operator in the field F_2 with the natural extension to binary vectors and words of arbitrary length;
- $\cdot$ is to be construed as the standard bitwise "and" operation, or multiplicative operator in the field F_2 with the natural extension to binary words of arbitrary length;
- $\neg$ is to be construed as the standard bitwise "not" operation or bit complementation;
- $+$ is to be construed as the standard arithmetic modular addition of two 64-bit words;
- $\lll$ shall define bitwise left rotate of the 64-bit word;
- $\ggg$ shall define bitwise right rotate of the 64-bit word;
- $s[i]$ where s is an array of byte elements shall denote the $(i+1)^{th}$ element (zero based indexing).

1.2.6 Endianness

Endianness is 64-bit word "little-endian" throughout the algorithm.

1.3 Finite Fields in Sgàil

1.3.1 Finite Field Definitions

A finite field is a field of finite order. Finite fields are also named Galois fields after the French mathematician Évariste Galois. The order of a finite field is always a prime number or prime power, with each finite field of a specific prime power being unique up to isomorphism. Finite fields are written F_p or $GF(p)$ for a field of order p, and F_{p^n} or $GF(p^n)$ for a field of order p^n. The p^n must not be evaluted as it is important to know what p is, i.e. F_{2^8} should not be written as F_{256}.

The characteristic of a field F_{p^n} is p. The characteristic of a finite field has several important consequences, not least that in a field of characteristic p, $(x+y)^p = (x^p + y^p)$.

Although a finite field of a specific order is unique up to isomorphism, the representation and identification of elements is important when performing actual calculations.

Consider $F_p[X]$, a polynomial with co-efficients in F_p and let $q \in F_p[X]$ be an irreducible polynomial of degree n, then $F_p[X]/(q)$ forms a field which is really just F_{p^n}. In Sgàil the field F_{2^8} is used and the representation is defined by $F_2[X]/(q)$ where $q = X^8 + X^4 + X^3 + X + 1$, and the co-efficients

of $F_p[X]$ correspond naturally to the binary bits of a byte value - i.e., the polynomial $a_7X^7 + a_6X^6 + \ldots + a_2X^2 + a_1X + a_0$ equates to the byte $a = \langle \hat{a}_7, \hat{a}_6, \hat{a}_5, \hat{a}_4, \hat{a}_3, \hat{a}_2, \hat{a}_1, \hat{a}_0 \rangle$.

In Sgàil, F_{2^8} is used to generate the Cauchy matrices that are used as the MDS matrices, and for performing calculations with the MDS matrices.

1.3.2 Finite Field Addition

If we consider the elements of F_{2^8} as elements of $F_2[X]$, then arithmetic operations in F_{2^8} are easily defined.

Addition is just addition of the two polynomials in $F_2[X]$, where the co-efficients are summed for each degree - in the field F_2 ofcourse. The possible co-efficients in F_2 are just 0 and 1, with $0 + 0 = 0, 0 + 1 = 1, 1 + 0 = 1, 1 + 1 = 0$ and $0 \centerdot 0 = 0, 0 \centerdot 1 = 0, 1 \centerdot 0 = 0.1 \centerdot 1 = 1$.

For example, $a, b \in F_2[X]$, let $a = X^6 + X^5 + X^4 + 1$ and $b = X^7 + X^4 + X$, then with co-efficients explicitly shown:

$$a + b = 1.X^7 + 1.X^6 + 1.X^5 + (1 + 1).X^4 + 1.X + 1.1 \quad (1)$$

$$= X^7 + X^6 + X^5 + X + 1 \quad (2)$$

Bitwise, this is just the familiar exclusive-or operation, $\oplus$. The above example would look like $a + b = 01110001 \oplus 10010010 = 11100011$

1.3.3 Finite Field Multiplication

Multiplication of two elements of F_{2^8} is similar to addition, except that this time the two polynomials are multiplied together modulo an irreducible polynomial, which we shall call the reduction polynomial.

Given $a, b, r \in F_2[X]$, with a, b as above and $r = X^8 + X^4 + X^3 + X + 1$, then

$$a \centerdot b = X^{13} + X^{12} + X^{11} + X^7 + X^{10} + X^9 + X^8 + X^4 + X^7 + X^6 + X^5 + X \quad (3)$$

$$= X^{13} + X^{12} + X^{11} + X^{10} + X^9 + X^8 + X^6 + X^5 + X^4 + X \quad (4)$$

Now do polynomial long division by the reduction polynomial and keep the remainder as the result:

$$X^{13}+X^{12}+X^{11}+X^{10}+X^9+X^8+X^6+X^5+X^4+X(\mathrm{mod}X^8+X^4+X^3+X+1) \quad (5)$$

$$= X^3 + X^2 + X + 1 \quad (6)$$

1.3.4 Finite Field Multiplicative Inverses

For an element $a \in F_{2^8}, a \neq 0$, the multiplicative inverse $b \in F_{2^8}$ is the unique element such that $a \centerdot b = 1$. The inverse of the zero element is conveniently defined to be the zero element. Computationally, the inverse can be found by exhaustive search, the extended Euclidean algorithm or by taking logarithms and performing a subtraction.

1.4 Maximum Distance Separable Codes, Matrices & Cauchy Matrices

1.4.1 Linear Codes, The Singleton Bound & MDS Matrices

Let F be a field, then we denote F^n as the set of n-tuples of elements of F. Naturally, F^n forms a vector space over F.

We define the hamming weight of a vector $\omega_w(\mathbf{u}), \mathbf{u} \in F^n$ as the number of non-zero elements of $\mathbf{u}$. The definition of hamming distance between two vectors $\mathbf{u}$ and $\mathbf{v}$ is $\omega_d(\mathbf{u}, \mathbf{v}) = \omega_w(\mathbf{u}-\mathbf{v})$. The hamming distance function $\omega_d : F^n \times F^n \rightarrow \mathbb{N}$ forms a metric on F^n.

The definitions to follow are obtained from [Rijmen and Barreto(2001)], as it is the most succinct explanation of the terms I have seen to date.

A linear $[n, k, d]$ code $\mathcal{C}$ over F is a k-dimensional subspace of F^n, where the hamming distance between any two distinct vectors is at least d. All linear codes obey the Singleton bound, $d \leq n - k + 1$. If the upper bound is met, $d = n - k + 1$, then the code is said to be "maximum distance separable".

For a given $[n, k, d]$ linear code $\mathcal{C}$, a generator matrix in the form $\mathbf{G}_{k,n}$ is a matrix with rows which form a basis for the code; or in other words, the span of $\mathbf{G}$ forms the vectorspace $\mathcal{C}$. The generator matrix is in "reduced echelon form" if it can be written as a partition matrix $\mathbf{G} = [\mathbf{I}_{k,k}|\mathbf{M}_{k,n-k}]$ where, $\mathbf{I}$ is (in this case at least) the identity matrix. The arbitrary matrix $\mathbf{M}$ is termed "maximal distance separable" (or MDS) iff every square submatrix of $\mathbf{M}$ is non-singular. Or in other terms the matrix $\mathbf{M}$ is MDS iff every square submatrix has a non-zero determinant (which is probably more useful when considering Cauchy Matrices).

1.4.2 Cauchy Matrices as MDS Matrices

For a field F and two sequences, $\mathbf{x} = (x_0, x_1, x_2, x_3, \ldots, x_{n-1}), x_i \in F$ and $\mathbf{y} = (y_0, y_1, y_2, y_3, \ldots, y_{m-1}), y_j \in F$, we define a matrix $\mathbf{C}_{n,m}$ with entries defined by the following equation:

$$c_{i,j} = (x_i - y_j)^{-1}, (0 \leq i \leq (n-1), 0 \leq j \leq (m-1)) \tag{7}$$

where

$$\begin{aligned} x_i &\neq x_j \quad \forall (0 \leq i \leq (n-1), 0 \leq j \leq (n-1), i \neq j) \\ y_i &\neq y_j \quad \forall (0 \leq i \leq (m-1), 0 \leq j \leq (m-1), i \neq j) \\ x_i &\neq y_j \quad \forall (0 \leq i \leq (n-1), 0 \leq j \leq (m-1)) \end{aligned}$$

A matrix $\mathbf{C}$ of this form is termed a "Cauchy Matrix" and has some useful properties. Firstly, all Cauchy Matrices are non-singluar and any submatrix of a Cauchy Matrix is also a Cauchy Matrix; hence any sub-matrix of a Cauchy matrix is non-singluar, the result of which means that any Cauchy Matrix is

automatically an MDS matrix. A good reference for the use of Cauchy matrices in cryptography is [Youssef et al.(1997)Youssef, Mister, and Tavares].

The MDS matrices used in Sgàil are actually Cauchy Matrices and hence guaranteed to be MDS. The MDS matrix is a critical security element, as such generating the matrix in a consistent manner increases confidence that no weaknesses were deliberately introduced. Furthermore, although the implementation cost in embedded devices may be greater, the hamming weight over F_{2^8} of the entries is more balanced than is likely to be achieved in a hand optimised matrix.

2 Design Objectives & Constraints

2.1 High-level Objectives

2.2 First Pre-image Resistance

This is the first of the NIST core requirements for a cryptographic hash algorithm (these are also the minimal criteria one would expect). Given $\mathbf{H} = \Sigma(\mathbf{m})$, where Σ is the hash function, $\mathbf{m}$ is an arbitrary message and $\mathbf{H}$ is the hash result of length n bits, then it should be infeasible to compute $\mathbf{m}$ given only $\mathbf{H}$. More accurately, it should take at least 2^n work to recover $\mathbf{m}$ from $\mathbf{H}$.

2.3 Collision Resistance

This is the second of the NIST core requirements, and is the requirement which is generally of most practical importance. It should be infeasible to find to messages, $\mathbf{m}_a, \mathbf{m}_b$ with hash values $\mathbf{H}_a = \Sigma(\mathbf{m}_a), \mathbf{H}_b = \Sigma(\mathbf{m}_b)$ respectively, such that $\mathbf{H}_a = \mathbf{H}_b$. More accurately, it should take at least $2^{n/2}$ work to find such $\mathbf{m}_1$ and $\mathbf{m}_2$.

2.4 Second Pre-image Resistance

This is the third NIST core requirements. Given a fixed $\mathbf{m}_a$, it should be infeasible to find another message $\mathbf{m}_b$ such that $\mathbf{H}_a = \mathbf{H}_b$ given $\mathbf{H}_a = \Sigma(\mathbf{m}_a), \mathbf{H}_b = \Sigma(\mathbf{m}_b)$. Any hash algorithm which is susceptible to a second pre-image attack is consequentially susceptible to a collision attack (the only difference here is the first $\mathbf{m}$ is fixed so there is only one degree of freedom to use). It should take at least 2^n work to find such a $\mathbf{m}_b$ for fixed $\mathbf{m}_a$.

2.5 Protection Against Length Extension & Message Input Block Manipulation

There are several requirements here:

- given $\mathbf{H} = \Sigma(\mathbf{m})$, it should not be possible to compute $\mathbf{H}' = \Sigma(\mathbf{m}|\mathbf{m}')$ for an arbitrary $\mathbf{m}'$ with less effort than the equivalent first pre-image attack on $\mathbf{H} = \Sigma(\mathbf{m})$;

- given $\mathbf{H} = \Sigma(\mathbf{m})$ and the length of $\mathbf{m}$, $l = \|\mathbf{m}\|$, then it should not be possible to compute $\mathbf{H}' = \Sigma(\mathbf{m}|\mathbf{m}')$ for arbitrary $\mathbf{m}'$ with any work less than $2^{(l-1)}$ full hash computations;
- given $\mathbf{H}_a = \Sigma(\mathbf{m}_a)$, $\mathbf{H}_b = \Sigma(\mathbf{m}_b)$, with respective lengths $l_a = \|\mathbf{m}_a\|, l_b = \|\mathbf{m}_b\|$ and $\mathbf{H}_a = \mathbf{H}_b, l_a \neq l_b$, then for arbitrary message $\mathbf{m}'$ the identity $\mathbf{H'}_a \neq \mathbf{H'}_b$ where $\mathbf{H'}_a = \Sigma(\mathbf{m}_a|\mathbf{m}')$, $\mathbf{H}_b = \Sigma(\mathbf{m}_b|\mathbf{m}')$ should hold with overwhelming probability, i.e. it should take $2^{(n-1)}$ work to find a $\mathbf{m}'$ such that the identity does not hold where n is the digest length of the hash function.

2.6 Bijectivity

The notion of bijectivity is essential to the creation of a secure hash algorithm. There are two requirements specified in this respect:

- given a fixed message input block $\mathbf{D}_i$ and a state matrix $\mathbf{S}_i$, then the application of the compression function $\mathbf{S}_{i+1} = \Psi_{\mathbf{D}_i}(\mathbf{S}_i)$ should be bijective with respect to $\mathbf{S}_i$;
- given either the left or right half of the message input block (2048-bit) $\mathbf{D}_{l,i}$, or $\mathbf{D}_{r,i}$, and a fixed state matrix $\mathbf{S}_i$, then the application of the compression function $\mathbf{S}_{i+1} = \Psi_{\mathbf{D}_i}(\mathbf{S}_i)$ should be bijective with respect to $\mathbf{D}_{l,i}$, or $\mathbf{D}_{r,i}$.

2.7 Non-degeneracy

Cryptographic hash functions are used in a very wide spectrum of applications including pseudo-random functions or pseudo-random number generators. For these purposes, it is desirable that if the output of the hash algorithm is repeatedly reused as an input then no short length cycles (degenerate cycles) be produced. Assuming that given an arbitrary input $\mathbf{m}$ then compute the sequence:

$$(\mathbf{H}_0 = \Sigma(\mathbf{m}), \mathbf{H}_1 = \Sigma(\mathbf{H}_0), \mathbf{H}_2 = \Sigma(\mathbf{H}_1), \ldots, \mathbf{H}_i = \Sigma(\mathbf{H}_{i-1}))$$

we should optimally have $\mathbf{H}_0 \neq \mathbf{H}_1 \neq \mathbf{H}_2 \neq \cdots \neq \mathbf{H}_{i-1}$ for $i < 2^n$ where n is the digest length, viz maximal cycle length. Note that if the previous condition holds, then it not only holds for the $\mathbf{M}$ we chose, but for any message.

Unfortunately, in practice it is generally very difficult to ensure (or at least prove) maximal cycle length. However, it is worthwhile to note that it is a *necessary* condition (although not *sufficient*) that the hash function Σ be a bijective mapping to achieve maximal cycle length. If the hash function is not bijective, then it is assured that its cycle length is non-maximal.

2.8 Parallelisation Opportunities

The trend in modern CPU design seems to be moving towards the use of multiple cores - it is common to see dual or quad core processors today. For a more extreme example and possible insight into where we're going [TheRegister(Sept2006)] as regards CPU design may be of interest.

The use of multiple cores and multiple CPUs in the same server or desktop implies that any software that wants to utilise the full processing power available must be able to run in a parallel manner. As such the candidate algorithm should offer opportunities of parallelism at both the intra-block calculations and the inter-block arrangements.

2.9 Randomimity

In a somewhat ambiguous notion, the output of the hash algorithm should pass any standard randomimity tests. More formally:

- given the situation where the output digest of the hash function is repeatedly reused as the input, then the resulting stream of outputs should satisfy standard randomimity tests;
- given the same situation where the output digest of the hash function is repeatedly reused as the input, then the entropy preserved in any digest result should be identical to the previous digest;
- given an arbitrary series of non-repeating input messages with and the resultant hash for each message, the stream of output digests should satisfy standard randomimity tests.

Disregarding the problems inherent in the use of the definition to follow; the algorithm should be indistinguishable from a random oracle.

2.10 Resistance to Standard Differential & Linear Cryptanalysis

The compression function Ψ should be demonstrably resistant to standard differential and linear cryptanalysis.

2.11 Local & Global Intra-block Properties

The output of the hash function should be unbiased in the number of ones or zeros in the output. More precisely, chose k arbitrary messages, each of arbitrary length, $(\mathbf{m}_0, \mathbf{m}_1, \ldots \mathbf{m}_{k-1})$, with outputs

$$(\mathbf{H}_0 = \Sigma(\mathbf{m}_0), \mathbf{H}_1 = \Sigma(\mathbf{m}_1), \ldots, \mathbf{H}_{k-1} = \Sigma(\mathbf{m}_{k-1}))$$

Then the following limit should hold:

$$\lim_{k\to\infty} \frac{1}{k} \sum_{i=0}^{k-1} \omega(\mathbf{H}_i) \cong \frac{n}{2} \tag{8}$$

where n is the digest size. The above limit should also hold where the evaluation of the hamming weight $\omega()$ is restricted to only a subset of the bits of $\mathbf{H}_i$ (with the $\frac{n}{2}$ term being altered accordingly).

2.12 Optimal & Non-linear Intra-round Diffusion

Joan Daemen and Vincent Rijmen in the "wide trail design" emphasised the use of an optimal diffusion layer. All diffusion layers utilised in the algorithm should be either optimal or near optimal with respect to the definition of "branch number" (whether that be linear or non-linear branch number).

Additionally, the diffusion layer should not be completely linear over F_2. There is no requirement on the algebraic degree other than it must not be totally linear or affine.

2.13 Keyed Diffusion

Algorithms such as Two-Fish [Schneier et al.(1998)Schneier, Kelsey, Whiting, Wagner, Hall, and Ferguson] use key-dependent substitution boxes. In a similar sense, the algorithm should employ key-dependent diffusion in a manner such that a small change in the input message block causes a major change in the arrangement of the diffusion.

2.14 Large State Size

The state size should be made as large as possible given the technical capabilities of the target systems the algorithm is to be implemented on. There are several reasons for using a large state size:

- it makes the process of finding collisions in the compression function far more difficult, hence it renders most generic attacks on the chaining scheme inapplicable;
- it hinders the ability of any potential advances in algebraic attacks by (hopefully exponentially) increasing the number of equations necessary to solve;
- assuming an optimal or near optimal diffusion layer, then the compression function's resistance against differential or linear cryptanalysis increases polynomially with the size of the state matrix given a fixed number of rounds in the compression function.

2.15 Position Dependence

The input to the hash function, or indeed intra-round operations, should be position dependant. More specifically, if identical inputs are passed to a function or transform but in a different position, be that a different block or a different row, then the output should be different. This stipulation makes it somewhat harder for an attacker to exploit any regularities or predictability in local behaviour if it is found.

2.16 Non-commutivity & Non-associativity

It is desirable that as many of the operations as possible are non-commutative and/or non-associative.

2.17 Performance

The algorithm should provide acceptable performance on the target platforms. By acceptable, it is considered that in normal conditions it may not be the fastest hash algorithm available but it should be of the same order of magnitude in performance to any other algorithm offering equivalent security. However, it should also offer means of improving its performance through the ability to parallelise the implementation. The design should also pay attention to preferential methods of achieving a desired result as not to adversely impede any specific target implementation (e.g. implementing S-Boxes in a manner amenable to fast hardware implementation).

3 Algorithm

3.1 Overview

Each element of Sgàil will be defined in isolation and then put together in the definition of the compression function and full algorithm.

For each state update, the algorithm takes the current 2048-bit state and encrypts it using a substitution-permutation network block cipher supplying a 4096-bit (possibly 2048-bit if it is the last input block) of message input as the key. Mayer-Davies chaining is then applied by exclusive-or'ing the original state onto the new state.

The key schedule is somewhat involved; unless it is the last block the compression function accepts a 4096-bit message input, which can be considered as two 2048-bit halves. Firstly, three data independent 512-bit preliminary keys are generated: the left, right and combined preliminary keys. The the left and right preliminary keys are each derived from a user supplied 256-bit secret key, two 64-bit block counters and a user supplied serial number; if it is the last input message block that is being processed, then the number of message bits used in the last block (normally this is set to zero) is also included in the preliminary key derivation. The block counter used for each preliminary key indicates the number of 2048-bit input message blocks processed, so the block counter for the right preliminary key is just the block counter for the left preliminary key plus one. The combined preliminary key is the xor sum of the left and right preliminary keys.

The left and right input message blocks are then processed along with the left and right preliminary keys to produce a single 2048-bit principle key which is essentially what all round keys are extracted from. There are several round keys, or extracted keys, used; a pre-whitening extracted key, a post-whitening extracted key and one extracted key for each round. The combined preliminary key is used in the round key extraction.

The compression function uses a key based permutation near the beginning and end of the update. The actual permutation used is derived from the principle key. The compression function is initiated by first exclusive-or'ing the pre whitening key onto the state. It then applies the key based permutation to reorder the arrangement of the matrix. The state matrix is then passed through

a S-Box and a 64-bit MDS transform applied to each of the 64-bit words in turn. The main rounds are then performed. Each round consists of applying a parallel list of pseudo-Hadamard transforms to the state matrix, followed by a special diffusion primative, more pseudo-Hadamard transforms, exclusive-or'ing the round key, and finally applying a S-Box and parallel 128-bit MDS transform. In the standard implementation there are usually four rounds for 512-bit or less digest sizes, six rounds for 768-bit or 1024-bit digests, and eight rounds for 1536-bit and 2048-bit digests. The compression function is finalised by applying the key based permutation and then exclusive-or'ing the post whitening key and the original state matrix.

The initial state IV is a copy of AES's S-Box (ξ_g). The initial IV must not be user definable.

3.2 Substitution Box Construction & Minibox Generation

In Sgàil, there is one generic 8x8bit S-Box which is used for the key derived permutation generation, namely the ε transform, and for initialisation values. This generic S-Box is can be written as a function ξ_g. The actual S-Box lookup table for ξ_g is the S-Box as used in AES as this has excellent properties and is well known. Efficiency in hardware for this S-Box is not a significant consideration as the ε generation will probably have to use RAM anyway so there is already a performance hit.

There are another 24 main 8x8bit S-Boxes, eight of which are used in ψ_8 and sixteen in ψ_{16}. These S-Boxes are designed with hardware implementation in mind as they are used in the core of the algorithm. Each S-Box was generated from 16 4x4bit miniboxes which have in turn been randomly generated.

There is little point in duplicating the full algorithm used here, but in the development source code included in the submission package, the code used to generate and test the miniboxes and S-Boxes is included in full. However it is necessary to make a number of assertions:

- the miniboxes were generated using a pseudo-random generator in a consistent and repeatable fashion;
- only the miniboxes which had a $DP_{max} = 1/4$, $LP_{max} = 0$, full nonlinearity and no fixed points were chosen;
- 16 miniboxes were generated to create one S-Box, resulting in the necessity to generate 384 separate miniboxes;
- the S-Boxes are generated by taking a loop from 0...255, and for each iteration to take the loop value and pass the high nibble through the first minibox and the low nibble through the second and then rotate left by 2 bits, repeat until the 16 miniboxes are used up and then go onto the next iteration in the loop;
- the worst DP_{max} for the S-Boxes is a single S-Box at 14/256, most are at 10/256 or 12/256;
- the LP_{max} of the S-Boxes is always 4/256 or less;

- the nonlinearity of the S-Boxes is always 100 or more (as measured as the distance from a linear or affine function);
- the maximum number of fixed points in any of the S-Boxes is 1;
- full stats are included in the supporting documentation directory.

The S-Boxes will generally be written as functions in this document labelled $\xi_0, \xi_1, \ldots, \xi_{23}$. The S-Boxes are included explicitly in the source code in the tables file.

3.3 Cauchy (MDS) Matrices

The $\mathbf{C}_8$ matrix is created from the two sequences:

$$(1, 2, 3, 5, 8, 13, 21, 34) \tag{9}$$

$$(222, 235, 243, 248, 251, 253, 254, 255) \tag{10}$$

The first sequence is the Fibonacci sequence starting from $(1, 2)$, the second sequence is just the first sequence subtracted from 256 (sorted in ascending order).

The $\mathbf{C}_{16}$ matrix is created from the two sequences:

$$(0, 3, 8, 15, 24, 35, 48, 63, 80, 109, 120, 143, 168, 195, 224, 255) \tag{11}$$

$$(1, 31, 60, 87, 112, 135, 146, 175, 192, 207, 220, 231, 240, 247, 252, 254) \tag{12}$$

The first sequence is generated from $n^2 - 1$, and the second sequence is the first sequence sorted in ascending order and subtracted from 256 (except for the last entry which was tweaked to make the Cauchy matrix valid).

The matrices are explicitly included in the appendix.

3.3.1 Round Constants

There is an array of "rounds constants" defined as the hexadecimal representation of the fractional digits of Pi. These are used in certain places throughout the algorithm, mainly to ensure that it is difficult for an attacker to artificially lower the hamming weight in any part of the key schedule. The exact representation is defined in the appendices.

3.4 General Structure

3.4.1 Word, Row, Quadrant & Byte Numbering

Almost all of the operations in Sgàil work on matrices in $\mathcal{M}_{16,16}$. These matrices can be conceptually considered as thirty-two words $(w_0 \ldots w_{31})$, or sixteen rows numbered $(r_0 \ldots r_{15})$, or four quadrants denoted (q_0, q_1, q_2, q_3). This segmentation - conceptually at least - renders the algorithm more easily amenable

to security analysis as a lot of the algorithm is designed on the basis of ensuring good properties locally (which is fast) and then applying appropriate mixing across these boundaries.

	$\mathbf{q_0}$	$\mathbf{q_1}$
$\mathbf{r_0}$	$\mathbf{w_0}$: *7, 6, 5, 4, 3, 2, 1, 0*	$\mathbf{w_1}$: *15, 14, 13, 12, 11, 10, 9, 8*
$\mathbf{r_1}$	$\mathbf{w_2}$: *23, 22, 21, 20, 19, 18, 17, 16*	$\mathbf{w_3}$: *31, 30, 29, 28, 27, 26, 25, 24*
$\mathbf{r_2}$	$\mathbf{w_4}$: *39, 38, 37, 36, 35, 34, 33, 32*	$\mathbf{w_5}$: *47, 46, 45, 44, 43, 42, 41, 40*
$\mathbf{r_3}$	$\mathbf{w_6}$: *55, 54, 53, 52, 51, 50, 49, 48*	$\mathbf{w_7}$: *63, 62, 61, 60 ,59, 58, 57, 56*
$\mathbf{r_4}$	$\mathbf{w_8}$: *71, 70, 69, 68, 67, 66, 65, 64*	$\mathbf{w_9}$: *79, 78, 77, 76, 75, 74, 73, 72*
$\mathbf{r_5}$	$\mathbf{w_{10}}$: *87, 86, 85, 84, 83, 82, 81, 80*	$\mathbf{w_{11}}$: *95, 94, 93, 92, 91, 90, 89, 88*
$\mathbf{r_6}$	$\mathbf{w_{12}}$: *103, 102, 101, 100, 99, 98, 97, 96*	$\mathbf{w_{13}}$: *111, 110, 109, 108, 107, 106, 105, 104*
$\mathbf{r_7}$	$\mathbf{w_{14}}$: *119, 118, 117, 116, 115, 114, 113, 112*	$\mathbf{w_{15}}$: *127, 126, 125, 124, 123, 122, 121, 120*
	$\mathbf{q_2}$	$\mathbf{q_3}$
$\mathbf{r_8}$	$\mathbf{w_{16}}$: *135, 134, 133, 132, 131, 130, 129, 128*	$\mathbf{w_{17}}$: *143, 142, 141, 140, 139, 138, 137, 136*
$\mathbf{r_9}$	$\mathbf{w_{18}}$: *151, 150, 149, 148, 147, 146, 145, 144*	$\mathbf{w_{19}}$: *159, 158, 157, 156, 155, 154, 153, 152*
$\mathbf{r_{10}}$	$\mathbf{w_{20}}$: *167, 166, 165, 164, 163, 162, 161, 160*	$\mathbf{w_{21}}$: *175, 174, 173, 172, 171, 170, 169, 168*
$\mathbf{r_{11}}$	$\mathbf{w_{22}}$: *183, 182, 181, 180, 179, 178, 177, 176*	$\mathbf{w_{23}}$: *191, 190, 189, 188, 187, 186, 185, 184*
$\mathbf{r_{12}}$	$\mathbf{w_{24}}$: *199, 198, 197, 196 ,195, 194, 193, 192*	$\mathbf{w_{25}}$: *207, 206, 205, 204, 203, 202, 201, 200*
$\mathbf{r_{13}}$	$\mathbf{w_{26}}$: *215, 214, 213, 212, 211, 210, 209, 208*	$\mathbf{w_{27}}$: *223, 222, 221, 220, 219, 218, 217, 216*
$\mathbf{r_{14}}$	$\mathbf{w_{28}}$: *231, 230, 229, 228, 227, 226, 225, 224*	$\mathbf{w_{29}}$: *239, 238, 237, 236, 235, 234, 233, 232*
$\mathbf{r_{15}}$	$\mathbf{w_{30}}$: *247, 246, 245, 244, 243, 242, 241, 240*	$\mathbf{w_{31}}$: *255, 254, 253, 252, 251, 250, 249, 248*

3.5 Low-level Transform Layers

3.5.1 ε - The Permutation Matrix Generation Function

The function $\varepsilon : \mathcal{R}_{256} \times \mathcal{M}_{16,16} \times \mathbb{Z} \rightarrow \mathcal{R}_{256}$ is the core of the principle of keyed diffusion in Sgàil- it creates a keyed method of permutating the order of the entries in the state matrix. [2]

The definition of ε is somewhat involved and best explained in prose rather than terse notation.

Let the initial supplied array $\mathbf{R}_0$ be a permutation - i.e., each of the values $0, 1, 2, \ldots, 245, 255$ is expressed exactly once in the array. Let $\mathbf{K}_{256}$ be an arbitrary array of byte values which will act as the key in this scenario.

[2]Note: When then term "permutation matrix" is used in this document - it is not the same as the permutation matrices found in Mathematics

The objective is to alter the array $\mathbf{R}_0$ according to the key $\mathbf{K}$ and still maintain $\mathbf{R}$ as a permutation. The result should be a permutation and should display sensitive dependence on the input key $\mathbf{K}$ and $\mathbf{R}_0$. The resulting permutation can then be used to translate the elements of the state matrix.

This modification of the permutation matrix is accomplished by taking inspiration from the RC4 stream cipher, with a few tweaks. It is probably best to display the pseudo-code used and then explain any relevant points.

Let $i, j \in F_{2^8}$, i is assumed to be zero and j is set to a seeding parameter to start with. Let $\mathbf{S}_{256}$ temporarily stand for ξ_g, $\mathbf{R}_0$ be the start permutation array (which is always a copy of ξ_g), $\mathbf{K}_{256}$ be an arbitrary byte array to be used as the key. Then the ε can be described with the following pseudo-code:

```
1: j = S[ j ];
2: for i = 0 ... 255 {
3: j = j + R[ j ] + K[ i ];
4: j = S[ j ];
5: swap( R[ S[ i ] ], R[ j ] );
6: }
```

The counter i makes sure we use every key value and that we permutate every entry in $\mathbf{R}$. When we reach the swap however we use the S-Box entry at i rather than i directly - both methods touch every entry of $\mathbf{R}$, however the S-Box lookup avoids the usual problem of the initial swap (and cryptographically weaker) results being clustered around the start of the array. The benefit of distributing the initial results throughout $\mathbf{R}$ is more apparent when combined with the 64 bit MDS layer that immediately follows; if the initial values were to remain at the beginning of the array then the first 64 bit word would be somewhat more predicable than required, by distributing these initial results any weakness is still present but is dispersed throughout the $\mathbf{R}$ matrix.

The counter j plays the rôle of introducing non-linearity into the permutation. Each iteration is in effect a non-linear transform on the previous value of j, a value of $\mathbf{R}$, and a key entry. The j counter is a parameter set by the caller. The transform is highly sensitive to the initial value of j.

Usually the permutation array $\mathbf{R}_0$ is initialised with a copy of the ξ_g S-Box - it would be just as easy to start with the sequence $(0, 1, 2, \ldots, 245, 255)$ or for that matter any permutation, however it is still assumed to be a parameter. Using the S-Box as $\mathbf{R}_0$ is easy, in C its just a simple `memcpy` and it also avoids any regularities that using an easier sequence would entail.

3.5.2 τ - The Permutation Transform

The function $\tau : \mathcal{M}_{16,16} \times \mathcal{R}_{256} \rightarrow \mathcal{M}_{16,16}$ takes a matrix $\mathbf{M}$ and a permutation matrix $\mathbf{R}$ and rearranges the elements of $\mathbf{M}$ according to the permutation $\mathbf{R}$.

τ can be defined as follows:

$$\mathbf{M} = \tau(\mathbf{M}_0) \Leftrightarrow \mathbf{M}[i] = \mathbf{R}[\mathbf{M}_0[i]], 0 \leq i < 256 \tag{13}$$

It would be possible to permutate the state matrix with a key directly, however by using an intermediate matrix $\mathbf{R}$ as generated by ε several advantages are reaslised: (i) the permutation array $\mathbf{R}$ once calculated can be efficiently reused later, (ii) the method reduces the chance of creating degenerate, or low entropy permutations of the state matrix, and (iii) it is more elegant when used with the S-Box and MDS transform immediately afterwards as defined in sec 3.5.3.

3.5.3 ψ_8 - The 64-bit Adapted MDS Transform

The canonical method of utilising MDS matrices in cryptography to provide intra-word diffusion is the simple pre-multiplication by an MDS matrix of a column matrix representing the elements of an input word. The net effect being that each element of the input column (input word) is essentially acting as a scalar multiple of a specific column of the MDS matrix, the collection of resulting columns corresponding to each element of the initial word are then summed. As the summation is over F_{2^8}, this is just the familiar exclusive-or operation. This effect can be exploited to combine the S-Box lookups and MDS multiplication in a highly optimised manner; by pre-calculating the result for each possible entry to an S-Box and then using the resulting scalar to multiply the appropriate column of the MDS matrix and storing it in a lookup table, explicit full calculations can be avoided. When it comes to implementation, all that needs to be done is to lookup each element of the input word in the table and exclusive-or the results together. So for an n element word (which requires an n, n MDS matrix) all that is required is n table lookups and $n-1$ exclusive-or operations - which is very fast on modern processors.

The approach taken in Sgàil takes a slight diversion from this. While retaining the table lookup structure, the arrangement is not a simple pre-multiply by the MDS matrix.

The first difference is that separate S-Boxes are used, one for each element of the input word to be precise. So for the ψ_8 transform there are eight separate S-Boxes, which shall be represented as functions $\xi_0, \xi_1, \ldots, \xi_7$. The second major difference is that instead of performing an S-Box lookup on an element of the input word and then using the result to scalar multiply a column of the MDS matrix, the elements of the input word are passed through the different S-Boxes in turn so creating a new word - each element of this new word is then multiplied by the corresponding element in the column from the MDS matrix and stored as a word in the lookup table.

This can be written more formally as a function $\psi_8 : \mathcal{W}_8 \to \mathcal{W}_8$, and is defined by:

$$\begin{aligned}
\psi_8 : \mathbf{x} \mapsto \quad & diag(c_{0,0}, c_{1,0}, \ldots, c_{7,0}) \times col(\xi_0(x_0), \xi_1(x_0), \ldots, \xi_7(x_0)) \\
+ \quad & diag(c_{0,1}, c_{1,1}, \ldots, c_{7,1}) \times col(\xi_0(x_1), \xi_1(x_1), \ldots, \xi_7(x_1)) \\
+ \quad & \vdots \quad \vdots \quad \vdots \quad \vdots \\
+ \quad & diag(c_{0,7}, c_{1,7}, \ldots, c_{7,7}) \times col(\xi_0(x_7), \xi_1(x_7), \ldots, \xi_7(x_7))
\end{aligned}$$

where $c_{i,j}$ are the elements of the MDS matrix $\mathbf{C}_8$, $\mathbf{x} \in \mathcal{W}_8$, $diag()$ indicates the creation of a diagonal matrix from the entries, $col()$ indicates the column vector

of the entries, and finally $\times$ is just matrix multiplication (written in explicitly in this case to make things more readable).

The natural question is: why use this construction? Well, if an attacker has found limited ability to control the entries to the S-Boxes then setting the input to the S-Box to the appropriate value so that the result is zero would be an obvious place to start. In the original construction this effectively zeros out an entire column of the MDS calculation (one word entry in the lookup table) - which may be useful when looking for collisions, especially if the effect can be cumulative. In the ψ_8 construction, it is not possible - or at least significantly more difficult - to judiciously select the inputs of the S-Box in this manner. As the same value goes through multiple S-Boxes it is impossible to make more than one of the results zero - hence that column of the MDS (or that word in the lookup table) is not nulled out, it is only one single element of that word that is affected which is good behaviour. Whether this has any real advantages remains to be seen, but it does introduce a certain degree of orthogonality to the setup.

Once the adapted MDS transform is complete, each of the 64-bit words is also rotated and exclusive-or'ed with itself (the rotation constants for this are listed in the appendices). The only reason for doing this is to destroy any byte alignment properties (and frustratingly complicate notation and description). As the rotation is fixed and commutes with exclusive-or, it can (and is) precomputed into the MDS tables.

3.5.4 ψ_{16} - The 128-bit Adapted MDS Transform

The 128-bit MDS is defined in an equivalent manner to ψ_8. The $\psi_{16} : \mathcal{W}_{16} \rightarrow \mathcal{W}_{16}$ transform can be defined by:

$$\begin{aligned}
\psi_{16} : \mathbf{x} \mapsto \quad & diag(c_{0,0}, c_{1,0}, \ldots, c_{15,0}) \times col(\xi_8(x_0), \xi_9(x_0), \ldots, \xi_{23}(x_0)) \\
+ \quad & diag(c_{0,1}, c_{1,1}, \ldots, c_{15,1}) \times col(\xi_8(x_1), \xi_9(x_1), \ldots, \xi_{23}(x_1)) \\
+ \quad & \vdots \quad \vdots \quad \vdots \quad \vdots \\
+ \quad & diag(c_{0,15}, c_{1,15}, \ldots, c_{15,15}) \times col(\xi_8(x_{15}), \xi_9(x_{15}), \ldots, \xi_{23}(x_{15}))
\end{aligned}$$

Again each of the 64-bit words are rotated and exclusive-or'd with itself (constants defined in the appendix).

3.5.5 θ_8 - The 64-bit Adapted MDS Layer

The 64-bit adapted MDS layer is the parallel application of the 64-bit adapted MDS transform, ψ_8, to the 32 words of a matrix and can be defined as a function $\theta_8 : \mathcal{W}_8^{32} \rightarrow \mathcal{W}_8^{32}$. [3]

$$\theta_8 : \langle \mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{31} \rangle \mapsto \langle \psi_8(\mathbf{x}_0), \psi_8(\mathbf{x}_1), \ldots, \psi_8(\mathbf{x}_{31}) \rangle, (\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{31} \in \mathcal{W}_8) \tag{14}$$

[3] Note the Endianness as defined in sec 1.2.6 for translation of byte values to word values.

or equivalently,

$$\mathbf{Y} = \theta_8(\mathbf{X}) \Leftrightarrow \mathbf{y}_i = \psi_8(\mathbf{x}_i), (\mathbf{X}, \mathbf{Y} \in \mathcal{W}_8^{32}, \mathbf{x}_i, \mathbf{y} \in \mathcal{W}_8, 0 \leq i < 32) \tag{15}$$

3.5.6 θ_{16} - The 128-bit Adapted MDS Layer

The 128-bit adapted MDS transform layer is the parallel application of the 128-bit adapted MDS transform, ψ_{16}, to the 16 rows of a matrix and can be defined as a function $\theta_{16} : \mathcal{W}_{16}^{16} \rightarrow \mathcal{W}_{16}^{16}$.

$$\theta_{16} : \langle \mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{15} \rangle \mapsto \langle \psi_{16}(\mathbf{x}_0), \psi_{16}(\mathbf{x}_1), \ldots, \psi_{16}(\mathbf{x}_{15}) \rangle, (\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{15} \in \mathcal{W}_{16}) \tag{16}$$

or equivalently,

$$\mathbf{Y} = \theta_{16}(\mathbf{X}) \Leftrightarrow \mathbf{y}_i = \psi_{16}(\mathbf{x}_i), (\mathbf{X}, \mathbf{Y} \in \mathcal{W}_{16}^{16}, \mathbf{x}_i, \mathbf{y} \in \mathcal{W}_{16}, 0 \leq i < 16) \tag{17}$$

3.5.7 ς - Two Word Pseudo-Hadamard Transform

Pseudo-Hadamard transforms, or PHTs, are a fast method of providing diffusion. Although addition in $\mathbb{Z}$ is obviously linear in $\mathbb{Z}$, over F_{2^8} this is a non-linear operation (although of fairly low algebraic degree), which can quickly make algebraic attacks non-trivial.

The function $\varsigma : \mathcal{W}_8 \times \mathcal{W}_8 \rightarrow \mathcal{W}_8 \times \mathcal{W}_8$ can be defined as:

$$\varsigma : \langle \bar{u}, \bar{v} \rangle \mapsto \langle \bar{u} + \bar{v}, \bar{u} + 2 \cdot \bar{v} \rangle \tag{18}$$

The diffusion obtained by using PHTs is generally sub-optimal, but the relative execution speed of the transform compared to other more involved methods renders it an exceedingly useful operation.

3.5.8 ϕ_α - The First PHT Layer

The $\phi_\alpha : \mathcal{M}_{16,16} \rightarrow \mathcal{M}_{16,16}$ layer of PHTs is the first of two layers in Sgàil that consists of the parallel application of PHTs. In this ϕ_α layer, the words in $\mathbf{Q}_0$ are transformed with the words in $\mathbf{Q}_3$, while the words in $\mathbf{Q}_1$ are transformed with the words in $\mathbf{Q}_2$.

The complete equation list is as follows:

$\mathbf{Q}_0 \longleftrightarrow \mathbf{Q}_3$:

$$\langle \bar{w}_0, \bar{w}_{17} \rangle = \varsigma(\bar{w}_0, \bar{w}_{17}) \tag{19}$$

$$\langle \bar{w}_2, \bar{w}_{19} \rangle = \varsigma(\bar{w}_2, \bar{w}_{19}) \tag{20}$$

$$\langle \bar{w}_4, \bar{w}_{21} \rangle = \varsigma(\bar{w}_4, \bar{w}_{21}) \tag{21}$$

$$\langle \bar{w}_6, \bar{w}_{23} \rangle = \varsigma(\bar{w}_6, \bar{w}_{23}) \tag{22}$$

$$\langle \bar{w}_8, \bar{w}_{25} \rangle = \varsigma(\bar{w}_8, \bar{w}_{25}) \tag{23}$$
$$\langle \bar{w}_{10}, \bar{w}_{27} \rangle = \varsigma(\bar{w}_{10}, \bar{w}_{27}) \tag{24}$$
$$\langle \bar{w}_{12}, \bar{w}_{29} \rangle = \varsigma(\bar{w}_{12}, \bar{w}_{29}) \tag{25}$$
$$\langle \bar{w}_{14}, \bar{w}_{31} \rangle = \varsigma(\bar{w}_{14}, \bar{w}_{31}) \tag{26}$$

$\mathbf{Q}_1 \longleftrightarrow \mathbf{Q}_2$:

$$\langle \bar{w}_9, \bar{w}_{16} \rangle = \varsigma(\bar{w}_9, \bar{w}_{16}) \tag{27}$$
$$\langle \bar{w}_{11}, \bar{w}_{18} \rangle = \varsigma(\bar{w}_{11}, \bar{w}_{18}) \tag{28}$$
$$\langle \bar{w}_{13}, \bar{w}_{20} \rangle = \varsigma(\bar{w}_{13}, \bar{w}_{20}) \tag{29}$$
$$\langle \bar{w}_{15}, \bar{w}_{22} \rangle = \varsigma(\bar{w}_{15}, \bar{w}_{22}) \tag{30}$$
$$\langle \bar{w}_1, \bar{w}_{24} \rangle = \varsigma(\bar{w}_1, \bar{w}_{24}) \tag{31}$$
$$\langle \bar{w}_3, \bar{w}_{26} \rangle = \varsigma(\bar{w}_3, \bar{w}_{26}) \tag{32}$$
$$\langle \bar{w}_5, \bar{w}_{28} \rangle = \varsigma(\bar{w}_5, \bar{w}_{28}) \tag{33}$$
$$\langle \bar{w}_7, \bar{w}_{30} \rangle = \varsigma(\bar{w}_7, \bar{w}_{30}) \tag{34}$$

Which words are transformed with which is important - the ϕ_α combined with the ϕ_β are designed with the row nature of the ψ_{16} transform in mind. A change in one of the inputs to ψ_{16} results in a change across all 16 elements of that row, which in turn due to the two PHT layers will cause a change in four other words which always sit in different rows (this is all neglecting the Quadrant Diffusion layer that is executed in between the two PHT layers). So with just the ψ_{16}, ϕ_α and ϕ_β, a change in one element at the input will cause a change in $6 \cdot 8 = 48$ of the byte inputs to the next round.

3.5.9 ϕ_β - The Second PHT Layer

The $\phi_\beta : \mathcal{M}_{16,16} \rightarrow \mathcal{M}_{16,16}$ layer of PHTs is the second of two layers in Sgàil that consists of the parallel application of PHTs.

The complete equation list for the ϕ_β layer is as follows:

$$\langle \bar{w}_0, \bar{w}_3 \rangle = \varsigma(\bar{w}_0, \bar{w}_3) \tag{35}$$
$$\langle \bar{w}_2, \bar{w}_7 \rangle = \varsigma(\bar{w}_2, \bar{w}_7) \tag{36}$$
$$\langle \bar{w}_4, \bar{w}_1 \rangle = \varsigma(\bar{w}_4, \bar{w}_1) \tag{37}$$
$$\langle \bar{w}_6, \bar{w}_5 \rangle = \varsigma(\bar{w}_6, \bar{w}_5) \tag{38}$$
$$\langle \bar{w}_8, \bar{w}_{11} \rangle = \varsigma(\bar{w}_8, \bar{w}_{11}) \tag{39}$$
$$\langle \bar{w}_{10}, \bar{w}_{15} \rangle = \varsigma(\bar{w}_{10}, \bar{w}_{15}) \tag{40}$$
$$\langle \bar{w}_{12}, \bar{w}_9 \rangle = \varsigma(\bar{w}_{12}, \bar{w}_9) \tag{41}$$
$$\langle \bar{w}_{14}, \bar{w}_{13} \rangle = \varsigma(\bar{w}_{14}, \bar{w}_{13}) \tag{42}$$
$$\langle \bar{w}_{16}, \bar{w}_{19} \rangle = \varsigma(\bar{w}_{16}, \bar{w}_{19}) \tag{43}$$
$$\langle \bar{w}_{18}, \bar{w}_{23} \rangle = \varsigma(\bar{w}_{18}, \bar{w}_{23}) \tag{44}$$

$$\langle \bar{w}_{20}, \bar{w}_{17} \rangle = \varsigma(\bar{w}_{20}, \bar{w}_{17}) \tag{45}$$

$$\langle \bar{w}_{22}, \bar{w}_{21} \rangle = \varsigma(\bar{w}_{22}, \bar{w}_{21}) \tag{46}$$

$$\langle \bar{w}_{24}, \bar{w}_{27} \rangle = \varsigma(\bar{w}_{24}, \bar{w}_{27}) \tag{47}$$

$$\langle \bar{w}_{26}, \bar{w}_{31} \rangle = \varsigma(\bar{w}_{26}, \bar{w}_{31}) \tag{48}$$

$$\langle \bar{w}_{28}, \bar{w}_{25} \rangle = \varsigma(\bar{w}_{28}, \bar{w}_{25}) \tag{49}$$

$$\langle \bar{w}_{30}, \bar{w}_{29} \rangle = \varsigma(\bar{w}_{30}, \bar{w}_{29}) \tag{50}$$

3.5.10 γ - The Quadrant Diffusion Transform

The $\gamma : \mathcal{W}_8^8 \to \mathcal{W}_8^8$ transform takes a quadrant and repeatedly applies addition, exclusive-or and rotation operations to achieve a fast and near optimal level of diffusion.

For this section, words of the quadrant are labeled $\langle \bar{v}_0, \bar{v}_1, \bar{v}_2, \bar{v}_3, \bar{v}_4, \bar{v}_5, \bar{v}_6, \bar{v}_7 \rangle$ instead of the numbering used in the state matrix. We also define r_0, r_1, r_2, the three rotation constants used (see appendices for actual constants).

The function $\gamma_\alpha : \mathcal{W}_8 \times \mathcal{W}_8 \times \mathbb{Z} \to \mathcal{W}_8$ is defined by:

$$\gamma_\alpha : (\bar{x}, \bar{y}, r) \mapsto \bar{x} + ((\bar{x} \oplus \bar{y}) \lll r) \tag{51}$$

The function $\gamma_\beta : \mathcal{W}_8 \times \mathcal{W}_8 \times \mathbb{Z} \to \mathcal{W}_8$ is defined by:

$$\gamma_\beta : (\bar{x}, \bar{y}, r) \mapsto \bar{x} \oplus ((\bar{x} + \bar{y}) \lll r) \tag{52}$$

The transform γ is then defined by the sequential evaluation [4] of:

$$v_0 = \gamma_\alpha(v_0, v_7, r_0) \tag{53}$$

$$v_1 = \gamma_\alpha(v_1, v_0, r_1) \tag{54}$$

$$v_2 = \gamma_\alpha(v_2, v_1, r_2) \tag{55}$$

$$v_3 = \gamma_\alpha(v_3, v_2, r_0) \tag{56}$$

$$v_4 = \gamma_\alpha(v_4, v_3, r_1) \tag{57}$$

$$v_5 = \gamma_\alpha(v_5, v_4, r_2) \tag{58}$$

$$v_6 = \gamma_\alpha(v_6, v_5, r_0) \tag{59}$$

$$v_7 = \gamma_\alpha(v_7, v_6, r_1) \tag{60}$$

$$v_0 = \gamma_\beta(v_0, v_7, r_2) \tag{61}$$

$$v_1 = \gamma_\beta(v_1, v_0, r_0) \tag{62}$$

$$v_2 = \gamma_\beta(v_2, v_1, r_1) \tag{63}$$

$$v_3 = \gamma_\beta(v_3, v_2, r_2) \tag{64}$$

$$v_4 = \gamma_\beta(v_4, v_3, r_0) \tag{65}$$

[4] The "sequential evaluation" part is important, as each evaluation of a function relies on previous results

$$v_5 = \gamma_\beta(v_5, v_4, r_1) \tag{66}$$

$$v_6 = \gamma_\beta(v_6, v_5, r_2) \tag{67}$$

$$v_7 = \gamma_\beta(v_7, v_6, r_0) \tag{68}$$

While the γ transform was described as "near optimal", it can be stated more formally. The rotation constants were derived empirically to be optimal (under certain conditions). In the samples performed the mean byte hamming weight of the γ transform is around 63 (out of a possible 64) which is near optimal, and is lower bounded by 25. This was tested by setting the entire quadrant to zero and doing a "walking one" through each of the 512-bits and applying a reduced version of the quad diffuse at each step and calculating the hamming distance (the reduced version replaced the addition operations with exclusive-or operations to reliably determine how many bit positions were modified). This was done for each of the possible $(2^{64})^3$ combinations and the results with the highest minimal hamming distance were chosen (which was 25).

In practice these results will be a lot better as the tests were performed on a reduced version of γ and the introduction of normal non-zero data increases the diffusion provided by the addition operations.

3.5.11 φ_0 - The First Quadrant Diffusion Layer

The φ_0 applies γ to $\mathbf{Q}_0$ with rotation constants as specified in the appendices. $\mathbf{Q}_0$ is then exclusive-or'ed over each of the other three quadrants with the words of $\mathbf{Q}_0$ rotated by a different amount for each quadrant, again see appendices for the constants used.

If one considers that the application of ψ_{16} then ϕ_α which are performed immediately before φ_0, then it becomes obvious why the results of γ are exclusive-or'ed over the other quadrants. Just before γ is applied, each word of $\mathbf{Q}_0$ is dependant on at least one word of the other three quadrants. So if the γ transform is applied and has a byte hamming distance of n, then we can say that a change to one input element to ψ_{16} will induce a change in at least n elements of $\mathbf{Q}_0$ after γ is applied. When the contents of $\mathbf{Q}_0$ are exclusive-or'd over the over three quadrants then they also share this property and near optimal diffusion is achieved.

The different rotation constants for each quadrant means that the inputs to the S-Boxes is in a small sense orthogonal across the different quadrants.

3.5.12 φ_1 - The Second Quadrant Diffusion Layer

φ_1 is identical to φ_0 except γ is applied to $\mathbf{Q}_1$ and the constants are changed.

3.5.13 φ_2 - The Third Quadrant Diffusion Layer

φ_2 is identical to φ_0 except γ is applied to $\mathbf{Q}_2$ and the constants are changed.

3.5.14 φ_3 - The Fourth Quadrant Diffusion Layer

φ_3 is identical to φ_0 except γ is applied to $\mathbf{Q}_3$ and the constants are changed.

3.5.15 μ - The Preliminary Key Processing Function

The preliminary key function takes a 256-bit secret key, a 64-bit serial number, two 64-bit words representing the current block count (of processed 2048-bit input blocks) and a 64-bit word representing the number of bits used in the final block (is otherwise always zero). The results of the preliminary key are used in the principle key extraction and round key extraction.

The 256-bit secret key can be used to create a keyed hash function and the format is user defined, however if used as a direct drop-in replacement to the SHA family of algorithms it should be set to zero. The serial number is user definable, but if being used as a drop-in replacement to SHA it should be set to zero. The two words representing the block count are the 128-bit representation of the 2048-bit message block to which the preliminary key is to be used with, which means that Sgàil can hash a theoretical maximum of 2^{127} 4096-bit blocks or $4096 \cdot 2^{127}$ input bits - this theoretical limit could obviously never be reached in reality (it was however thought that it may be necessary in extreme circumstances to hash 2^{64} blocks, hence the 128-bit representation). The last word, the final bit count, is normally set to zero and is only non-zero for the last input block in which case its value should be the bit length of the last message block.

The four words of the 256-bit key $\mathbf{T}$ shall be labeled as $\bar{t}_0, \bar{t}_1, \bar{t}_2, \bar{t}_3$. The serial number shall be labeled as u. The block counter words shall be labeled as a high-word and a low-word, b_h, b_l. The final block bit count word shall be labeled as f_b.

The result of the μ function is the preliminary key vector $P_= \langle \bar{p_0}, \bar{p_1}, \bar{p_2}, \bar{p_3}, \bar{p_4}, \bar{p_5}, \bar{p_6}, \bar{p_7} \rangle$.

The function $\mu : \mathcal{W}_8^4 \times \mathcal{W}_8 \times \mathcal{W}_8 \times \mathcal{W}_8 \times \mathcal{W}_8 \rightarrow \mathcal{W}_8^8$ can be defined by the sequential evaluation of:

$$
\begin{aligned}
\bar{w} &= u \\
\bar{p_0} &= \psi_8(\bar{w}) \\
\bar{w} &= \bar{p_0} + \mathbf{T}_0 \\
\bar{p_1} &= \psi_8(\bar{w}) \\
\bar{w} &= \bar{p_1} + b_l \\
\bar{p_2} &= \psi_8(\bar{w}) \\
\bar{w} &= \bar{p_2} + \mathbf{T}_1 \\
\bar{p_3} &= \psi_8(\bar{w}) \\
\bar{w} &= \bar{p_3} + f_b \\
\bar{p_4} &= \psi_8(\bar{w}) \\
\bar{w} &= \bar{p_4} + \mathbf{T}_2 \\
\bar{p_5} &= \psi_8(\bar{w}) \\
\bar{w} &= \bar{p_5} + b_h \\
\bar{p_6} &= \psi_8(\bar{w}) \\
\bar{w} &= \bar{p_6} + \mathbf{T}_3 \\
\bar{p_7} &= \psi_8(\bar{w})
\end{aligned}
$$

$$
\begin{aligned}
\bar{p_0} &= \bar{p_0} + \bar{p_7} \\
\bar{p_1} &= \bar{p_1} + \bar{p_0} \\
\bar{p_2} &= \bar{p_2} + \bar{p_1} \\
\bar{p_3} &= \bar{p_3} + \bar{p_2} \\
\bar{p_4} &= \bar{p_4} + \bar{p_3} \\
\bar{p_5} &= \bar{p_5} + \bar{p_4} \\
\bar{p_6} &= \bar{p_6} + \bar{p_5} \\
\bar{p_7} &= \bar{p_7} + \bar{p_6}
\end{aligned}
$$

3.5.16 π_p - The Pair Block Principle Key Extraction

All input message blocks are 4096-bits and are reduced to a 2048-bit principle key with π_p. The only exception is if the finalise (last) block is being processed in which case, if there are 2048-bits or less, the block is passed directly through π_s (with the left preliminary key being used) and the result used as the principle key.

$\pi_p : \mathcal{M}_{16,16} \times \mathcal{M}_{16,16} \times \mathcal{W}_8^8 \times \mathcal{W}_8^8 \rightarrow \mathcal{M}_{16,16}$ takes the 4096-bit input block (considered as two 2048-bit blocks) and the left and right preliminary keys returning a 2048-bit principle key:

$$
\begin{aligned}
\pi_p : \mathbf{X}_l, \mathbf{X}_r, \mathbf{P}_l, \mathbf{P}_r \mapsto \quad & \pi_s(\mathbf{X}_l, \mathbf{P}_l, \varepsilon(\xi_g, \mathbf{X}_r, 0), \varepsilon(\xi_g, \mathbf{X}_l, 0)) && (69) \\
+_* \quad & \pi_s(\mathbf{X}_r, \mathbf{P}_r, \varepsilon(\xi_g, \mathbf{X}_l, 0), \varepsilon(\xi_g, \mathbf{X}_r, 0)) && (70)
\end{aligned}
$$

where $+_*$ in this case means addition of each 64-bit word in parallel of the two matrix operands.

3.5.17 π_s - The Single Block Principle Key Extraction

The function $\pi_s : \mathcal{M}_{16,16} \times \mathcal{W}_8^8 \times \mathcal{R}_{256} \times \mathcal{R}_{256} \rightarrow \mathcal{M}_{16,16}$ acts on 2048-bit blocks and is used either as part of π_p to reduce input 4096-bit blocks to a 2048-bit principle key, or on a final input block of 2048-bits or less in length to obtain a principle key.

If $\oplus_{q0}$ is taken to mean exclusive-or quadrant zero of the left operand (16,16) with the right operand (8,8), then π_s can be defined as:

$$\pi_s : \mathbf{X}, \mathbf{P}, \mathbf{R}_{pre}, \mathbf{R}_{post} \mapsto \mathbf{X} \oplus (\theta_8 \circ \tau_{\mathbf{R}_{post}})(\theta_{16}((\phi_\beta \circ \varphi_0 \circ \phi_\alpha)((\theta_8 \circ \tau_{\mathbf{R}_{pre}})(\mathbf{X}) \oplus_{q0} \mathbf{P})) \oplus_{q0} \mathbf{P}) \tag{71}$$

π_s has a natural extension to two or three rounds, but is not used. It is however included in the development source code included in the submission package.

3.5.18 κ_4 - The x4 Round Key Extract Function

The $\kappa_4 : \mathcal{W}_8^8 \times \mathcal{W}_8^8 \times \mathbb{Z} \rightarrow \mathcal{M}_{16,16}$ round key extraction takes as its input one quadrant of the principle key, a preliminary key and a round index. It then does a very light transform and returns a full round key.

If $\mathbf{E}$ is the resultant extracted round key with words $\bar{e}_0, \bar{e}_1, \ldots, \bar{e}_{31}$, the words of the supplied principle key quadrant $\mathbf{K}$ are labeled $\bar{k}_0, \bar{k}_1, \ldots, \bar{k}_7$ and the words of the preliminary key P are labeled $\bar{p_0}, \bar{p_1}, \ldots, \bar{p_7}$. The round constants are used here and will be labeled as $\bar{c}_0, \bar{c}_1, \ldots, \bar{c}_{63}$. There are three rotation constants used (and listed in the appendix), which shall be labeled r_0, r_1, r_2. The round index shall be denoted i. Then κ_4 is defined by the sequential evaluation of (note the indexing of principle key quadrant here is relative to the quadrant as passed, not the global principle key):

$$\begin{aligned}
\bar{e}_0 &= \bar{k}_0 \oplus \bar{p_0} \oplus \bar{c}_{i(\text{mod } 64)} \\
\bar{e}_2 &= \bar{k}_1 \oplus \bar{p_1} \oplus \bar{c}_{(i+1)(\text{mod } 64)} \\
\bar{e}_4 &= \bar{k}_2 \oplus \bar{p_2} \oplus \bar{c}_{(i+2)(\text{mod } 64)} \\
\bar{e}_6 &= \bar{k}_3 \oplus \bar{p_3} \oplus \bar{c}_{(i+3)(\text{mod } 64)} \\
\bar{e}_8 &= \bar{k}_4 \oplus \bar{p_4} \oplus \bar{c}_{(i+4)(\text{mod } 64)} \\
\bar{e}_{10} &= \bar{k}_5 \oplus \bar{p_5} \oplus \bar{c}_{(i+5)(\text{mod } 64)} \\
\bar{e}_{12} &= \bar{k}_6 \oplus \bar{p_6} \oplus \bar{c}_{(i+6)(\text{mod } 64)} \\
\bar{e}_{14} &= \bar{k}_7 \oplus \bar{p_7} \oplus \bar{c}_{(i+7)(\text{mod } 64)}
\end{aligned}$$

$$
\begin{aligned}
\bar{e}_1 &= \bar{e}_2 \lll ((r_0 + i)(\text{mod } 64)) \\
\bar{e}_3 &= \bar{e}_0 \lll ((r_0 + i)(\text{ mod } 64)) \\
\bar{e}_5 &= \bar{e}_6 \lll ((r_0 + i)(\text{mod } 64)) \\
\bar{e}_7 &= \bar{e}_4 \lll ((r_0 + i)(\text{mod } 64)) \\
\bar{e}_9 &= \bar{e}_{10} \lll ((r_0 + i)(\text{ mod } 64)) \\
\bar{e}_{11} &= \bar{e}_8 \lll ((r_0 + i)(\text{mod } 64)) \\
\bar{e}_{13} &= \bar{e}_{14} \lll ((r_0 + i)(\text{mod } 64)) \\
\bar{e}_{15} &= \bar{e}_{12} \lll ((r_0 + i)(\text{mod } 64\text{c}))
\end{aligned}
$$

$$
\begin{aligned}
\bar{e}_{16} &= \bar{e}_6 \lll ((r_1 + i)(\text{mod } 64)) \\
\bar{e}_{18} &= \bar{e}_4 \lll ((r_1 + i)(\text{mod } 64)) \\
\bar{e}_{20} &= \bar{e}_2 \lll ((r_1 + i)(\text{mod } 64)) \\
\bar{e}_{22} &= \bar{e}_0 \lll ((r_1 + i)(\text{mod } 64)) \\
\bar{e}_{24} &= \bar{e}_{14} \lll ((r_1 + i)(\text{mod } 64)) \\
\bar{e}_{26} &= \bar{e}_{12} \lll ((r_1 + i)(\text{mod } 64)) \\
\bar{e}_{28} &= \bar{e}_{10} \lll ((r_1 + i)(\text{mod } 64)) \\
\bar{e}_{30} &= \bar{e}_8 \lll ((r_1 + i)(\text{mod } 64))
\end{aligned}
$$

$$
\begin{aligned}
\bar{e}_{17} &= \bar{e}_{14} \lll ((r_2 + i)(\text{mod } 64)) \\
\bar{e}_{19} &= \bar{e}_{12} \lll ((r_2 + i)(\text{mod } 64)) \\
\bar{e}_{21} &= \bar{e}_{10} \lll ((r_2 + i)(\text{mod } 64)) \\
\bar{e}_{23} &= \bar{e}_8 \lll ((r_2 + i)(\text{mod } 64)) \\
\bar{e}_{25} &= \bar{e}_6 \lll ((r_2 + i)(\text{mod } 64)) \\
\bar{e}_{27} &= \bar{e}_4 \lll ((r_2 + i)(\text{mod } 64)) \\
\bar{e}_{29} &= \bar{e}_2 \lll ((r_2 + i)(\text{mod } 64)) \\
\bar{e}_{31} &= \bar{e}_0 \lll ((r_2 + i)(\text{mod } 64))
\end{aligned}
$$

3.5.19 κ_2 - The x2 Round Key Extract Function

The $\kappa_2 : \mathcal{W}_8^8 \times \mathcal{W}_8^8 \times \mathcal{W}_8^8 \times \mathbb{Z} \to \mathcal{M}_{16,16}$ round key extraction takes as its input two quadrants of the principle key, a preliminary key and a round index. It then does a very light transform and returns a full round key. The first principle key quadrant passed shall be labeled $\bar{k}_{a,x}$, and the second as $\bar{k}_{b,x}$ where x is the index relative to the passed quadrant.

κ_2 is defined in a similar manner to κ_4:

$$
\begin{aligned}
\bar{e}_0 &= \bar{k}_{a,0} \oplus \bar{p_0} \oplus \bar{c}_{i(\mathrm{mod}\ 64)} \\
\bar{e}_2 &= \bar{k}_{a,1} \oplus \bar{p_1} \oplus \bar{c}_{(i+1)(\mathrm{mod}\ 64)} \\
\bar{e}_4 &= \bar{k}_{a,2} \oplus \bar{p_2} \oplus \bar{c}_{(i+2)(\mathrm{mod}\ 64)} \\
\bar{e}_6 &= \bar{k}_{a,3} \oplus \bar{p_3} \oplus \bar{c}_{(i+3)(\mathrm{mod}\ 64)} \\
\bar{e}_8 &= \bar{k}_{a,4} \oplus \bar{p_4} \oplus \bar{c}_{(i+4)(\mathrm{mod}\ 64)} \\
\bar{e}_{10} &= \bar{k}_{a,5} \oplus \bar{p_5} \oplus \bar{c}_{(i+5)(\mathrm{mod}\ 64)} \\
\bar{e}_{12} &= \bar{k}_{a,6} \oplus \bar{p_6} \oplus \bar{c}_{(i+6)(\mathrm{mod}\ 64)} \\
\bar{e}_{14} &= \bar{k}_{a,7} \oplus \bar{p_7} \oplus \bar{c}_{(i+7)(\mathrm{mod}\ 64)}
\end{aligned}
$$

$$
\begin{aligned}
\bar{e}_1 &= \bar{k}_{b,0} \lll ((r_0 + i) \mathrm{mod}\ 64)) \\
\bar{e}_3 &= \bar{k}_{b,1} \lll ((r_0 + i) \mathrm{mod}\ 64)) \\
\bar{e}_5 &= \bar{k}_{b,2} \lll ((r_0 + i) \mathrm{mod}\ 64)) \\
\bar{e}_7 &= \bar{k}_{b,3} \lll ((r_0 + i) \mathrm{mod}\ 64)) \\
\bar{e}_9 &= \bar{k}_{b,4} \lll ((r_0 + i) \mathrm{mod}\ 64)) \\
\bar{e}_{11} &= \bar{k}_{b,5} \lll ((r_0 + i) \mathrm{mod}\ 64)) \\
\bar{e}_{13} &= \bar{k}_{b,6} \lll ((r_0 + i) \mathrm{mod}\ 64)) \\
\bar{e}_{15} &= \bar{k}_{b,7} \lll ((r_0 + i) \mathrm{mod}\ 64))
\end{aligned}
$$

$$
\begin{aligned}
\bar{e}_{16} &= \bar{e}_2 \lll ((r_1 + i) \mathrm{mod}\ 64)) \\
\bar{e}_{18} &= \bar{e}_0 \lll ((r_1 + i) \mathrm{mod}\ 64)) \\
\bar{e}_{20} &= \bar{e}_6 \lll ((r_1 + i) \mathrm{mod}\ 64)) \\
\bar{e}_{22} &= \bar{e}_4 \lll ((r_1 + i) \mathrm{mod}\ 64)) \\
\bar{e}_{24} &= \bar{e}_{10} \lll ((r_1 + i) \mathrm{mod}\ 64)) \\
\bar{e}_{26} &= \bar{e}_8 \lll ((r_1 + i) \mathrm{mod}\ 64)) \\
\bar{e}_{28} &= \bar{e}_{14} \lll ((r_1 + i) \mathrm{mod}\ 64)) \\
\bar{e}_{30} &= \bar{e}_2 \lll ((r_1 + i) \mathrm{mod}\ 64))
\end{aligned}
$$

$$
\begin{aligned}
\bar{e}_{17} &= \bar{e}_7 \lll ((r_2 + i) \mathrm{mod}\ 64)) \\
\bar{e}_{19} &= \bar{e}_5 \lll ((r_2 + i) \mathrm{mod}\ 64)) \\
\bar{e}_{21} &= \bar{e}_3 \lll ((r_2 + i) \mathrm{mod}\ 64)) \\
\bar{e}_{23} &= \bar{e}_1 \lll ((r_2 + i) \mathrm{mod}\ 64)) \\
\bar{e}_{25} &= \bar{e}_{15} \lll ((r_2 + i) \mathrm{mod}\ 64)) \\
\bar{e}_{27} &= \bar{e}_{13} \lll ((r_2 + i) \mathrm{mod}\ 64)) \\
\bar{e}_{29} &= \bar{e}_{11} \lll ((r_2 + i) \mathrm{mod}\ 64)) \\
\bar{e}_{31} &= \bar{e}_9 \lll ((r_2 + i) \mathrm{mod}\ 64))
\end{aligned}
$$

3.5.20 κ_{pre} - The Pre-Whitening Round Key Extract Function

The $\kappa_{pre} : \mathcal{W}_8^{32} \times \mathcal{W}_8^8 \rightarrow \mathcal{M}_{16,16}$ round key extraction takes as its input the entire principle key and a preliminary key. It then does a very light transform and returns a full round key.

Similar to the other key extract functions (note that the normal indexing is used here for the principle key, whereas in κ_4 and κ_2 the indexing was just incremental) :

$$\begin{aligned}
\bar{e}_0 &= \bar{k}_0 \oplus \bar{p_0} \\
\bar{e}_2 &= \bar{k}_2 \oplus \bar{p_1} \\
\bar{e}_4 &= \bar{k}_4 \oplus \bar{p_2} \\
\bar{e}_6 &= \bar{k}_6 \oplus \bar{p_3} \\
\bar{e}_8 &= \bar{k}_8 \oplus \bar{p_4} \\
\bar{e}_{10} &= \bar{k}_{10} \oplus \bar{p_5} \\
\bar{e}_{12} &= \bar{k}_{12} \oplus \bar{p_6} \\
\bar{e}_{14} &= \bar{k}_{14} \oplus \bar{p_7}
\end{aligned}$$

$$\begin{aligned}
\bar{e}_1 &= \bar{k}_1 \oplus \bar{p_7} \\
\bar{e}_3 &= \bar{k}_3 \oplus \bar{p_6} \\
\bar{e}_5 &= \bar{k}_5 \oplus \bar{p_5} \\
\bar{e}_7 &= \bar{k}_7 \oplus \bar{p_4} \\
\bar{e}_9 &= \bar{k}_9 \oplus \bar{p_3} \\
\bar{e}_{11} &= \bar{k}_{11} \oplus \bar{p_2} \\
\bar{e}_{13} &= \bar{k}_{13} \oplus \bar{p_1} \\
\bar{e}_{15} &= \bar{k}_{15} \oplus \bar{p_0}
\end{aligned}$$

$$\begin{aligned}
\bar{e}_{16} &= \bar{k}_{16} \\
\bar{e}_{18} &= \bar{k}_{18} \\
\bar{e}_{20} &= \bar{k}_{20} \\
\bar{e}_{22} &= \bar{k}_{22} \\
\bar{e}_{24} &= \bar{k}_{24} \\
\bar{e}_{26} &= \bar{k}_{26} \\
\bar{e}_{28} &= \bar{k}_{28} \\
\bar{e}_{30} &= \bar{k}_{30}
\end{aligned}$$

$$
\begin{aligned}
\bar{e}_{17} &= \bar{k}_{17} \\
\bar{e}_{19} &= \bar{k}_{19} \\
\bar{e}_{21} &= \bar{k}_{21} \\
\bar{e}_{23} &= \bar{k}_{23} \\
\bar{e}_{25} &= \bar{k}_{25} \\
\bar{e}_{27} &= \bar{k}_{27} \\
\bar{e}_{29} &= \bar{k}_{29} \\
\bar{e}_{31} &= \bar{k}_{31}
\end{aligned}
$$

Then there is a single PHT layer,

$$\mathbf{E} = \phi_{\alpha}(\mathbf{E}_0) \tag{72}$$

3.5.21 κ_{post} - The Post-Whitening Round Key Extract Function

This is exactly the same as κ_{post} , except the ϕ_{α} is changed for ϕ_{β}.

3.5.22 $\varrho_{224}, \varrho_{256}, \varrho_{384}, \varrho_{512}, \varrho_{768}, \varrho_{1024}, \varrho_{1536}, \varrho_{2048}$ - The Finalise Layer

The finalise layer is used on the final state matrix to convert it to the hash digest value. If the required digest length is 512-bits or less, a reduction operation is first performed. Continuing to use the word numbering as defined in 3.4.1 then the reduction operation is defined as:

$$
\begin{aligned}
\bar{w}_0 &= \bar{w}_1 \oplus \bar{w}_{16} \oplus \bar{w}_{17} \\
\bar{w}_2 &= \bar{w}_3 \oplus \bar{w}_{18} \oplus \bar{w}_{19} \\
\bar{w}_4 &= \bar{w}_5 \oplus \bar{w}_{20} \oplus \bar{w}_{21} \\
\bar{w}_6 &= \bar{w}_7 \oplus \bar{w}_{22} \oplus \bar{w}_{23} \\
\bar{w}_8 &= \bar{w}_9 \oplus \bar{w}_{24} \oplus \bar{w}_{25} \\
\bar{w}_{10} &= \bar{w}_{11} \oplus \bar{w}_{26} \oplus \bar{w}_{27} \\
\bar{w}_{12} &= \bar{w}_{13} \oplus \bar{w}_{28} \oplus \bar{w}_{29} \\
\bar{w}_{14} &= \bar{w}_{15} \oplus \bar{w}_{30} \oplus \bar{w}_{31}
\end{aligned}
$$

If the required digest length is more than 512-bits then the reduction operation is omitted.

The final step is to truncate the state array to the desired length.

3.6 High-level Transform Layers

3.6.1 $\Gamma_{pre}, \Gamma_{post}$ - The Pre and Post Permutation Function Groups

The function $\Gamma_{pre} : \mathcal{M}_{16,16} \times \mathcal{R}_{256} \times \mathcal{M}_{16,16} \rightarrow \mathcal{M}$ is performed at the start of the compression function - it applies the pre-whitening key, permutates the state matrix and performs a 64-bit adapted MDS transform layer. It can be defined as:

$$\Gamma_{pre} : \mathbf{M}, \mathbf{R}, \mathbf{E} \mapsto (\theta_8 \circ \tau)((\mathbf{M} \oplus \mathbf{E}), \mathbf{R}) \tag{73}$$

In a similar way we define the post permutation Γ_{post} by:

$$\Gamma_{post} : \mathbf{M}, \mathbf{R}, \mathbf{E} \mapsto (\theta_8 \circ \tau)(\mathbf{M}, \mathbf{R}) \oplus \mathbf{E} \tag{74}$$

3.6.2 Υ - The Main Round Function Group

The main round function $\Upsilon : \mathcal{M}_{16,16} \times \mathcal{M}_{16,16} \times \{0, 1, 2, 3\} \rightarrow \mathcal{M}_{16,16}$ is the main round update and takes the state matrix, a key and an index for which quad diffuse layer to use and returns the updated state matrix. The operation consists of applying a PHT layer, followed by the quad diffuse layer, then a PHT layer, exclusive-or in the key and then pass through the adapted 128-bit MDS layer. It can be defined as:

$$\Upsilon : \mathbf{M}, \mathbf{E}_r, r_q \mapsto \theta_{16}((\phi_\beta \circ \varphi_{r_q} \circ \phi_\alpha)(\mathbf{M}) \oplus \mathbf{E}_r) \tag{75}$$

3.6.3 Ψ - The Compression Function

The main compression function $\Psi : \mathcal{M}_{16,16} \times \mathcal{M}_{16,16}^2 \times \mathbb{Z}$ will now be defined.

If $\mathbf{D}_i$ is the current input message block (4096-bit) then we define $\mathbf{D}_{l,r}$ as the 2048-bit left hand half and $\mathbf{D}_{r,i}$ as the 2048-bit right hand half. If the message block is the last block and it is less then 2048-bits in length, then we just write $\mathbf{D}_i$. Let b_h, b_l be the 64-bit high and low word of the left hand side block counter (i.e the first of the two 2048-bit input message block halves). Then $b_h, b_l + 1$ is the block counter values for the right hand side (i.e the second of the two 2048-bit halves). Let f_b represent the bit count of the final block (otherwise it is zero), and u be the serial number. The secret key $\mathbf{T}$ words are denoted $\bar{t}_0, \bar{t}_1, \bar{t}_2, \bar{t}_3$.

To apply the compression function at input block i, the keys must first be derived:

The preliminary keys are defined as $P_{l,i} = \mu(\mathbf{T}, u, b_h, b_l, f_b)$, $P_{r,i} = \mu(\mathbf{T}, u, b_h, b_l + 1, f_b)$ and $P_{c,i} = P_{l,i} \oplus P_{r,i}$. Where $P_{l,i}$ is the left preliminary key, $P_{r,i}$ is the right preliminary key and $P_{c,i}$ is the combined preliminary key.

If it is not a finalise block or if it is a final block with more than 2048-bits then principle key $\mathbf{K}_i = \pi_p(\mathbf{D}_{l,i}, \mathbf{D}_{r,i}, P_{l,i}, P_{r,i})$, else $\mathbf{K}_i = \pi_s(\mathbf{D}_i, P_{l,i})$ where the message data is padded with zero bits to make up to a 2048-bit block (there

is no need to do the normal MD length padding as the final bit length is already used in the preliminary key processing).

The number of rounds is always of the form $4n$ or $4n+2$ with $n \geq 1$. If the current round number setting is of the form (4n) rounds, r is taken to be the round number and $0 \leq m < n$ then generate the round keys as:

$$\begin{array}{l} \left\{ \ \mathbf{E}_{i,pre} = \kappa_{pre}(\mathbf{K}_i, P_{c,i}) \ \right\} \\ m = 0, 1, \ldots, n-1 \quad \left\{ \begin{array}{ll} \mathbf{E}_{i,r} = \kappa_4(\vartheta_0(\mathbf{K}_i), P_{c,i}, r) & r = 4m \\ \mathbf{E}_{i,r} = \kappa_4(\vartheta_1(\mathbf{K}_i), P_{c,i}, r) & r = 4m+1 \\ \mathbf{E}_{i,r} = \kappa_4(\vartheta_2(\mathbf{K}_i), P_{c,i}, r) & r = 4m+2 \\ \mathbf{E}_{i,r} = \kappa_4(\vartheta_3(\mathbf{K}_i), P_{c,i}, r) & r = 4m+3 \end{array} \right\} \\ \left\{ \ \mathbf{E}_{i,post} = \kappa_{post}(\mathbf{K}_i, P_{c,i}) \ \right\} \end{array}$$

If the current number of rounds being used is of the form (4n + 2), r is taken to be the round number and $0 \leq m < n$ then generate the round keys as:

$$\begin{array}{l} \left\{ \ \mathbf{E}_{i,pre} = \kappa_{pre}(\mathbf{K}_i, P_{c,i}) \ \right\} \\ \left\{ \ \mathbf{E}_{i,0} = \kappa_2(\vartheta_0(\mathbf{K}_i), \vartheta_1(\mathbf{K}_i), P_{c,i}, r) \quad r = 0 \ \right\} \\ m = 0, 1, \ldots, n-1 \quad \left\{ \begin{array}{ll} \mathbf{E}_{i,r} = \kappa_4(\vartheta_0(\mathbf{K}_i), P_{c,i}, r) & r = 4m+1 \\ \mathbf{E}_{i,r} = \kappa_4(\vartheta_1(\mathbf{K}_i), P_{c,i}, r) & r = 4m+2 \\ \mathbf{E}_{i,r} = \kappa_4(\vartheta_2(\mathbf{K}_i), P_{c,i}, r) & r = 4m+3 \\ \mathbf{E}_{i,r} = \kappa_4(\vartheta_3(\mathbf{K}_i), P_{c,i}, r) & r = 4m+4 \end{array} \right\} \\ \left\{ \ \mathbf{E}_{i,4n-1} = \kappa_2(\vartheta_2(\mathbf{K}_i), \vartheta_3(\mathbf{K}_i), P_{c,i}, r) \quad r = 4n-1 \ \right\} \\ \left\{ \ \mathbf{E}_{i,post} = \kappa_{post}(\mathbf{K}_i, P_{c,i}) \ \right\} \end{array}$$

Now that the key schedule has been established, the the shorthand of writing the compression function $\Psi_{i,\mathbf{D}_i}(\mathbf{S}_i) = \Psi(\mathbf{S}_i, \mathbf{D}_i, i)$ shall be used.

Once the round keys are created, the only other preparation is the permutation matrix, which is created as $\mathbf{R}_i = \varepsilon(\xi_g, \mathbf{K}_i, b_l)$

If we let $\Gamma_{pre,\mathbf{R}_i}(\mathbf{M}) = \Gamma_{pre}(\mathbf{M}, \mathbf{R}_i, \mathbf{E}_{pre})$, $\Gamma_{post,\mathbf{R}_i}(\mathbf{M}) = \Gamma_{post}(\mathbf{M}, \mathbf{R}_i, \mathbf{E}_{post})$, and $\Upsilon_{\mathbf{E},r}(\mathbf{M}) = \Upsilon(\mathbf{M}, \mathbf{E}, r)$ for notational convenience, then $\Psi : \mathcal{M}_{16,16} \times \mathcal{M}_{16,16} \to \mathcal{M}_{16,16}$ can now be defined.

If the current number of rounds being used is of the form $r = 4n$ then,

$$\Psi_{i,\mathbf{D}_i} : \mathbf{S}_i \mapsto (\Gamma_{post,\mathbf{R}_i} \circ \underbrace{\Upsilon_{\mathbf{E}_r, r(\mathrm{mod}4)}}_{\text{repeated } r \text{ times}} \circ \Gamma_{pre,\mathbf{R}_i})(\mathbf{S}_i)$$

If the current number of rounds being used is of the form $r = 4n + 2$ then,

$$\Psi_{i,\mathbf{D}_i} : \mathbf{S}_i \mapsto (\Gamma_{post,\mathbf{R}_i} \circ \Upsilon_{\mathbf{E}_{4n-1},3} \circ \underbrace{\Upsilon_{\mathbf{E}_{r+1}, r(\mathrm{mod}4)}}_{\text{repeated } r-2 \text{ times}} \circ \Upsilon_{\mathbf{E}_0,0} \circ \Gamma_{pre,\mathbf{R}_i})(\mathbf{S}_i)$$

3.6.4 Σ - The Full Algorithm

For the full algorithm the initial state is set to a copy of the generic S-Box ξ_g (the AES S-Box) which will be identified as $\mathbf{S}_0$. The IV value must not be user specified as it may have a detrimental on the algorithm's security.

The full algorithm

$$\Sigma : \{0,1\}^* \times \{224, 256, 384, 512, 768, 1024, 1536, 2048\} \rightarrow \{0,1\}^{\{224,256,384,512,768,1024,1536,2048\}} \tag{76}$$

for n message blocks, where h is the required digest length is defined by:

$$\Sigma : \mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_{n-1}, h \mapsto \varrho_h(\Psi_{n-1,\mathbf{D}_{n-1}} \circ \ldots \Psi_{1,\mathbf{D}_1} \circ \Psi_{0,\mathbf{D}_0})(\mathbf{S}_0) \tag{77}$$

4 Design Rationale

4.1 High-level Priorities

The fundamental priority when designing Sgàil was that of security. Given that the successful candidate for SHA-3 is likely to be in service for several decades, security considerations must trump any potential marginal performance gains. SHA-3 must also be resistant not only to attacks existing today, but any likely attack to be developed in the future. In light of this it is prudent to design in several orders of magnitude of safety, to create a design that is simple to verify against today's cryptanalytic techniques but not so simple that it is easy to base or apply new attacks against, and most importantly to rely on computational complexity as far as possible. Computationally hard problems generally stand the test of time better than clever short-cuts.

The next priority in line was flexibility. In the event of an unexpected requirement or weakness, the ability for the successful algorithm to be easily modified or adapted, or even for a parameter to be changed may make the difference between the necessity for a complete redesign and an easy modification.

The third priority was purposefulness. Every element of the design should be there for a specific and definite purpose. Using something because it looks nice or without reason will lead to disaster. Sounds obvious, but it should really be mentioned explicitly.

The final priority considered was that of ease of implementation and performance. As long as the algorithm isn't obtusely complex and will run in reasonable speed, then it should be acceptable.

4.2 Target Platform Priorities

Sgàil while implementable in software, hardware and embedded devices, is definitely orientated towards performance on 64-bit general purpose CPUs. The following shows in descending order, the design priorities for target platforms.

1. 64-bit general purpose CPUs : Native 64-bit CPUs are now becoming the norm in both desktop and server hardware with the idea of using multiple execution cores to replace the focus on raw clock frequency becoming wide spread. It is submitted that any performance improvement in a cryptographic hash algorithm would be most beneficial being targeted towards these CPUs, principally due to their ubiquity of use from server applications to VPN devices. As such the native word size of Sgàil is 64-bits wide, it makes extensive use of native 64-bit word operations and it has been designed from the ground up to be parallelisable to many execution threads.

2. Hardware : The design of the Sgàil was with hardware implementation in mind - standard bitwise operations are used extensively, most of the table lookups can be reduced to bitslice operations (the 8x8 S-Boxes are generated from smaller 4x4 mini-boxes) and be run in parallel and there is a lot of opportunity to duplicate operations. The major hurdle to efficient hardware implementation is the generation of the matrix used to permutate the state array which is created from an RC4 style algorithm; it is expected that this will unfortunately have to be performed using RAM (however as it is not dependent on the state matrix it is parellelisable on the message block level).

3. Embedded processors : With embedded processors, it is unlikely that any vast amount of data will need to be processed. As such, as long as it is possible to implement the algorithm and that it executes in an acceptable amount of time, then it will suffice. The principle hindrance to implementation in embedded processors would be the available RAM, however with devices offering 1Kb or more of RAM now widely available and the fact that processing power is only going in one direction - it is unlikely the RAM requirements of Sgàil would pose any significant hindrance. There is a significant requirement for storage of the S-Boxes, MDS matrices and code - however, again any real practical difficulty will be minimal and would be expected to decrease over time.

4.3 Trade-offs

There are a number of trade-offs that can conceivably be made, dependent on the target environment. The MDS tables can be pre-calculated or the transform could be performed explicitly (as would probably be required in a embedded implementation). The generation of the Cauchy matrices can even be done online, although this is probably for entertainment purposes only.

On most implementations, if the tables can be stored in memory it is probably best to do so. There was effort made to separate the key processing from the compression function so that it may be performed separately (e.g. in separate memory space) if resources are limited.

4.4 Tunable Parameters

The main tunable parameters would be the number of rounds in the state update and the number of rounds in the principle key processing. It is unlikely that this would be required, however the option is supplied in the development code for assessment purposes. Essentially, the number of rounds in the state update controls protection against pre-image attacks, while the number of rounds in the key derviation controls protection against collision attacks. If either of these ever needed to be increased, it would likely be the number of key derivation rounds.

Using the τ with θ_8 in each round (whether that be state update or key derivation) would improve the security however it would also have a severe performance penalty. An additional quad diffuse in each layer would also increase the security, again this would incur a significant performance hit.

5 Performance and Implementation

5.1 General Performance

Sgàil was designed to give a high level of security with good performance for large quantities of data. Taking Whirlpool as a standard, then Sgàil suffers quite badly for short message lengths, however when the quantity of input data starts to rise then Sgàil becomes more efficient and the performance levels out. This will be due in part to Whirlpool's input block size being 512-bits and the minimum input block size that Sgàil can accommodate is 2048-bits - even at input message sizes between 1536 and 2048 bits, Whirlpool is faster by a good margin. However when the input size starts to hit near the 4096-bit mark and for much larger data sets, then Sgàil exhibits the same speed. With an optimised implementation, on asymptotic behaviour, it is expected that Sgàil will be faster than Whirlpool for large data sets.

Sgàil has consistent timings on most new processors, although older processors with smaller on chip caches can hit a performance barrier if other processes are also using the cache.

5.2 NIST Reference Platform

The following test was done on a Intel Dual Core 2.6Ghz server running FreeBSD amd64 7.0 and compiled with gcc 4.2.1. This should be fairly close to the reference platform.

The test was the time taken to hash $10,000,000$ instantiations of increasing input sizes. The left hand column is the number of input bytes, the next two columns indicate the time taken for the Sgàil and Whirlpool's reference implementation, and the last two columns are the corresponding cycles per byte for Sgàil and Whirlpool.

Bytes	Sgàil (secs)	Whirlpool (secs)	Sgàil (cpb)	Whirlpool (cpb)
64	112	34	455	136
128	113	51	230	104
256	113	86	114	88
384	143	121	96	82
512	257	156	130	79
1024	398	296	101	75
2048	684	576	87	73
4096	1257	1136	80	72
8192	2401	2256	76	72
16384	4642	4498	74	71
32768	9141	8978	72	71
65536	18190	17900	72	71
131072	36108	35880	71	71
262144	72080	71720	71	71

As you can see, Sgàil suffers for small block sizes but the performance quickly becomes appreciable with larger quantities of data.

The asymptopic speed for the reference implementation of Sgàil is roughly 71 cycles per byte; better perfomance would be expected with an optimised implementation.

5.3 Hardware

The core of the state update should run efficiently in hardware, the only word operations used are modular addition, exclusive-or and fixed rotations. The S-Boxes used in the core of the state update are all generated from smaller miniboxes which should make optimisation easier. The main sticking point is the permutation layer and generation of the permutation matrix - which will probably be the main impedance to performance.

Any estimate of gate count would be a total guess, but something like 10-15 times the count of an algorithm like TwoFish would likely be in the right ball park.

5.4 Embedded

The performance on embedded (8-bit) processors is not likely to be good. The algorithm is optimised for 64-bit architectures. However it should run in similar times to comparable hash algorithms, and in available RAM. There will be a substantial amount of ROM required for the code, S-Boxes and MDS Matrices - almost 7Kb for the S-Boxes and MDS matrices alone.

5.5 Parallelisation

Sgàil offers parallelisation opportunities both within the compression function and at the inter block level. The key processing for message blocks can be done in advance with the key processing accounting for roughly 30% of the work load of the algorithm, it is an appreciable amount to be able to offload.

6 Security Analysis

6.1 Resistance to Standard Linear and Differential Cryptanalysis

Using the same argument as in [Rijmen and Barreto(2001)], we can use the branch number of the diffusion layer and S-Box strength to estimate the upper bound on a differential or linear characteristic.

The worst S-Box in Sgàil has a $DP_{max} = 14/256$, also the 128-Bit MDS combined with assured diffusion gives the maximal branch number of $\eta = 17$. Using these figures gives any differential characteristic an upper probability bound of $\delta^{\eta^2} = (2^{-5.5})^{289} \simeq 2^{-1590}$ which already far exceeds the requirements for a 512-bit digest (for which four inner rounds are used). This calculation excludes additional beneficial factors including; the nonlinearity in the diffusion layer, the two S-Box and 64-bit MDS transforms at the start and end of each round (so the number of rounds is really nearer 6), and the keyed diffusion. It is also only applicable to pre-image attacks. Collision attacks would have to be considered seperately and focus on how the principle key is arrived at.

6.2 Orthogonality

Hash algorithms have different pitfalls as compared to block cipher design - one eye has to be kept on ensuring that collision attacks are not feasible. One of the very deliberate techniques employed in Sgàil is the use of orthogonality - that is using the same input to alter two things at once in an assuredly different way. At least conceptually, this should make it harder to manipulate any inputs as achieving the desired effect in one area will totally destroy it in another. If one looks at the design of Sgàil this concept is tied into almost every operation from the key extraction routines to the design of the non-linear layer.

6.3 Complexity Argument

While obscurity is almost always a very bad idea, precisely ordered complexity is generally a very good thing. Given that most public key systems are based on the value of computational complexity, then an algorithm presenting with high computational complexity to break and no easy way of simplification should be something that can be relied upon.

7 Appendices

7.1 Constants

The quad diffuse rotation constants are as follows:

```
QD_0_ROT_0 28
QD_0_ROT_1 6
```

```
QD_0_ROT_2 55
QD_1_ROT_0 36
QD_1_ROT_1 58
QD_1_ROT_2 9
QD_2_ROT_0 8
QD_2_ROT_1 24
QD_2_ROT_2 43
QD_3_ROT_0 9
QD_3_ROT_1 47
QD_3_ROT_2 39
QD_X_ROT_0 9
QD_X_ROT_1 18
QD_X_ROT_2 27
QD_X_ROT_3 36
```

The main rotation constants are empirically optimised and the "X" constants are just $(1 \cdot 8) + 1, (2 \cdot 8) + 2, (3 \cdot 8) + 3, (4 \cdot 8) + 4$ with a view to making sure the result of the quad diffuse affects different S-Box inputs in each quadrant.

The key extract rotate constants are chosen as odd numbers near 0, 16 and 32 - in implementation the round number is added anyway, so as long as they are well spread out - that is the main thing.

```
KE_ROT_1 3
KE_ROT_2 17
KE_ROT_3 29
```

The MDS post rotate constants are chosen pretty much at random - prime numbers spread over the range were thought to be good. They can be replaced by almost anything, the main purpose is to destroy the byte alignment - so anything other than zero can be chosen.

```
MDS__64BIT__ROTATE 23
MDS__128BIT__ROTATE_LHS 11
MDS__128BIT__ROTATE_RHS 37
```

7.2 Cauchy Matrices

This is the $\mathbf{C}_{16}$ matrix:

```
01 b2 77 bf 79 56 32 62  0b e6 9d ad 5b 8c cd 41
8d ff 19 4c 85 96 6a ce  a3 1b 6b c6 34 68 1c 1a
4f 5f f3 17 b6 a4 9f 48  a9 0f c5 b3 dd 1c 68 03
e5 74 6c 18 82 9b dc fb  e6 0b 63 08 1c dd 34 23
3f d1 55 09 f4 9a 95 71  94 ea da 1c 08 b3 c6 cf
5a 77 b2 10 ca 8f e0 f7  eb 5d 1c da 63 c5 6b f8
45 c2 b0 43 1d 71 2e 9a  5b 1c 5d ea 0b 0f 1b ac
59 3a f6 f4 09 a5 e7 de  1c 5b eb 94 e6 a9 a3 28

5c 09 33 d1 3a ea 2f 1c  de 9a f7 71 fb 48 ce d2
```

```
33 97 5c 20 40 d7 1c 2f  e7 2e e0 95 dc 9f 6a 6d
70 43 2d c2 e8 1c d7 ea  a5 71 8f 9a 9b a4 96 be
b9 de ef 94 1c e8 40 3a  09 1d ca f4 82 b6 85 b7
c8 71 d8 1c 94 c2 20 d1  f4 43 10 09 18 17 4c 87
2f 9d 1c d8 ef 2d 5c 33  f6 b0 b2 55 6c f3 19 bb
0d 1c 9d 71 de 43 97 09  3a c2 77 d1 74 5f ff ee
41 b1 a3 26 a4 b6 93 ed  19 2c f1 58 c7 e8 f6 01
```

This is the $\mathbf{C}_8$ matrix:

```
01 b2 77 bf 79 56 32 62
0b e6 9d ad 5b 8c cd 41
8d ff 19 4c 85 96 6a ce
a3 1b 6b c6 34 68 1c 1a
4f 5f f3 17 b6 a4 9f 48
a9 0f c5 b3 dd 1c 68 03
e5 74 6c 18 82 9b dc fb
e6 0b 63 08 1c dd 34 23
```

7.3 Round Constants

Probably badly named, the round constants are used in certain places throughout Sgàil and is an array of 64 x 64-bit words, which is just the hexadecimal representation of Pi after the decimal point. The array is reproduced here

```
0x25d479d8f6e8def7
0xe3fe501ab6794c3b
0x976ce0bd04c006ba
0xc1a94fb6409f60c4
0x5e5c9ec2196a2463
0x68fb6faf3e6c53b5
0x1339b2eb3b52ec6f
0x6dfc511f9b30952c
0xcc814544af5ebd09
0xbee3d004de334afd
0x660f2807192e4bb3
0xc0cba85745c8740f
0xd20b5f39b9d3fbdb
0x5579c0bd1a60320a
0xd6a100c6402c7279
0x679f25fefb1fa3cc
0x8ea5e9f8db3222f8
0x3c7516dffd616b15
0x2f501ec8ad0552ab
0x323db5fafd238760
0x53317b483e00df82
0x9e5c57bbca6f8ca0
0x1a87562edf1769db
0xd542a8f6287effc3
```

```
0xac6732c68c4f5573
0x695b27b0bbca58c8
0xe1ffa35db8f011a0
0x10fa3d98fd2183b8
0x4afcb56c2dd1d35b
0x9a53e479b6f84565
0xd28e49bc4bfb9790
0xe1ddf2daa4cb7e33
0x62fb1341cee4c6e8
0xef20cada36774c01
0xd07e9efe2bf11fb4
0x95dbda4dae909198
0xeaad8e716b93d5a0
0xd08ed1d0afc725e0
0x8e3c5b2f8e7594b7
0x8ff6e2fbf2122b64
0x8888b812900df01c
0x4fad5ea0688fc31c
0xd1cff191b3a8c1ad
0x2f2f2218be0e1777
0xea752dfe8b021fa1
0xe5a0cc0fb56f74e8
0x18acf3d6ce89e299
0xb4a84fe0fd13e0b7
0x7cc43b81d2ada8d9
0x165fa26680957705
0x93cc7314211a1477
0xe6ad206577b5fa86
0xc75442f5fb9d35cf
0xebcdaf0c7b3e89a0
0xd6411bd3ae1e7e49
0x00250e2d2071b35e
0x226800bb57b8e0af
0x2464369bf009b91e
0x5563911d59dfa6aa
0x78c14389d95a537f
0x207d5ba202e5b9c5
0x832603766295cfa9
0x11c819684e734a41
0xb3472dca7b14a94a
```

References

[Rijmen and Barreto(2001)] Rijmen, V., Barreto, P. S. L. M., 2001. The WHIRLPOOL hash function. World-Wide Web document. URL `http://paginas.terra.com.br/informatica/paulobarreto/WhirlpoolPage.html`; `http://planeta.terra.com.br/informatica/paulobarreto/whirlpool.zip`

[Schneier et al.(1998)Schneier, Kelsey, Whiting, Wagner, Hall, and Ferguson] Schneier, B., Kelsey, J., Whiting, D., Wagner, D., Hall, C., Ferguson, N., 1998. Twofish: A 128-bit block cipher. In: in First Advanced Encryption Standard (AES) Conference.

[TheRegister(Sept2006)] TheRegister, Sept2006. Intel fabs 80 core teraflop processor.
URL http://www.reghardware.co.uk/2006/09/26/intel_teraflop_processor/

[Youssef et al.(1997)Youssef, Mister, and Tavares] Youssef, A. M., Mister, S., Tavares, S. E., 1997. On the design of linear transformations for substitution permutation encryption networks. In: School of Computer Science, Carleton University. pp. 40–48.