Scalable PMC Hash

C. B. Roellgen

PMC Ciphers, Inc, Josephsburgstr. 85, 81673 Munich, Germany

initial date of release: 25.09.2004, latest revision: 11.07.2007

Abstract. We present Scalable PMC Hash, a 1024-bit hash function operating on messages less than 2^{512} bits, and block size can be scaled to any practical length. The function structure is designed for compatibility with any existing 32 or 64 bit microprocessor.

1. Introduction

In this document we describe Scalable PMC Hash, a one-way, collision resistant 1024-bit hashing function operating on messages less than 2^{512} bits in length. Scalable PMC Hash consists of the application of a cascade of standard compression functions. Input to each of the standard compression functions is a permutation of the original data block. This permutation is keyed by an additive, as well as multiplicative combination of standard cryptographic functions employing popular symmetric ciphers and hash functions in a Miyaguchi-Preneel structure. Bit and word permutation is additionally applied to the output bit string of the cascaded standard compression functions. Initialisation vectors for this operation are generated by a polymorphic pseudorandom number generator function.
The design is in principle scalable.

2. Description of the Scalable PMC Hash primitive

The Scalable PMC Hash primitive is a Merkle hashing function (cf. [1, algorithm 9.25]) based on a set of cascaded standard compression functions that take the place of a dedicated block cipher. The set of standard compression functions operate on independent internal states all derived from the same input data which is permutated individually for each of the standard compression functions. 2048 bit of chained key state which is derived from input data after processing with a polymorphic pseudorandom number generator, is provided by a Lagged Fibonacci RNG.

In the following the component mappings and constants that build up Scaled PMC Hash are individually defined. The complete hashing function is subsequently defined in terms of these components.

2.1 Input

The message block *M* is internally viewed as a block of 8n bit which are represented as an array of n/4 chunks m_i each 32 bit wide.

The hash state *H* is internally viewed as a block of 1024 bit which are represented as an array of 32 chunks each 32 bit wide. Intermediate data is mapped by function *μ: CH(a) = b* to the hash state:

$\mu(a) = b \Leftrightarrow b_{ij} = a_{32i+j}$, $0 =< i, j =< 31$

2.2 32 bit rotation layer

In order to introduce nonlinearity at an early stage, 32 bit chunks of message block *m:* m_i are rotated left with dependence on preceeding 32 bit chunks m_{i-1} using the following algorithm:

```
for (i=1; i<n;i++) {
        m[i]=ROTATE_LEFT32(m[i],((m[i-1]>>28) & 0x000F)+3);
        m[i]=ROTATE_LEFT32(m[i],((m[i-1]>>24) & 0x000F)+3);
        m[i]=ROTATE_LEFT32(m[i],((m[i-1]>>20) & 0x000F)+3);
        m[i]=ROTATE_LEFT32(m[i],((m[i-1]>>16) & 0x000F)+3);
        m[i]=ROTATE_LEFT32(m[i],((m[i-1]>>12) & 0x000F)+3);
        m[i]=ROTATE_LEFT32(m[i],((m[i-1]>>8) & 0x000F)+3);
        m[i]=ROTATE_LEFT32(m[i],((m[i-1]>>4) & 0x000F)+3);
        m[i]=ROTATE_LEFT32(m[i],(m[i-1] & 0x000F)+3);
}
```

with `ROTATE_LEFT32(v,n) = ((v) << (n)) | ((v) >> (32 - (n))) ;`

The result of this process is denoted Intermediate Block *IB*. *IB* is internally viewed as a block of n/4 bit which are represented as an array of n/4 chunks ib_i each 32 bit wide.

2.3 Keyed permutation (transposition) layer for 32 bit chunks

16 bit chunks of Intermediate Block *IB:*

$c_{ij} = ib_{16i+j}$, $0 =< i, j =< 15$

are transposed by iterating through *IB* from left to right (i = 0 .. n/2) while the 16 bit chunk c_i is exchanged with another 16 bit chunk c_j. Index *j* is selected by a Lagged Fibonacci PRNG:

$c_i, c_j = Exchange(c_i, c_j)$, $j_x = lf(K_T, IS_{lf}, x)$, *lf = Lagged Fibonacci PRNG function,* K_T *= 2048 bit key of Lagged Fibonacci PRNG,* IS_{lf} *= 2048 bit Internal State of Lagged Fibonacci PRNG , x = iteration index of Lagged Fibonacci PRNG*

The function is subsequently denoted *p*.

2.4 Keyed rotation layer for 32 bit chunks

32 bit chunks of Intermediate Block *IB* are rotated by iterating through *IB* from left to right (i = 0 .. n/2) and by rotating each element ib_i by applying the following function:

$ROTATE\_LEFT32(v,j_x) = ((v) << (j_x)) \mid ((v) >> (32 - (j_x)))$;

with $v = c_i$, $j_x = lf(K_T, IS_{lf}, x)$, , $j_x = 1, 2, .. 31$, *lf = Lagged Fibonacci PRNG function,* K_T *= 2048 bit key of Lagged Fibonacci PRNG,* IS_{lf} *= 2048 bit Internal State of Lagged Fibonacci PRNG , x = iteration index of Lagged Fibonacci PRNG*

The function is subsequently denoted *r*.

2.5 Lagged Fibonacci PRNG

The Lagged Fibonacci PRNG *lf* is used as source for pseudorandom numbers in function *p* according 2.3 and *r* according 2.4. This algorithm is also known as ARCFOUR [2]. The most important weakness is bias in the first few bytes. After initialisation with the first message-dependent key, the algorithm is executed 512 times and the resulting bit stream is not used further.

```
Initialisation Mode:
Input: n=8,
        key length (in bytes) = k;
        S[i]=i for i=0 to 255
        Key : K[0…(k-1)]
```

```
j=0;
For i=0 to (n-1)
        j=j+S[i]+K[i mod k]
        swap(S,i,j)
        i=j=0;

Output Mode:
Repeat
        i=i+1
        j=j+S[i]
        swap(S,i,j)
        output=S[S[i]+S[j]]
Until done
```

Rekeying of the Lagged Finonacci PRNG *lf*:

A new partial key `ctx->transposition_key`, which is generated by the Static Polymorphic Compression Function acc. 2.7, is shifted 256 bit-wise into the lagged Fibonacci key array.

```
        for (i=0;i<32;i++) {
                ctx->lagged_fibonacci_key[i+224]=ctx->lagged_fibonacci_key[i+192];
                ctx->lagged_fibonacci_key[i+192]=ctx->lagged_fibonacci_key[i+160];
                ctx->lagged_fibonacci_key[i+160]=ctx->lagged_fibonacci_key[i+128];
                ctx->lagged_fibonacci_key[i+128]=ctx->lagged_fibonacci_key[i+96];
                ctx->lagged_fibonacci_key[i+96]=ctx->lagged_fibonacci_key[i+64];
                ctx->lagged_fibonacci_key[i+64]=ctx->lagged_fibonacci_key[i+32];
                ctx->lagged_fibonacci_key[i+32]=ctx->lagged_fibonacci_key[i];
                ctx->lagged_fibonacci_key[i]=ctx->transposition_key[i];
        }
```

Subsequently the algorithm is run in Initialisation Mode and the first 8 output bytes are discarded.

2.6 Creation of sub-blocks for internal key generation

The internal state of the Lagged Fibonacci PRNG *lf* and indirectly the internal state of the Polymorphic PRNG layer are both computed from the Intermediate Block *IB* by partitioning the Intermediate Block *IB* into a variable number of Sub-Blocks $IBS_1 .. IBS_{ns}$ each 128 bit in length. The number *ns* of Sub-Blocks $IBS_0 .. IBS_{ns-1}$ computes from the length *n* in bytes of message *m* in bytes:

$$ns = 8n/128;$$

Message *m* is regarded as 2 bit array (of tuples):

$$d_{ij} = ib_{2i+j}\ , \qquad 0 =< i =< n/2\ ,\ 0 =< j =< 1$$

Consecutive tuples *i* of d_j are added to Sub-Blocks IBS_x using the algorithm below in a way that tuple *i* is copied in sub-block IBS_x and tuple *i+1* is copied in sub-block IBS_{x+1} :

```
int bit_count;
int count_to_15;
int i;
int     ns=n*8/128;

        // clear all sub-blocks       (treated as one array)
for (i=0; i<n;i++) sub_blocks[i]=0;

count_to_15=-1;               // (16*8 = 128 bit granularity of sub-blocks)
bit_count=0;                  // counter for 2-bit chunks

for (i=0; i<n/4;i++) {        // no. of DWORDs in intermediate block
        count_to_15++;
        count_to_15=count_to_15 & 0x000F;

        for (block_cnt=0;block_cnt<ns;block_cnt++) {
                sub_blocks[count_to_15+block_cnt*128/4]=
                              (uint8)((m[i] >> bit_count) & 0x03)
                              | (sub_blocks[count_to_15+block_cnt*128/4] << 2);
```

```
            bit_count=bit_count+2;
            bit_count=bit_count & 0x001f;
        } // for (block_cnt=0;block_cnt<ns;block_cnt++)
} // for (i=0; i<n/4;i++)
```

This operation breaks up byte boundaries and portions out the full content of Intermediate Block *IB* to a number of 128 bit blocks to be further processed by the Combined Miyaguchi-Preneel Hashing Function. The Static Polymorphic Compression Function acc. 2.7 requires a multiple of 4 Sub-Blocks to exist. With the default message length *n* of 256 bytes, *ns* equals 16.

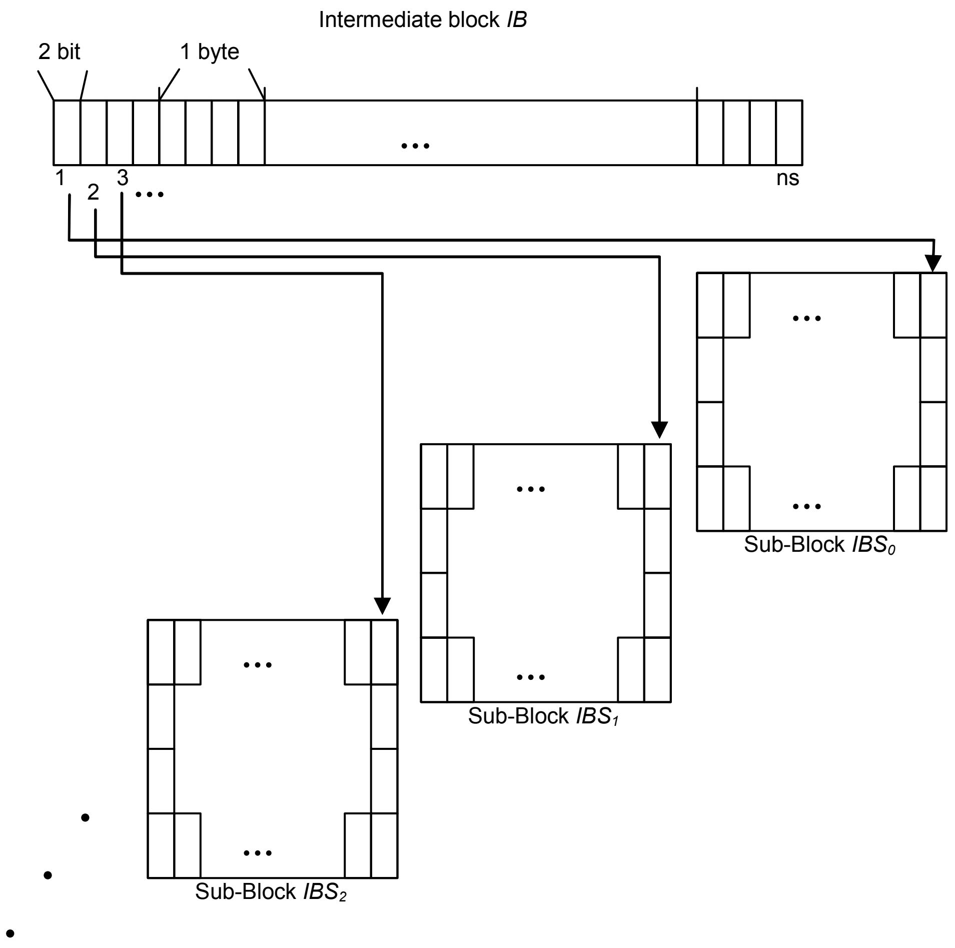

Fig. 1 Generation of 128 bit Sub-Blocks IBS_1 .. IBS_{ns} from the Intermediate Block *IB*

2.7 Combined Miyaguchi-Preneel Hashing Function

Scalable PMC Hash relies for the generation of transposition- and rotation keys on a combined Miyaguchi-Preneel hashing scheme. Input to the this building block is an initialisation vector that is computed by the Polymorphic PRNG acc. 2.8 and on Sub-Blocks IBS_0 .. IBS_{ns-1} .
The 32 bit rotation layer acc. 2.2 and the creation of Sub-Blocks acc. 2.7 yield 2 bit wide tuples of intermediate block *IB* to be copied into Sub-Blocks Blocks IBS_0 .. IBS_{ns-1} in a way that makes use of message *m* as key. Through this keyed operation, any tuple can potentially reside in any Sub-Block IBS_i and as a matter of consequence be fed into one out of four conceptually different hash- and encryption algorithm combinations in a Miyaguchi-Preneel scheme. Figure 2 explains the interactions of the primitives:

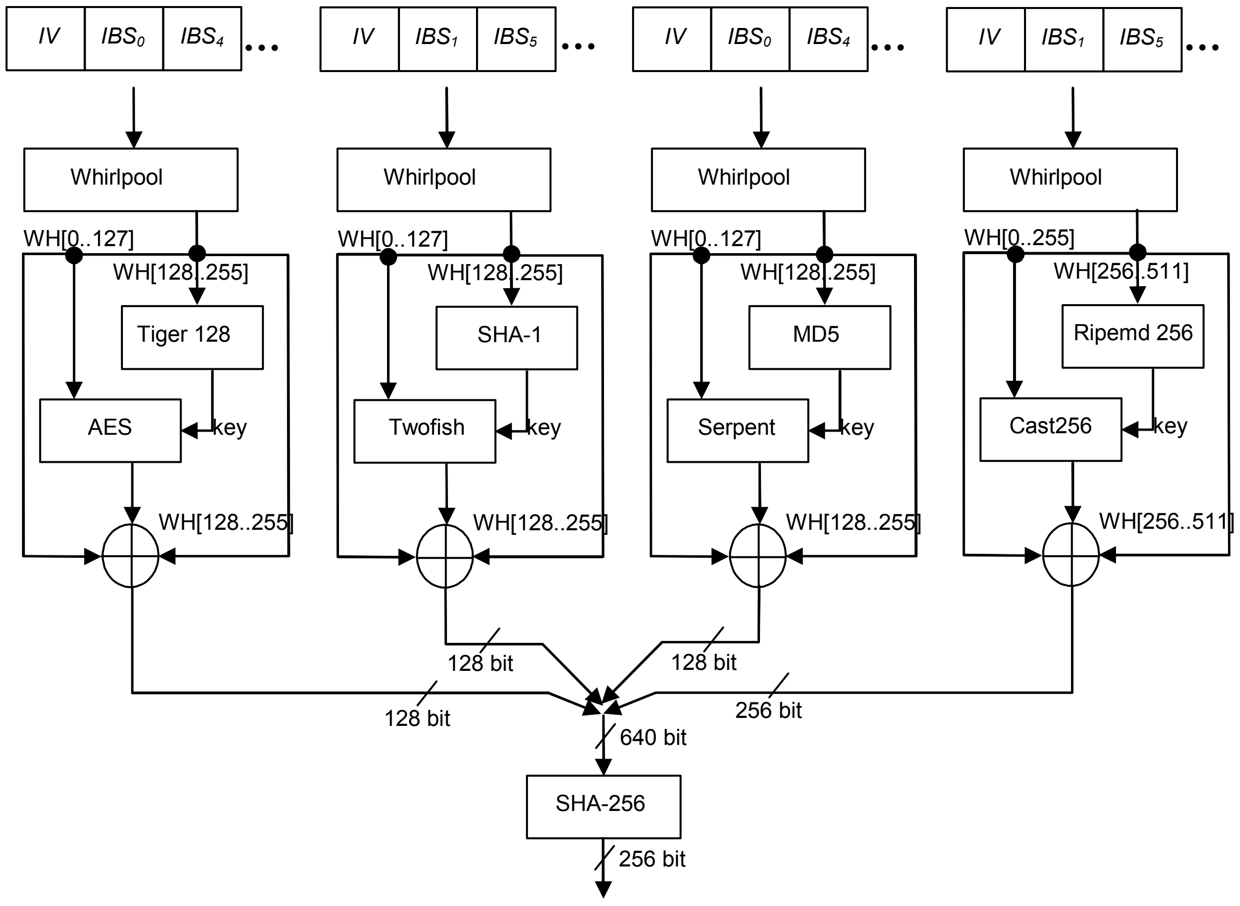

Fig. 2 Combined Miyaguchi-Preneel Hashing Function that compresses the initialisation vector IV and Sub-Blocks IBS_1 .. IBS_{ns} to yield a 256 bit hash which is used to rekey the Lagged Fibonacci PRNG acc. 2.5

The IV, as well as Sub-Block data are first compressed by a Whirlpool primitive in order to yield fixed array lengths.

2.8 Polymorphic PRNG

Scalable PMC Hash utilizes a Polymorphic PRNG in order to compute the initialisation vector *IV* that biases the Whirlpool primitive in the Combined Miyaguchi-Preneel Hashing Function acc. 2.7. A set of 16 conceptually similar PRNG functions is available from which it is possible to choose one at a time freely by the part of the algorithm that generates initialisation vector *IV*. The 16 PRNG functions are identical in machine code size, execution time, but differ in functionality. They are called from an array containing the entry points of all 16 functions. In the C programming language all 16 PRNG functions must be of the same type:

```
typedef void (static_pmc_routine)(scalable_pmc_hash_context * ctx);
```

In order to select and to execute a specific Lagged Fibonacci PRNG function, a call must be made to the pointer of the respective function type:

```
static_pmc_routine * pStatic_pmc_routine;    // pointer on one of the available
                                                polymorphic routines

pStatic_pmc_routine=(static_pmc_routine *)ctx->routine_array[j];   // select a function
(*pStatic_pmc_routine)(ctx);                 // execute function
```

Scalable PMC uses part of the internal state of the Polymorphic PRNG and the ARCFOUR algorithm to generate initialisation vector *IV*:

```
j=(int)((ctx->count*SCALABLE_PMC_HASH_DATALEN+(uint64)ctx->index));

for (i=0;i<16;i++) {
       pStatic_pmc_routine=(static_pmc_routine *)ctx->routine_array[j & 16];
       (*pStatic_pmc_routine)(ctx);
       transposition_iv[i]=(uint8)((ctx->pmc_routines_secondary_index +
                              ctx->pmc_routines_index + i +
                              (uint32)lagged_fibonacci_compute_next_8_bit(ctx)) & 0x00ff);
       j=(j + (int)lagged_fibonacci_compute_next_8_bit(ctx)) & 0x00ff;
}
```

Due to the nature of Polymorphism, notably the calling of a variety of operations using the same interface, code is turned into a variable itself.
Scalable PMC Hash uses the following algorithm:

```
Internal state:
uint32 ui1 and ui2;

Local variables:
uint32 i1 and i2;     //



ui1=pmc_is[i1 & 15]+lf();

       // can be + or -
ui2=pmc_is[i2 & 15]    +    ui1 +lf();

       // can be + or -
ui1=ui2    +    pmc_is[pmc_running_counter]+lf();
pmc_running_counter=(pmc_running_counter+1) & 0x000F;

pmc_is[ui1 & 15]=pmc_is[ui1 & 15] ^ lf() ^ (lf() << 8) ^ (lf() << 16) ^ (lf() << 24);
pmc_is[ui2 & 15]=pmc_is[ui2 & 15] ^ lf() ^ (lf() << 8) ^ (lf() << 16) ^ (lf() << 24);

       // can be + or -
i2=i2 ^ i1 ^ (ui2    +    ui1);

       // can be + or -
i1=(i1 << 4)    +    ui2+pmc_running_counter;
```

The PRNG is a Modified Lagged Fibonacci Pseudorandom Number Generator. The new term is some combination of any two previous terms with an additional bias by the ARCFOUR algorithm *lf*.
The recurrence relation for this sequence of pseudorandom numbers is given with

ui_2 = pmc_is[$i_2(n-1)$ AND 15] (*) pmc_is[$i_1(n-1)$]+*lf*(11 n) + *lf*(11 n +1)
ui_1 = pmc_is[$i_2(n-1)$ AND 15] (*) pmc_is[$i_1(n-1)$] (*) pmc_is[n AND 15]+ *lf*(11 n) + *lf*(11 n +1) + *lf*(11 n+2)

by the following equations:

$i_2(n) = i_2(n-1)$ XOR $i_1(n-1)$ XOR (ui_2 (*) ui_1)
$i_1(n) = (i_1(n-1) \cdot 16)$ (*) ui_2 + (n AND 15)

The operator (*) denotes a general binary operation, which is limited in Scalable PMC Hash to addition and subtraction. It would also be possible to add binary XOR to the list of available operations.

2.9 Cascaded hashing and block compression function

Compression to the fixed 1024 bit size as well as hashing of intermediate block IB ib_i is performed by a cascade *CH* of three standard compression functions f_1 , f_2 and f_3 that are known to be free of collisions:

$CH_i = r[p[\, f_1(p(b) \,||\, f_2(r(p(b))) \,||\, f_3(p(r(p(b))))]]$; *with* f_1 *= Whirlpool primitive,* f_2 *= Haval 256 primitive,*
f_3 *= Ripemd 256 primitive,*
r = keyed rotation of 32 bit word,
p = keyed permutation of 32 bit chunks

f_1 , f_2 and f_3 are cascaded which leads according [1, fact 9.27] to a collision resistant hash function with the number of output bits being the sum of the output bits of the cascaded functions.
It is important to note that Scalable PMC Hash does not simply concatenate output values from f_1 , f_2 and f_3. A keyed byte permutation and bit rotation layer mixes results from all three hash functions. With multi-collision attacks that are very effective to find local collisions in concatenated Merkle-Damgård hash functions, finding collisions for H = H(H1(M) || H2(M)) takes time O((n/2)*2^n/2) [13]. Keyed transposition of output bits in conjunction with keyed transposition of the inputs to f_1 , f_2 and f_3 although take away the basis for such attacks, notably the possibility to find local collisions for f_1 , f_2 *or* f_3 .

2.10 Padding and MD-strengthening

Before being subjected to the hashing operation, a message M of bit length $L < 2^{512}$ is padded with a 0x01-byte, then with 0-bits up to the block size boundary minus 64 bit. If the previously described operation was possible, a right-justified binary representation of L is added. If there is not enough space in the block, another block is filled with 0-bits and a right-justified binary representation of L is added to that block. The resulting bit string is the padded message m, partitioned in t blocks $m_1, \dots, m_t$. These blocks are viewed as byte arrays by sequentially grouping the bits in 8-bit chunks.

2.11 The compression function

Scalable PMC Hash iterates over the *t* padded message blocks m_i, $1 =< i =< t$, using the XOR function:

$H_0 = CH(m_0)$,
$H_i = H_{i-1} + CH(m_i)$, $1 =< i =< t$

2.12 Output

The iterated hash state H_i is repeatedly permutated (acc. 2.3) and rotated (acc. 2.4) prior to outputting the final hash state *HF*:

```
for (i=0;i<1024/32;i++) {
        HFi=p(Hi);
        HFi=r(Hi);
}
```

The Scalable PMC Hash message digest for message M is defined as the output *HF* of the compression function, mapped back to a bit string:
Scalable PMC(M) = μ^{-1}(HF).

3 Security goals

In this section, we present the goals we have set for the security of Scalable PMC. A cryptanalytic attack will be considered successful by the designers if it demonstrates that a security goal described herein does not hold.

3.1 Expected strength

It is assumed that the value of any *n*-bit substring of the full a hash result of Scalable PMC Hash output is taken. Then:

- The expected workload of generating a collision is of the order of $2^{n/2}$ executions of Scalable PMC Hash.
- Given an *n*-bit value, the expected workload of finding a message that hashes to that value is of the order of 2^n executions of Scalable PMC Hash.
- Given a message and its *n*-bit hash result, the expected workload of finding a second message that hashes to the same value is of the order of 2^n executions of Scalable PMC Hash.

Moreover, it is infeasible to detect systematic correlations between any linear combination of input bits and any linear combination of bits of the hash result. It is also infeasible to predict what bits of the hash result will change value when certain input bits are flipped, i.e. Scalable PMC Hash is resistant against differential attacks.
These claims result from the considerable safety margin taken with respect to all known attacks. It is however impossible to make non-speculative statements on future developments of cryptanalysis.

4 Analysis

Public research on hashing function cryptanalysis is, compared with available research on block cipher cryptanalysis, less extensively available.
Among the available research that is applicable to Scalable PMC Hash is the following:

4.1 Security of cascaded hash functions

According [1, fact 9.27], the cascading of hash functions leads to a collision resistant hash function, even if one of the cascaded hash functions is not collision resistant!
If hash functions h_1 and h_2 are independent, then finding a collision for $h = h_1 \| h_2$ requires finding a collision for both h_1 and h_2 simultaneously (i.e., on the same input). This could require the product of the efforts to attack h_1 and h_2 individually. Keyed transposition of output bits in conjunction with keyed transposition of the inputs to h1 and h2 are although needed to strengthen cascaded hash functions. Compressing the word length of *h* would be another option, but this is not really desirable as h_1 or h_2 would then required to be larger.

4.2 Security of the employed base hash functions

Scalable PMC Hash uses three base hash functions: Whirlpool, Haval 256 and RIPEMD 256.
All three base hash functions are supposed to be secure according 3. as to date. Consequently satisfies the cascade of these three base hash functions the outlined security goals. For a discussion of the implemented base hash functions, please see the respective papers [2, 3, , 12].

4.3 Security of additional diffusion measures of Scalable PMC Hash

Transposition adds diffusion [1, Remark 1.36], while confusion is provided by the base hash functions. Scalable PMC Hash additionally uses 32 bit rotation and permutation of 32 bit chunks on input and output of the three base hash functions. 32 rounds of 32 bit rotation and permutation of 32 bit chunks is applied on output bits in order to disguise the origin of individual hash bits, be it original bit position as well as the base function that is responsible for its state. This additional operation is solely in place to increase security if weaknesses are found in the future for one of the base hash functions and if only part of the combined hash is used in an implementation of Scalable PMC Hash.

The recurrence relation for this sequence of pseudorandom numbers is given by the two equations:

i_2(n) = i_2(n-1) XOR i_1(n-1) XOR ((pmc_is[i_2(n-1) AND 15] (*) pmc_is[i_1(n-1)]+*lf*(11 n) + *lf*(11 n +1)) (*) (pmc_is[i_2(n-1) AND 15] (*) pmc_is[i_1(n-1)] (*) pmc_is[n AND 15]+ *lf*(11 n) + *lf*(11 n +1) + *lf*(11 n+2)))

i_1(n) = (i_1(n-1) · 16) (*) (pmc_is[i_2(n-1) AND 15] (*) pmc_is[i_1(n-1)]+*lf*(11 n) + *lf*(11 n +1)) + (n AND 15)

The operator (*) denotes the binary operation addition and subtraction, which are chosen in a pseudorandom way and which are expressed as (*) == x + R(i) · y with randomiser R(i) = z for z € {+1, -1}, i = 6n.

This yields for the sequence of pseudorandom numbers:

i_2(n) = i_2(n-1) XOR i_1(n-1) XOR ((pmc_is[i_2(n-1) AND 15] + R(6n) · pmc_is[i_1(n-1)]+*lf*(11 n) + *lf*(11 n +1)) + R(6n+1) · (pmc_is[i_2(n-1) AND 15] + R(6n+2) · pmc_is[i_1(n-1)] + R(6n+3) · pmc_is[n AND 15]+ *lf*(11 n) + *lf*(11 n +1) + *lf*(11 n+2)))

i_1(n) = (i_1(n-1) · 16) + R(6n+4) · (pmc_is[i_2(n-1) AND 15] + R(6n+5) · pmc_is[i_1(n-1)]+*lf*(11 n) + *lf*(11 n +1)) + (n AND 15)

In order to determine the values of all eight unknowns i_1(n-1), i_2(n-1), pmc_is[i_2(n-1) AND 15], etc., the same number of consecutive samples as there are unknowns are sufficient to crack the generator.
The randomiser although complicates the task. Six additional unknowns R(6n), R(6n+1), .., R(6n+5) are added with each cycle. An opponent gets n samples to crack but has to cope with n+6n variables.

The implemented ARCFOUR algorithm that keys 32 bit rotation and permutation is known to run on standard microprocessors as fast an on dedicated hardware because it is inherently serial. ARCFOUR is suitable for rekeying. The algorithm withstands standard divide and conquer attacks as a known division is required, which is prevented by the dynamic updating of tables. The large internal state prevents any kind of table-based attacks.
The key that rekeys ARCFOUR is computed by a combined Miyaguchi-Preneel hashing function acc. 2.7 from the original message and an initialisation vector that is generated by the Polymorphic PRNG acc. 2.8. The Miyaguchi-Preneel scheme has not been broken so far. By cascading hashing functions and by compressing the initialisation vector first, breaking this key generation function requires the breaking of all implemented building blocks - building blocks that are known to withstand any known attack as to date.
The Polymorphic PRNG finally has the unique property that the available set of PRNG base functions all feature the same power consumption pattern on CMOS hardware. Scalable PMC Hash is thus hardened against DPA (Differential Power Attack).

6 Design rationale

Scalable PMC Hash uses a set of cryptographic primitives (Whirlpool, AES, Twofish, etc.) that are known to be strong and combines them in an additive as well as multiplicative way.
Cascading of hash functions is the additive component, leading to a collision resistant hash function even if one of the cascaded hash functions is not collision resistant.
32 bit rotation and permutation of 32 bit chunks on input and output of the three base hash functions represents the multiplicative component, with a large number of standard cryptographic primitives in turn combined in a multiplicative, as well as additive way.

Weaknesses that might be found for individual building blocks (major bugs would already have been discovered in the evaluation phase) are invisible due to the polymorphic PRNG.
Additionally, a larger block size than available with any of the implemented cryptographic primitives, is provided.
Scalable PMC Hash is mainly intended for use as key compression function for symmetric ciphers. High speed is generally a disadvantage for this kind of application as the time to try possible keys for exhaustive search should be as long as possible. Key compression can as well be a preferred target for Differential Power Attack. By including a Polymorphic PRNG for the additional diffusion layers and by selecting a set of conceptually different cryptographic base functions that are not highly susceptible to DPA, Scalable PMC Hash is the first of a new class of engineered cryptographic algorithms featuring high complexity and use of

a Polymorphic PRNG.

References

1. A. Menezes, P. van Oorschot, and S. Vanstone, *Handbook of applied cryptography*, CRC Press, 1997, http://www.cacr.math.uwaterloo.ca/hac/ .

2. Dawson, Helen Gustafson, Matt Henricksen, Bill Millan, *Evaluation of RC4 Stream Cipher, 2002,* http://www.ipa.go.jp/security/enc/CRYPTREC/fy15/doc/1043_IPA-RC4_%20report_final.pdf .

3. P. S. L. M. Barreto and V. Rijmen, *The Whirlpool Hashing Function*, First open NESSIE Workshop, Leuven, Belgium, 13 -14 November 2000*,* http://planeta.terra.com.br/informatica/paulobarreto/whirlpool.zip .

4. R. Anderson and E. Biham, *Tiger: A Fast New Hash Function*, Fast Software Encryption - FSE'96, LNCS 1039, Springer-Verlag (1996)*,* http://www.cs.technion.ac.il/~biham/Reports/Tiger/tiger.ps .

5. Joan Daemen, Vincent Rijmen, *TheRijndael Block Cipher (AES Proposal: Rijndael)*, 1999, http://www.esat.kuleuven.ac.be/~rijmen/rijndael/rijndaeldocV2.zip .

6. U.S. DEPARTMENT OF COMMERCE/National Institute of Standards and Technology, *SECURE HASH STANDARD (SHA-1)*, 1995, http://www.itl.nist.gov/fipspubs/fip180-1.htm .

7. Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, Niels Ferguson, *Twofish: A 128-Bit Block Cipher*, 1998, http://www.schneier.com/paper-twofish-paper.pdf .

8. Ronald L. Rivest, *The MD5 Message-Digest Algorithm*, 1992, ftp://ftp.umbc.edu/pub/unix/rfc/rfc1321.txt.gz .

9. Ross Anderson, Eli Biham, Lars Knudsen, *Serpent: A Proposal for the Advanced Encryption Standard*, 1999 (??), http://www.cl.cam.ac.uk/ftp/users/rja14/serpent.pdf .

10. Hans Dobbertin, Antoon Bosselaers, Bart Preneel, *RIPEMD-160: A Strengthened Version of RIPEMD*, 1996, http://www.esat.kuleuven.ac.be/~cosicart/pdf/AB-9601/AB-9601.pdf .

11. Carlisle Adams, *The CAST-256 Encryption Algorithm*, 1997, http://www.mirrors.wiretapped.net/security/cryptography/algorithms/cast/cast-256.pdf .

12. Yuliang Zheng, Josef Pieprzyk, Jennifer Seberry, *HAVAL — A One-Way Hashing Algorithm with Variable Length of Output*, Advances in Cryptology — AUSCRYPT'92," Lecture Notes in Computer Science, Vol.718, pp.83-104, Springer-Verlag, 1993, http://www.calyptix.com/files/haval-paper.pdf .

13. Antoine Joux, *Multicollisions in iterated hash functions, application to cascaded constructions* - Crypto 04, LNCS 3152, pp. 306 - 316