The Hash Function Cheetah: Specification and Supporting Documentation

Dmitry Khovratovich, Alex Biryukov, Ivica Nikolic

University of Luxembourg

October 31, 2008

Contents

1 Specification

1.1 Preliminaries

The Galois field used in Cheetah is defined as a set of polynomials of degree up to 7 with coefficients from $\mathbb{F}_2$. The multiplication is performed modulo the following irreducible polynomial:

$$m(x) = x^8 + x^4 + x^3 + x + 1. \quad (1)$$

The hexadecimal and binary representations of the field elements are also used in this proposal. E.g., the polynomial x^6+x^3+x+1 is defined as $'01001011'$ in the binary and $'47'$ in the hexadecimal representation, respectively.

The addition is the bitwise xor throughout this paper, except for loop and array indices.

1.2 Hash function

Cheetah is a set of hash functions each dedicated to a particular digest length. The supported digests are of size 224, 256, 384, and 512 bits, which are produced by Cheetah-224, Cheetah-256, Cheetah-384, and Cheetah-512, respectively.

Cheetah supports *salt*, a method for randomized hashing, which prevents a few types of generic attacks (see Section 3). The salt is a randomly chosen parameter, which shall be fixed just before hashing. The salt, if used, is a 128-bit value in Cheetah-224 and Cheetah-256, and a 256-bit value in Cheetah-384 and Cheetah-512. If salt is used it is inserted to each message block. Since it affects performance we propose two separate procedures for hashing: with and without salting.

Let us define the message padding using the size s of salt as a parameter (0 bits stand for hashing without salt). A message to be hashed is padded to the multiple of $(1024 - s)$ bits using the following procedure. First, bit 1 is appended to the end, and bits 0 are appended. Then the message length (8-byte value) and the digest size (2-byte value) are added. Both values are treated in the big-endian model. The number of zero bits is minimal such that the resulting bit length is divisible by $(1024 - s)$. The padded message is divided to $n + 1$ $(1024 - s)$-bit blocks. Each block is concatenated with the salt value if it is used. With respect to the salt size, the padding defines procedures FreshPadding and SaltyPadding, which are a part of the Cheetah pseudo-code (see below).

The compression function is the core of the Cheetah hash function, which takes the current message block and its index (8-byte value), the intermediate hash value, and the salt (optionally) as an input and outputs the next intermediate hash value. The first hash value is set to zero by default. After the last message block has been processed the resulting value is used to produce the final digest. This can be summarized in the following pseudo-code:

```
Cheetah_Fresh_Hash(Message, InitialValue) {
    MessageBlocks[0..n] = FreshPadding(Message);
    IntermediateHashValue = InitialValue;
    for(i=0; i<n; i++)
        IntermediateHashValue +=
            CheetahCompress(IntermediateHashValue, MessageBlock[i], i);
    LastBlockPermutation(IntermediateHashValue);
    IntermediateHashValue +=
        CheetahCompress(IntermediateHashValue, MessageBlock[n], n);
    if(Cheetah-224 or Cheetah-384)
        return Truncate(IntermediateHashValue);
    else
        return IntermediateHashValue;
}

Cheetah_Salty_Hash(Message, Salt, InitialValue) {
```

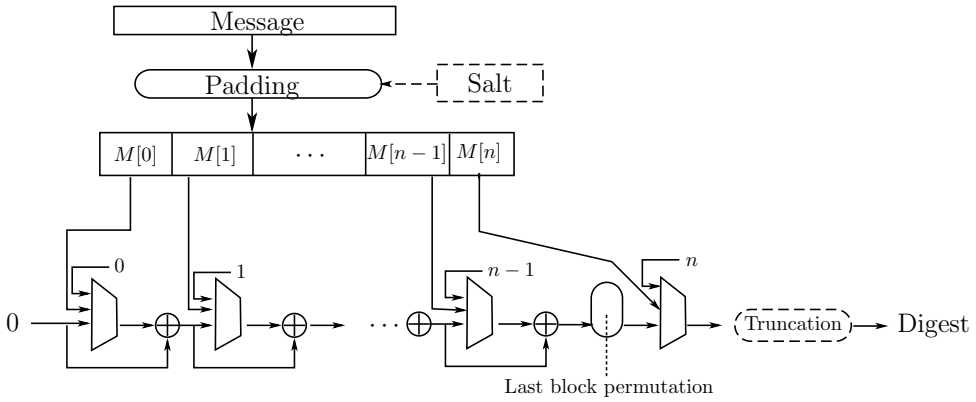

Figure 1: The outline of Cheetah.

```
    MessageBlocks[0..n] = SaltyPadding(Message,Salt);
    IntermediateHashValue = InitialValue;
    for(i=0; i<n; i++)
        IntermediateHashValue +=
            CheetahCompress(IntermediateHashValue, MessageBlock[i], i);
    LastBlockPermutation(IntermediateHashValue);
    IntermediateHashValue +=
        CheetahCompress(IntermediateHashValue, MessageBlock[n], n);
    if(Cheetah-224 or Cheetah-384)
        return Truncate(IntermediateHashValue);
    else
        return IntermediateHashValue;
}
```

The versions of Cheetah differ in these functions as follows. First, the InitialValue and the IntermediateHashValue are 32-byte (256-bit) vectors in Cheetah-224 and Cheetah-256, and 64-byte (512-bit) vectors in Cheetah-384 and Cheetah-512. The difference in the salt length has been described above. The CheetahCompress function is specified in Section 1.3.

The procedures FreshPadding and SaltyPadding are specified above. The LastBlockPermutation function swaps the first and the second halves of the IntermediateHashValue, i. e. column 0 is swapped with column 4, column 1 with column 5, column 2 with column 6, column 3 with column 7.

Finally, the Truncate function removes the last 32 bits of the input in Cheetah-224 and the last 128 bits of the input in Cheetah-384.

1.3 Compression function

The Cheetah compression function is an iterative transformation based on the Rijndael block cipher. The size of the internal state (S_I) and the number of rounds (N_r) varies for different versions of Cheetah, based on the output digest size.

Digest size	Internal state	Message block	Internal rounds	Message schedule rounds
224	256	1024	16	3
256	256	1024	16	3
384	512	1024	12	5
512	512	1024	12	5

The message block is expanded by the means of the message schedule. The expanded message is divided into N_r vectors, which are xored to the internal state before every round. The Cheetah compression function is then defined by the following pseudo-code:

```
CheetahCompression(IntermediateHashValue, MessageBlock,
BlockCounter) {
    InternalState = IntermediateHashValue + BlockCounter;
    ExpandedBlock = MessageExpansion(MessageBlock);
    for(i=1; i<= Nr; i++)
    {
        InternalState +=RoundBlock(ExpandedBlock,i);
        InternalState = InternalRound(InternalState);
    }
    return InternalState + BlockCounter;
}
```

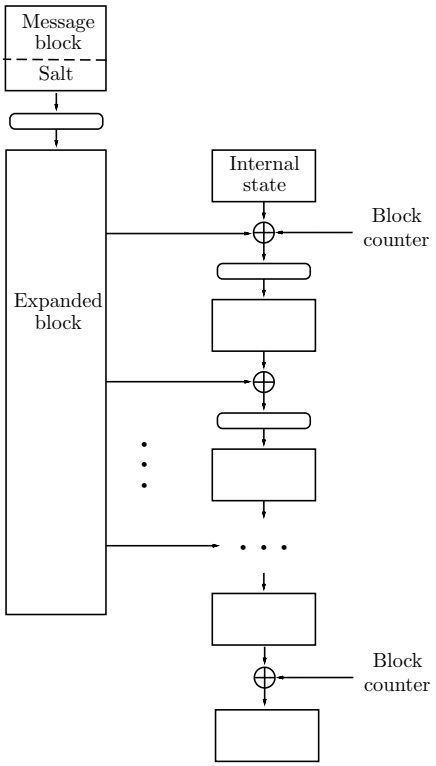

Figure 2: The outline of the compression function.

The block counter B is treated as an 8-byte vector in the big-endian model and is xored to the first 8 bytes of the IntermediateHashValue IV:

$$IV_j \leftarrow IV_j + B_j,\ 0 \le j \le 7.$$

The block counter is also added at the end of the compression function in the same way.

The procedures *MessageExpansion* and *InternalRound* are determined below for every version of Cheetah.

1.3.1 Cheetah-224 and Cheetah-256

The Cheetah-224 and Cheetah-256 use the same compression function. The 128-byte message block is expanded to a 512-byte block. The internal state is of size 32 bytes and is iterated for 16 rounds.

Message Schedule. The *MessageExpansion* procedure is a Rijndael-like transformation, which is defined in pseudocode as follows:

```
MessageExpansion(byte MessageBlock[128]) {
    byte ExpandedBlock[512];
    ExpandedBlock[0..127] = MessageBlock;
    for(i=1; i<=3; i++)
    {
        SubBytes(MessageBlock);
        ShiftRows8(MessageBlock);
        MixColumn8(MessageBlock);
        AddRoundConstant(MessageBlock,i);
        ExpandedBlock[128*i..128*i+127] = MessageBlock;
    }
}
```

MessageBlock is treated as a byte array of size 8×16:

$m_{0,0}$	$m_{0,1}$	$\cdots$	$m_{0,15}$
$m_{1,0}$	$m_{1,1}$	$\cdots$	$m_{1,15}$
$\cdots\cdots\cdots\cdots\cdots\cdots\cdots$			
$m_{7,0}$	$m_{7,1}$	$\cdots$	$m_{7,15}$

The original 128-byte vector $(v_0, v_1, \ldots, v_{127})$ is arranged into the array using the following rule[1]: $m_{i,j} = v_{8*i+7-j}$ (see).

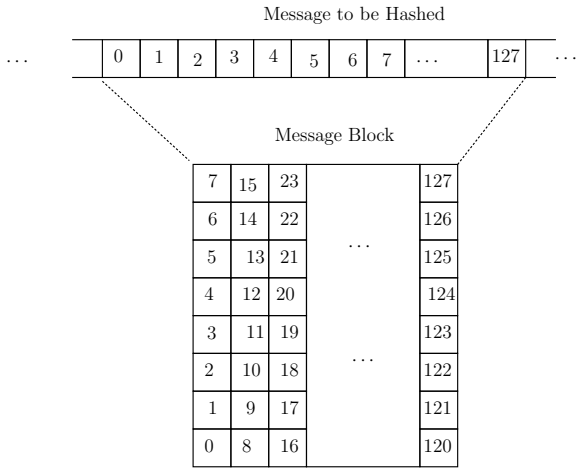

Figure 3: Byte arrangement.

[1]This arrangement is optimized for the little-endian architecture.

The *SubBytes* transformation is the byte-wise SubBytes transformation used in Rijndael:

$$S(X_1X_2) = Y$$

$S:$ $X_1\backslash X_2$	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	63	$7c$	77	$7b$	$f2$	$6b$	$6f$	$c5$	30	01	67	$2b$	fe	$d7$	ab	76
1	ca	82	$c9$	$7d$	fa	59	47	$f0$	ad	$d4$	$a2$	af	$9c$	$a4$	72	$c0$
2	$b7$	fd	93	26	36	$3f$	$f7$	cc	34	$a5$	$e5$	$f1$	71	$d8$	31	15
3	04	$c7$	23	$c3$	18	96	05	$9a$	07	12	80	$e2$	eb	27	$b2$	75
4	09	83	$2c$	$1a$	$1b$	$6e$	$5a$	$a0$	52	$3b$	$d6$	$b3$	29	$e3$	$2f$	84
5	53	$d1$	00	ed	20	fc	$b1$	$5b$	$6a$	cb	be	39	$4a$	$4c$	58	cf
6	$d0$	ef	aa	fb	43	$4d$	33	85	45	$f9$	02	$7f$	50	$3c$	$9f$	$a8$
7	51	$a3$	40	$8f$	92	$9d$	38	$f5$	bc	$b6$	da	21	10	ff	$f3$	$d2$
8	cd	$0c$	13	ec	$5f$	97	44	17	$c4$	$a7$	$7e$	$3d$	64	$5d$	19	73
9	60	81	$4f$	dc	22	$2a$	90	88	46	ee	$b8$	14	de	$5e$	$0b$	db
a	$e0$	32	$3a$	$0a$	49	06	24	$5c$	$c2$	$d3$	ac	62	91	95	$e4$	79
b	$e7$	$c8$	37	$6d$	$8d$	$d5$	$4e$	$a9$	$6c$	56	$f4$	ea	65	$7a$	ae	08
c	ba	78	25	$2e$	$1c$	$a6$	$b4$	$c6$	$e8$	dd	74	$1f$	$4b$	bd	$8b$	$8a$
d	70	$3e$	$b5$	66	48	03	$f6$	$0e$	61	35	57	$b9$	86	$c1$	$1d$	$9e$
e	$e1$	$f8$	98	11	69	$d9$	$8e$	94	$9b$	$1e$	87	$e9$	ce	55	28	df
f	$8c$	$a1$	89	$0d$	bf	$e6$	42	68	41	99	$2d$	$0f$	$b0$	54	bb	16

Here and later the values are given in the hexadecimal representation.

The *ShiftRows8* operation processes each row of the MessageBlock array independently. A row is cyclically shifted to the left over the offset with respect to the row index:

$$m_{i,j}^{\text{new}} \leftarrow m_{i,(j+c_i)\%16}.$$

Row index i	Offset c_i
0	0
1	1
2	2
3	3
4	5
5	6
6	7
7	8

The *MixColumn8* operation processes each column of the the MessageBlock array independently. A column is treated as an 8-element vector $\overline{m}$ over GF(2^8) and is multiplied by the following matrix over GF(2^8):

$$\overline{m}^{\text{new}} \leftarrow A\overline{m}; \quad A = \begin{pmatrix} 02 & 0c & 06 & 08 & 01 & 04 & 01 & 01 \\ 01 & 02 & 0c & 06 & 08 & 01 & 04 & 01 \\ 01 & 01 & 02 & 0c & 06 & 08 & 01 & 04 \\ 04 & 01 & 01 & 02 & 0c & 06 & 08 & 01 \\ 01 & 04 & 01 & 01 & 02 & 0c & 06 & 08 \\ 08 & 01 & 04 & 01 & 01 & 02 & 0c & 06 \\ 06 & 08 & 01 & 04 & 01 & 01 & 02 & 0c \\ 0c & 06 & 08 & 01 & 04 & 01 & 01 & 02 \end{pmatrix}.$$

The multiplication is performed in GF(2^8).

The *AddRoundConstant* operation adds a 32-bit constant to the message block. The constant is a function of the round index r:

$$m_{i,0} += S[4 * r + i],\ 0 \le i \le 3,$$

where S stands for the S-box.

The *RoundBlock* operation selects a 32-byte block from the ExpandedBlock $= (E_0, E_1, E_2, E_3)$. Define the round index r as $r = 4l + m$, $0 \le l, m \le 3$. Then the selected block is the 4×8 byte array M_r, that is defined as follows:

$$M_r = (m_{i,j})^{4\times 8},\ E_l = (e_{i,j})^{8\times 16};\quad m_{i,j} = e_{4*(m\%2)+i,4*(m/2)+j}.$$

This selection is illustrated in Figure 4.

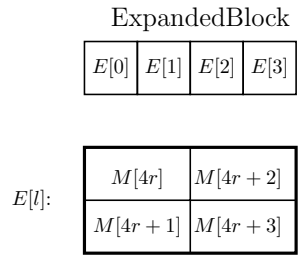

Figure 4: Composition of the expanded message.

The selected block is bytewise xored to the InternalState:

$$a_{i,j}^{\text{new}} \leftarrow a_{i,j} + m_{i,j}.$$

Internal round. The *InternalRound* transformation is actually the Rijndael round as it would be used with 32-byte block. It consists of three operations: *SubBytes*, *ShiftRows4*, and *MixColumn4*.

```
InternalRound(byte InternalState[256]) {
    SubBytes(InternalState);
    ShiftRows4(InternalState);
    MixColumn4(InternalState);
}
```

The *SubBytes* operation has already been defined above. Both the *ShiftRows4* and *MixColumn4* operations treat the InternalState as a byte array of size 4×8, with 4 rows and 8 columns:

$a_{0,0}$	$a_{0,1}$	$\cdots$	$a_{0,7}$
$a_{1,0}$	$a_{1,1}$	$\cdots$	$a_{1,7}$
$a_{2,0}$	$a_{2,1}$	$\cdots$	$a_{2,7}$
$a_{3,0}$	$a_{3,1}$	$\cdots$	$a_{3,7}$

The original 32-byte vector $(v_0, v_1, \ldots, v_{31})$ is arranged into the array using the following rule: $a_{i,j} = v_{4*i+3-j}$.

ShiftRows4 cyclically shifts each row with respect to the following offset table:

Row index i	Offset c_i
0	0
1	1
2	3
3	4

Thus the *ShiftRows4* operation can be expressed as follows:

$$a_{i,j}^{\text{new}} \leftarrow a_{i,(j+c_i)\%4}.$$

The *MixColumn4* transformation treats each column as a 4-element vector $\overline{a}$ over GF(2^8) and multiplies it by the following matrix:

$$B = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix}.$$

Thus *MixColumn4* can be expressed via the following formulae:

$$\overline{a}^{\text{new}} \leftarrow B\overline{a}.$$

Output. The output 32-byte vector $(v_0, v_1, \ldots, v_{31})$ is derived from the internal state using the following rule: $v_i = a_{i/4,\, 3-i\%4}$.

1.3.2 Cheetah-384 and Cheetah-512

The Cheetah-384 and Cheetah-512 use the same compression function. The 128-byte message block is expanded to a 1024-byte block. The internal state is of size 64 bytes and is iterated for 12 rounds.

Message Schedule. The *MessageExpansion* procedure is similar to that of Cheetah-224 and Cheetah-256. 5 rounds are required to get the blocks that are xored to the internal state.

```
MessageExpansion(byte MessageBlock[128]) {
    byte ExpandedBlock[768];
    ExpandedBlock[0..127] = MessageBlock;
    for(i=1; i<=5; i++)
    {
        SubBytes(MessageBlock);
        ShiftRows8(MessageBlock);
        MixColumn8(MessageBlock);
        AddRoundConstant(MessageBlock,i);
        ExpandedBlock[128*i..128*i+127] = MessageBlock;
    }
}
```

The *RoundBlock* operation selects a 64-byte block from ExpandedBlock. Define the round index r as $r = 2l + m$, $0 \le l \le 2$, $0 \le m \le 1$. Then the selected block is the 8×8 byte array M_r, that is defined as follows:

$$M_r = (m_{i,j})^{8\times 8},\ E_l = (e_{i,j})^{8\times 16};\quad m_{i,j} = e_{8*m+i,j}.$$

Internal round. The *InternalRound* transformation in Cheetah-384 and Cheetah-512 differs from that of smaller versions. Its operations resemble the *MessageExpansion* procedure, but the parameters are different:

```
InternalRound(byte InternalState[512]) {
    SubBytes(InternalState);
    ShiftRows64(InternalState);
    MixColumn64(InternalState);
}
```

All the transformations treat the input as a byte array of size 8×8:

$a_{0,0}$	$a_{1,0}$	$\cdots$	$a_{7,0}$
$a_{0,1}$	$a_{1,1}$	$\cdots$	$a_{7,1}$
.....			
$a_{0,7}$	$a_{1,7}$	$\cdots$	$a_{7,7}$

The original 64-byte vector $(v_0, v_1, \ldots, v_{63})$ is arranged to the array using the following rule: $a_{i,j} = v_{8*i+7-j}$.

The offset table of the *ShiftRows64* transformation:

Row index i	Offset c_i
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7

The matrix of the *MixColumn64* transformation:

$$A = \begin{pmatrix} 01 & 04 & 01 & 01 & 02 & 0c & 06 & 08 \\ 08 & 01 & 04 & 01 & 01 & 02 & 0c & 06 \\ 06 & 08 & 01 & 04 & 01 & 01 & 02 & 0c \\ 0c & 06 & 08 & 01 & 04 & 01 & 01 & 02 \\ 02 & 0c & 06 & 08 & 01 & 04 & 01 & 01 \\ 01 & 02 & 0c & 06 & 08 & 01 & 04 & 01 \\ 01 & 01 & 02 & 0c & 06 & 08 & 01 & 04 \\ 04 & 01 & 01 & 02 & 0c & 06 & 08 & 01 \end{pmatrix}.$$

Output. The output 64-byte vector $(v_0, v_1, \ldots, v_{64})$ is derived from the internal state using the following rule: $v_i = a_{i/8,\, 7-i\%8}$.

1.4 Cheetah as PRF and MAC

The Cheetah hash functions are suitable for the construction of HMAC [14]:

$$\mathrm{HMAC}_K(m) = h\big((K + c_1)||h((K + c_2)||m)\big),$$

where K is the key, m is the message to be hashed, h is the Cheetah hash function, $c_1 = 0x5c5c \ldots 5c$, $c_2 = 0x3636 \ldots 36$.

Cheetah can be used as a base of pseudo-random function (PRF) under different constructions, e.g., as HMAC-PRF.

1.5 Other digest sizes

The output of the Cheetah hash functions may be truncated to any digest size up to 512 bits though versions with digests smaller than of 160 bits can be broken by brute-force and thus not recommended. Thus we propose to truncate the output of Cheetah-256 in order to get digests from 160 to 256 bits or truncate the output of Cheetah-512 to get digests from 257 to 512 bits.

2 Design rationale

2.1 Hash function

The Cheetah design is the improved Merkle-Damgard construction [6, 16], which is now resistant to the generic attacks that appeared recently. The countermeasures we use have been summarized in a recently proposed HAIFA framework [2]. Salt is used to prevent generic attacks mounted off-line. As a result, the attacks on real applications that require a lot of precomputations have to be mounted only after the salt has been chosen by a user.

Since salt can be chosen independently of the message it should be carefully mixed with the hash value and the "salt schedule" must not be exploited. Thus we decided to insert salt to each message block so that message and salt are treated in the same way, and the analysis becomes easier.

We also added the permutation before the last message block is processed in order to prevent length-extension attacks (see Section 3).

2.2 Compression function

The design of the Cheetah compression function is inspired by AES [5]. We use the standard Davies-Meyer approach [19] to build a compression function from a block cipher: the key becomes the message, the plaintext is the IV, the ciphertext is the hash value. The feedforward operation is also applied. In order to fit the digest length the internal state has been enlarged. The size of the internal state is now 256 or 512 bits with respect to the desired digest size.

The original key schedule of Rijndael [5], which might become the message schedule in the Rijndael-based compression function, seems to be rather weak for our purpose. We made a modified message schedule somewhat similar to that of Rijndael so that diffusion is quite fast and trails with small number of byte differences are impossible (see the next section). However, the performance would substantially drop if a message block were of the same size as the internal state. That is why we use a larger message block.

Now we explain this in more details. The internal state is chosen as a multiple of 256 bits such that it can be used directly as the resulting hash value or should be truncated. The message block is a 1024-bit state. The smaller message block would reduce the speed, the bigger one might require tables that would not fit the first-level cache of some CPU.

Since the internal state is bigger than that of AES more rounds are required for good diffusion. The exact number of rounds has been determined by the message schedule. The message schedule should provide full diffusion: difference in one byte in the message block should affect all bytes in the last block of the expanded message. This requires 3 rounds of message schedule. Thus 4096 bits should be added to the internal state, which is of size 256 bit for Cheetah-224 and Cheetah-256. As a result, 16 rounds are required. The Cheetah-384 and Cheetah-512 versions have more thorough message schedule and faster diffusion so fewer rounds are enough. We use 12 rounds.

2.2.1 Round primitives

Irreducible polynomial. To the best of our knowledge there is no attack motivated by the choice of an irreducible polynomial used for creating the Galois field. The polynomial defined in (1) was used in the design of Rijndael [5].

ShiftRows offsets. When a state is treated as a matrix with 4 rows and 8 columns we use the offset values defined in the s pecification of Rijndael, which provide the best diffusion for such a matrix [5]. When a state is represented by a matrix with 8 rows and 8 or 16 columns we use the offset values which provide full diffusion after 3 rounds.

Diffusion matrices. In order to get the best possible diffusion we use MDS matrices, which provide the maximum branch number. Although the choice of the matrix does not influence

the software implementation on a PC[2], it is of importance on low-cost platforms. We used two additional criteria: the number of ones and the number of distinct elements in the matrix. These criteria affect the number of XOR operations and table lookups in the implementation on, e.g., 8-bit platforms [10].

The 4×4 matrix follows the design of Rijndael. The 8×8 matrix was first proposed by the designers of Grindahl [13]. The latter matrix is optimal in terms of the number of ones (24) and close to optimal in terms of the number of distinct elements (6). For more details we refer to [10].

Message schedule. We avoided possible weaknesses of the Rijndael-like key schedule by the use of a block cipher for the message expansion. Although three rounds provide full diffusion the versions Cheetah-384 and Cheetah-512 have more message rounds in order to make 12 message injections to the internal state.

We also a bit modified the 8×8 matrix for the message schedule in order to avoid possible attacks based on their identity. The MixColumn matrix used in the message schedule is obtained by that of Cheetah-512 internal transformation by swapping the rows.

2.3 Number of rounds as tunable parameter

Although we fix the number of internal rounds for all the versions of Cheetah, it can be changed in order to get a better security or performance. While we do not expect that the current versions are weaker than versions with more rounds, reduced-round versions may suffer from attacks. However, we believe that Cheetah-224 and Cheetah-256 with 12 rounds are as strong as Cheetah-256 truncated to 160 bits. We also believe that Cheetah-384 and Cheetah-512 with 10 rounds are as strong as Cheetah-256. We do not recommend to reduce the number of rounds further because Rijndael-specific properties (such as the Square property) might be exploited for theoretical attacks.

3 Resistance to attacks

3.1 Generic attacks

3.1.1 Herding attacks

The herding attacks were proposed by Kelsey and Kohno [11]. The attacker presents a hash value before she gets the information that is converted to the hash value. After the information is known she seeks a "garbage" string that should be added so that the resulting hash value coincides with the declared one. The amount of strings to be tried can be significantly reduced if the hash value is a kind of a multicollision: 2^t different IVs can be "herded" to the hash value under appropriate messages. If the multicollision is organized in a form of so called *diamond structure*, the whole attack costs less than the exhaustive search.

If an application uses salt then the attack requires the salt value to build the diamond structure. If salt is at least an $n/2$-bit value (n — size of the internal state) there is no benefit from trying all possible salts.

We also claim that the diamond structure can not be easily rebuilt for another salt value. This is derived from the fact that there is no direct evidence how to build pseudo-collisions with fixed IV and random message and salt.

3.1.2 Fast second preimage attack

A generic second preimage attack on the Merkle-Damgard construction was proposed by Kelsey and Schneier [12]. The key idea is to use so called *expandable messages*. As soon as the attacker finds two collided expandable messages of different length, collisions with messages of the same size can be derived fast. As a result, the padding of the message with its length does not prevent from second preimage attacks.

[2] An optimized implementation treats the whole round function as a sum of precomputed functions.

The attack is prevented in our construction by the use of the block counter in each iteration of the compression function. Since the underlying Rijndael in the compression function is invertible, the compression function as a function of block counter as an argument (the other input is constant) is an injection. As a result, an expandable message becomes not-expandable unless more sophisticated ideas are proposed. So far there is no evidence how to carry out the Kelsey-Schneier attack to the construction with the block counter.

3.1.3 Multicollisions

The multicollision attacks were proposed by Joux [9]. The idea is to build collisions one after another, which leads to 2^k colliding messages after only k trials of the collision search. If a hash function has an iterative structure, the attack can be always maintained. The actual complexity, however, depends on the size of the internal state. Since the internal state of versions Cheetah-256 and Cheetah-512 is of the same size as the corresponding digests these are susceptible to multicollisions. However, the real threat is the use of multicollisions in the fast second preimage search (see above), but such attack is prevented by the use of block counter.

3.1.4 Length-extension attacks

The well-known weakness of the original Merkle-Damgard construction is a length extension property: if digests of messages M and M' collide than adding a common suffix also lead to a collision: $h(M||\overline{M}) = h(M'||\overline{M})$ (see [1, 7] for more details). Even if we concatenate an original message with its length ($M' = M||\text{length}(M)$) the attack is valid because M' can be itself considered as an extendable message.

We prevent this attack by applying a trivial permutation to the intermediate value before the last message block is processed (the idea was proposed in [8]). If there is a collision after the last iteration of the compression function then adding any data to the end of colliding message will change the input of the compression function that led to a collision.

3.2 Attacks on the compression function

3.2.1 Collisions

Most collision attacks on dedicated hash functions are based on differential cryptanalysis, which originates from block ciphers. The idea is to consider a differential trail (or path, or characteristic) — a sequence of differences in the internal state throughout the iteration (or several iterations) of the compression function. If the trail ends with zero difference then any message pair, such that the two iterations provide the desired difference during the iteration, gives a collision. The workload of the attack is usually determined by the round probabilities, or the proportion of pairs which gives the output difference providing the input difference. The probabilities are partly compensated by the freedom in the choice of a message pair.

Attacks on AES [5] and AES-based hash functions (such as Grindahl [18]) deal mostly with XOR differences. The differences form either the 256-value set (all possible 8-bit values) or 2-value set (either zero and non-zero difference or zero and any difference). We denote these attacks by the attack with standard differentials and the attack with truncated differentials, respectively.

Now we claim that any differential trail in the Cheetah compression function has very low probability and provide some observations in support of this claim.

Standard differentials. The only non-linear transformation in Cheetahis the SubBytes transformation. The probability that a non-zero byte difference δ_1 is converted by the S-box to a difference δ_2 is upper bounded by 2^{-6}. Thus upper bound on the probability of a differential trail is determined by the number of non-zero differences entering S-boxes, or the number of *active S-boxes*.

The following observation gives us the minimum number of active S-boxes in a one-block differential trail for Cheetah-256. Such a trail has zero difference in the input and in the output of

the compression function. Let us denote the number of non-zero differences in InternalState before the SubBytes transformation by s_i, $1 \le i \le 16$. The last s_i is equal to zero. Let us also denote by c_i the number of non-zero differences in InternalState after the MixColumn4 transformation. The last c_i is equal to 0 as well. Finally, we denote by m_i the number of non-zero differences in RoundBlock that is xored to InternalState in the beginning of a round. These differences either cancel non-zero differences in InternalState or create them. Thus the following condition holds

$$s_i + c_{i-1} \ge m_i \tag{2}$$

Due to the branch number of the MixColumn4 transformation c_i is upper bounded: $c_i \le 4s_i$. Thus we obtain the following:

$$s_i + 4s_{i-1} \ge m_i \;\Rightarrow\; \sum_i s_i + \sum_i 4s_{i-1} \ge \sum_i m_i \;\Rightarrow\; S \ge \frac{M}{5},$$

where S is the number of active S-boxes in the internal state of the compression function, and M is the number of non-zero byte differences in ExpandedBlock.

Now we estimate the minimum number of non-zero byte differences in the message scheduling only. First we note that this number is equal to the number of the active S-boxes in the message scheduling extended to 4 rounds. Such a 4-round transformation is actually a Rijndael-like block cipher, which can be investigated using the theory proposed by Daemen and Rijmen [4].

They estimated the minimum number of active S-boxes in 4 rounds of a Rijndael-like block cipher. The sufficient condition to apply their theorem is that the ShiftRows8 should be diffusion optimal: bytes from a single column should be distributed to different columns, which is the case. Thus the number of active S-boxes can be estimated as the square of the MixColumn8 branch number (9). As a result, any pair of different message blocks has difference in at least $M = 81$ bytes of ExpandedBlock. This implies the lower bound 17 for S.

Thus we obtain the following proposition.

Proposition 1. *Any 1-block collision trail of Cheetah-224 and Cheetah-256 has at least 17 active S-boxes in the internal state.*

Though any real trail with only 17 active S-boxes would probably lead to a collision attack, we expect that the values of M close to minimal do not give actual collision trails due to the following reasons:

- Small number of active S-boxes in the internal state implicitly assumes many local collisions;
- The distribution of non-zero differences in the message scheduling is not suitable for local collisions due to high diffusion;
- The MixColumn matrix in the message scheduling differs from that of the internal transformation so, e.g., 4-byte difference collapse to 1-byte difference may only happen in one of the two transformations.

Truncated differentials. The complexity of collision search with truncated differentials is estimated in a different way. Recall that we deal with two types of byte differences: zero difference and non-zero difference.

Consider such a trail. While the SubBytes transformation keeps these values unchanged the additions can modify them. More precisely, addition cancels the non-zero difference with probability 2^{-8}. Every zero difference in an active column after the MixColumn transformation costs the same. The multiple of these probabilities gives the trail probability, which can be treated in several ways with respect to the type of the attack. Generally, the lower the probability is the harder the attack is. Thus we try to compute an upper bound on the probability, or the lower bound on the *byte weight* — the number of non-deterministic non-zero to zero transformations.

Let us estimate first the *round weight* p_i — the number of the zero differences in the internal state that are the result of linear transformation of non-zero differences. Here i stands for the

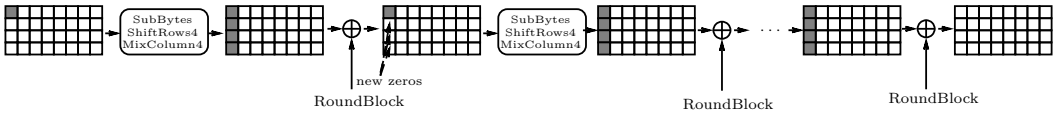

Figure 5: An example of an optimal subtrail.

round index. Two effects contribute to p_i: zero differences after MixColumn and zero difference due to the injection of RoundBlock.

The idea is to consider various possibilities for the number k_i of active columns in the MixColumn transformation in round i. Indeed, non-active columns in round $i+1$ are equivalent to diagonal zeros after round i. If these diagonals intersect with active columns in round i then the resulting number of zeros in the intersection contribute to p_i. One can make the following table, which lists the minimum number of p_i as a function of k_i and k_{i+1} for Cheetah-256:

$k_i \backslash k_{i+1}$	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	4	3	2	1	0	0	0	0	0
2	8	6	4	3	2	1	0	0	0
3	12	9	7	5	3	2	1	0	0
4	16	12	10	7	4	3	2	0	0
5	20	16	13	10	7	5	3	1	0
6	24	20	16	13	10	7	4	2	0
7	28	24	20	16	12	9	6	3	0
8	32	28	24	20	16	12	8	4	0

Table 1: Byte weight of round transformation with truncated differentials as a function of the number of active columns (Cheetah-256).

Let us now estimate the probability of a k-round trail with truncated differentials. First, we introduce the notion of a *subtrail*. A subtrail is a trail such that it covers only internal states, the first and the last internal state have zero difference, and the other internal states have at least one active column. Using Table 1 we obtain the lower bounds for subtrails. They are given in Table 2.

Rounds	2	3	4	5	6	7	8	9	10	11	12−
Weight	4	7	10	13	16	19	22	25	28	31	32

Table 2: Minimum subtrail weight for Cheetah-256.

One may notice that the weight of a k-round subtrail is bounded by $3k-2$. An example of such subtrail is provided in Figure 5.

However, a good trail may be composed of several subtrails. More precisely, internal state may have zero difference in the middle of the trail, not only at rims. It is unlikely that two such states may appear consecutively due to good diffusion properties in message schedule. Assuming that we obtain that a k-round trail has at least weight $2k$.

In our observations we did not consider trails in message schedule. We expect that any full trail (message schedule+internal transformation) has worse properties than the corresponding internal trail due to the following:

- Trails in the message schedule are likely to have low probability;

- Good message trails may impose bad internal ones, and vice versa — due to different round functions.

The same observations hold for Cheetah-384 and Cheetah-512 though weight values in Tables 1 and 2 are different (Table 3).

Rounds	2	3	4	5	6	7	8	9	10−
Weight	8	15	22	29	36	45	52	59	64

Table 3: Minimum subtrail weight for Cheetah-512.

Thus the weight of an k-round subtrail can be lower bounded by $7k - 6$. An internal k-round trail with no two consecutive zero-difference states has weight at least $4k$.

3.2.2 Preimages and second preimages

Only few successful preimage attacks have been introduced so far, and most of them are specific for a concrete compression function. Since the compression function of Cheetah is based on Rijndael, the preimage search is equivalent to the key recovery attack on the corresponding block cipher (adding feedforward does not make the attack significantly faster). Due to improved message schedule and more rounds, we expect that Cheetah is resistant to preimage attacks as Rijndael is. The same observation holds for second preimage attacks.

We also claim that the strong message schedule we use prevents from the attacks similar to recent preimage attacks on SHA [3], where slow diffusion was exploited.

3.3 HMAC-PRF security

It is well known that the HMAC is a PRF if the underlying compression function is a PRF, and the iterated hash function is weakly collision-resistant [1]. We claim that the Cheetah compression function is a PRF since it is based on Rijndael, and there is no distinguishing attacks on Rijndael with > 10 rounds. Secondly, the Cheetah compression function is considered weakly collision-resistant since it is claimed to be collision-resistant (Section 3.2.1).

3.4 Randomized hashing attack

The randomized hashing attack is defined in [17] as follows. The attacker chooses a message, M_1. The specified construct is then used on M_1 with a randomization value r_1 that has been randomly chosen without the attackers control after the attacker has supplied M_1. Given r_1, the attacker then attempts to find a second message M_2 and randomization value r_2 that yield the same randomized hash value.

The randomization value is actually the salt value in Cheetah. Thus the attack can be reformulated as follows: the attacker choose a first message, then the first salt value is given, and he must find another pair (message, salt) that yield the same hash value. Thus the attack is the second preimage search for the hash digest obtained after the first pair is defined. Since we expect that the compression function is resistant to second preimage attack for any hashed message we claim that the Cheetah is resistant to the randomized hashing attack as well.

3.5 Security level

The claimed security level for Cheetah-N is outlined in Tables 4.

4 Advantages and limitations

We claim that Cheetah has the following advantages:

- It is fast and significantly faster than SHA-2.
- It is based on Rijndael so many tricks designed for implementations of Rijndael and AES can be used for Cheetah.
- It supports salt and thus randomized hashing.
- Lower bounds for the differential trail parameters are given thus providing support for the resistance of Cheetah to the most powerful collision attacks.
- Generic attacks on the Merkle-Damgard construction are prevented by using block counter and salt.

However, we expect that Cheetah may have the following limitations in some applications:

- It is byte-oriented but the best speed is achieved on 32-bit architecture or more advanced one.
- Cheetah-224 and Cheetah-384 have the same speed as Cheetah-256 and Cheetah-512, respectively.

5 Software and hardware implementation

5.1 Software

Cheetahcan be seen as two instances of Rijndael running in parallel. The first instance, Rijndael-256, is used in the compression function to update the state of the function, which is 4×8 matrix of bytes. The second instance, which transforms the much bigger matrix 8×16, is used for the message expansion. Hence, for all of the transformations of Cheetah, software implementation tricks of Rijndael can be used. It is well known that SubBytes, ShiftRows, and MixColumns can be implemented as simple table lookups and xors. Let us give an estimate for the efficiency of Cheetah-256. Later, an estimate for Cheetah-512 will be given. One round of the compression function of Cheetah-256, which works with 4×8 matrix, requires 32 table look-ups, and 32 xors. Hence, the whole 16 rounds will take 512 table look-ups, and 512 xors. The message expansion, on 32-bit platforms, needs 256 table look-ups and 224 xors for one round. In the compression function, three rounds of this message expansion are required. Therefore, only for the message expansion 768 table look-ups and 672 xors are needed. So for the whole compression function of Cheetah-256, one needs 1280 table look-ups and 1184 xors. This effort is spent to process a 128 bytes of message. Let us try to compare the speed of Cheetah-256 with the speed of AES (Rijndael-128). AES uses 4 table look-ups and 4 xors per column per round. Therefore for processing 16

	Attack complexity				Supporting rationale
Digest size (bits)	224	256	384	512	
HMAC-PRF distinguisher	2^{112}	2^{128}	2^{192}	2^{256}	Sec. 3.3
Randomized hashing attack	2^{224}	2^{256}	2^{384}	2^{512}	Sec. 3.4
Collision search	2^{112}	2^{128}	2^{192}	2^{256}	Sec. 3.2.1
Preimage search	2^{224}	2^{256}	2^{384}	2^{512}	Sec. 3.2.2
Second-preimage search for a message shorter than 2^k bits	2^{224}	2^{256}	2^{384}	2^{512}	Sec. 3.1.2, Sec. 3.2.2
Length-extension attacks	Invulnerable				Sec. 3.1.4

Table 4: Security claims for Cheetah.

bytes, AES uses 160 table look-ups and 160 xors. For processing 128 bytes, AES requires 1200 table look-ups and 1200 xors. Hence, the speed of Cheetah and AES are almost the same.

The optimal implementation of Cheetah requires 4KB for the tables and 160 bytes for the internal value and the message expansion. Hence, in total, it requires around 4 KB of memory. Memory speed trade-offs are possible. Since the look-up tables are rotated versions of each other, an implementation with only one table, and a total memory of around 1KB is achievable.

Cheetah-512. The longer digest functions are best implemented on 64-bit platforms. Optimal implementation of Cheetah-512 requires 16KB for 8 look-up table with 2KB each. The same trade-off is possible as for Cheetah-256. An implementation with only 2KB of memory is possible.

The summary of the Cheetah speed is given in Tables 5, 6, 7, and 8.

Platform: Intel Core 2 Duo 32-bit, 2.4 GHz, 2 GB RAM, Windows XP.

	Standard				Memory optimized	
Digest size (bits)	224	256	384	512	256	512
One message digest	2464	2464	5472	5472	3520	7744
Algorithm setup	32	32	64	64	32	64

Table 5: Estimates on NIST Reference Platform (32 bit).

Platform: Intel Core 2 Duo 64-bit, 2.66 GHz, 4 GB RAM, Windows XP x64.

	Standard				Memory optimized	
Digest size (bits)	224	256	384	512	256	512
One message digest	1744	1744	2736	2736	2464	3872
Algorithm setup	32	32	64	64	32	64

Table 6: Estimates on NIST Reference Platform (64 bit).

Platform: Atmel 8-bit AVR ATmega128, 16 MHz, 128Kbytes of flash program memory and 4K SRAM.

Digest size (bits)	224	256	384	512
One message digest	16768	16768	26368	26368
Algorithm setup	32	32	64	64

Table 7: Estimates on 8-bit processors.

Digest size (bits)	224	256	384	512
32-bit	15.3	15.3	83.8	83.8
64-bit	10.5	10.5	15.6	15.6
8-bit	131	131	206	206

Table 8: Speed in cycles/byte (optimized implementation).

5.2 Hardware

Since there are many hardware architectures the gate complexity of crypto implementations varies dramatically. Speak generally, there is a tradeoff between gate count and speed. Moreover, special tricks can drastically reduce the gate complexity as compared to the reference implementation. As a result, AES can be implemented in 10^6 gates as well as in $1.7 \cdot 10^5$ gates [15]. Taking the latter result as a reference one, we can roughly estimate the hardware requirements for Cheetah.

Cheetah-256 has an internal state, which is twice bigger than that of AES-128. Providing the number of rounds be equal to 16, we derive that the internal operations should require three times bigger hardware implementation than those of AES. Analogously, the message expansion is roughly equal in the number of operations to the sum of internal rounds so we estimate its gate count as the same as of the latter one. Finally, we estimate that an implementation of Cheetah-512 on an ASIC platform [15] would require about 600, 000 gates.

The same analysis for Cheetah-512 gives an estimate of 900, 000 gates.

References

[1] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In *CRYPTO*, pages 1–15, 1996.

[2] Eli Biham and Orr Dunkelman. A framework for iterative hash functions — HAIFA. *Cryptology ePrint Archive: Report 2007/278*, 2007.

[3] Christophe De Cannière and Christian Rechberger. Preimages for reduced sha-0 and sha-1. In *CRYPTO*, pages 179–202, 2008.

[4] Joan Daemen and Vincent Rijmen. The wide trail design strategy. In *IMA Int. Conf.*, pages 222–238, 2001.

[5] Joan Daemen and Vincent Rijmen. *The Design of Rijndael. AES — the Advanced Encryption Standard*. Springer, 2002.

[6] Ivan Damgård. A design principle for hash functions. In *CRYPTO'89*, volume 435 of *LNCS*, pages 416–427. Springer, 1989.

[7] Niels Ferguson and Bruce Schneier. *Practical Cryptography*. John Wiley & Sons, 2003.

[8] Shoichi Hirose, Je Hong Park, and Aaram Yun. A simple variant of the merkle-damgård scheme with a permutation. In *ASIACRYPT*, pages 113–129, 2007.

[9] Antoine Joux. Multicollisions in iterated hash functions. Application to cascaded constructions. In *CRYPTO'04*, volume 3152 of *LNCS*, pages 306–316. Springer, 2004.

[10] Pascal Junod and Serge Vaudenay. Perfect diffusion primitives for block ciphers. In *Selected Areas in Cryptography*, pages 84–99, 2004.

[11] John Kelsey and Tadayoshi Kohno. Herding hash functions and the Nostradamus attack. In *EUROCRYPT'06*, volume 4004 of *LNCS*, pages 183–200. Springer, 2006.

[12] John Kelsey and Bruce Schneier. Second preimages on n-bit hash functions for much less than 2^n work. In *EUROCRYPT'05*, volume 3494 of *LNCS*, pages 474–490. Springer, 2005.

[13] Lars R. Knudsen, Christian Rechberger, and Søren S. Thomsen. The Grindahl hash functions. In *FSE'07*, volume 4593 of *LNCS*, pages 39–57. Springer, 2007.

[14] H. Krawczyk, M. Bellare, and R. Canetti. *HMAC: Keyed-Hashing for Message Authentication, Request for Comments (RFC 2104)*. Internet Activities Board, Internet Privacy Task Force, 1997.

[15] Henry Kuo and Ingrid Verbauwhede. Architectural optimization for a 1.82gbits/sec vlsi implementation of the aes rijndael algorithm. In *CHES*, pages 51–64, 2001.

[16] Ralph C. Merkle. One way hash functions and DES. In *CRYPTO'89*, volume 435 of *LNCS*, pages 428–446. Springer, 1989.

[17] National Institute of Standards and Technology. *Announcing Request for Candidate Algorithm Nominations for a New Cryptographic Hash Algorithm (SHA3) Family*, volume 72, No. 212 of *Federal Register*. November 2007.

[18] Thomas Peyrin. Cryptanalysis of Grindahl. In *ASIACRYPT'07*, volume 4833 of *LNCS*, pages 551–567. Springer, 2007.

[19] Bart Preneel, René Govaerts, and Joos Vandewalle. Hash functions based on block ciphers: A synthetic approach. In *CRYPTO'93*, volume 558 of *LNCS*, pages 368–378. Springer, 1993.