CHAPTER 5

MNF-256: A 256-BIT PARALLEL STRUCTURED HASH FUNCTION BASED ON DITHER CONSTRUCTION

The most popular method of designing compression functions of dedicated hash functions is a serial successive iteration of a small step function, as like round functions of block ciphers. Many hash functions such as MD4 [95], MD5 [96], HAVAL [105], SHA-family [97, 98] etc. follow that idea. Attacks on serial hash functions have been focused on vanishing the difference of intermediate values caused by the difference of messages. On the other hand, a hash function has been considered secure if it is computationally hard to vanish such difference in its compression function. Usually, the lower the probability of the differential characteristic is, the harder the attack is. Therefore a step function is regarded as a good candidate if it causes a good avalanche effect in the serial structure. A function which has a good diffusion property cannot be so light in general. However, most step functions have been developed to be light for efficiency. This may be why MD4-type hash functions including SHA-1 are vulnerable to Wang et al.’s collision-finding attack. RIPEMD-family [163] has somewhat different approach for designing a secure hash function. The attacker who tries to break members of RIPEMD-family should aim simultaneously at two ways where the message difference passes. This design strategy is still successful because so far there is not any effective attack on RIPEMD-family except the first proposal of RIPEMD. However, RIPEMD-family has heavier compression functions than hash functions with serial structure. For example, the first proposal of RIPEMD consists of two lines of MD4. Total number of steps is twice as many as that of MD4. Also, the number of steps of RIPEMD-160 is almost twice as many as that of SHA-0. They were designed to have two parallel lines, which is different from MD4, MD5 and SHA-family. This makes an attacker take into account two lines simultaneously. However, since each line needs almost same operation of MD5 and SHA algorithms, its efficiency was degraded almost half of them. This motivated FORK-family designers. They use four lines instead of two. FORK-family’s construction can be seen as further extension of the design principle of two parallel lines used in RIPEMD [163]. In order to overcome disadvantage of RIPEMD algorithms, designers of FORK-family reduce operations for
step functions of each line. The message reordering of each branch is deliberately designed to be resistant against differential attacks. The function \( f \) and \( g \) in each step are chosen to have good avalanche effects.

In this chapter we propose a dedicated hash function MNF-256 based on the design principle of NewFORK-256. It iterates the compression function using dither construction in three parallel branches. Both FORK-256 and NewFORK-256 use 4 branches. The motive to have three branches in a proposed hash function is to reduce the computation time of hash value without compromising the security of existing hash functions. Dither construction assures additional security against generic and differential attacks.

The rest of this chapter is organized as follows; Section 5.1 presents the short description of hash functions from FORK-family and attacks on them. Section 5.2 contains rationale behind design choices of MNF-256. Section 5.3 describes formal algorithm of hash function MNF-256. Section 5.4 shows the collision resistance power of the hash function against different collision strategies. In section 5.5 we have analyzed the security of MNF-256 against the attack of Saarinen. The chapter is concluded in Section 5.5.

### 5.1 Short Description of FORK-family

FORK-256 is a first member of dedicated hash function family FORK. It is proposed by Hong et al. [102] with the particularity that it uses four parallel branches for the computation of the hash output. It is based on the classical Merkle-Damgård iterative construction with a compression function that maps 256 bits of state \( CV_n \) and 512-bits of message block \( M_i \) to 256-bits of a new state \( CV_{n+1} \). The compression function uses a set \( \{BRANCH_j\}_{j=1,2,3,4} \) of four branches running in parallel, each one of them using a different scheduling of sixteen 32-bit message blocks \( m_i, i = 0, \ldots, 15 \) by permuting them through \( \sigma_j(t) \). The same set of chaining variables \( CV_0 = (A_0, B_0, C_0, D_0, E_0, F_0, G_0, H_0) \) is used in the four branches. After computing outputs of parallel branches \( h_j = BRANCH_j(CV, M) \) the compression function updates the set of chaining variables according to the formula:

\[
CV_{i+1} = CV_i + [(h_1 + h_2) \oplus (h_3 + h_4)], \quad \ldots(5.1)
\]
where + and ⊕ are performed word-wise. Each branch function \( \text{BRANCH}_j \), \( j = 1, 2, 3, 4 \) consists of eight steps. In each step \( k = 0, \ldots, 7 \) the branch function updates its own copy of eight chaining variables using the step transformation. \( R_{j,k} \) denotes the value of the register \( R \in \{A, \ldots, H\} \) in \( j^\text{th} \) branch after step \( k \) and all \((A_{j,0}, \ldots, H_{j,0})\), are initialized with corresponding values of eight chaining variables \((A_0, \ldots, H_0)\).

\[
\begin{align*}
A_{j,k+1} &= H_{j,k} + g(E_{j,k} + m_{\sigma_j(2k+1)}) \lll 21 \oplus f(E_{j,k} + M_{\sigma_j(2k+1)} + \beta_{j,k}) \lll 17, \\
B_{j,k+1} &= A_{j,k} + m_{\sigma_j(2k)} + \alpha_{j,k}, \\
C_{j,k+1} &= B_{j,k} + f(A_{j,k} + m_{\sigma_j(2k)}) \oplus g(A_{j,k} + m_{\sigma_j(2k)} + \alpha_{j,k}), \\
D_{j,k+1} &= C_{j,k} + f(A_{j,k} + m_{\sigma_j(2k)}) \lll 5 \oplus g(A_{j,k} + m_{\sigma_j(2k)} + \alpha_{j,k}) \lll 9, \\
E_{j,k+1} &= D_{j,k} + f(A_{j,k} + m_{\sigma_j(2k)}) \lll 17 \oplus g(A_{j,k} + m_{\sigma_j(2k)} + \alpha_{j,k}) \lll 21. \\
F_{j,k+1} &= E_{j,k} + m_{\sigma_j(2k+1)} + \beta_{j,k}, \\
G_{j,k+1} &= F_{j,k} + g(E_{j,k} + m_{\sigma_j(2k+1)}) \oplus f(E_{j,k} + m_{\sigma_j(2k+1)} + \beta_{j,k}), \\
H_{j,k+1} &= G_{j,k} + g(E_{j,k} + m_{\sigma_j(2k+1)}) \lll 9 \oplus f(E_{j,k} + m_{\sigma_j(2k+1)} + \beta_{j,k}) \lll 5. \\
\end{align*}
\]

The functions \( f \) and \( g \) are defined as,

\[
\begin{align*}
f(x) &= x + (x \lll 7 \oplus x \lll 22), \\
g(x) &= x \oplus (x \lll 13 + x \lll 27).
\end{align*}
\]

The constants \((\sigma_1, \ldots, \sigma_{13})\) are given in Table 5.1 and permutations of message words and permutations of constants are defined in Table 5.2 and Table 5.3 respectively.

Matusiewicz, Contini, and Pieprzyk attacked FORK-256 by using the fact that the functions \( f \) and \( g \) in the step function were not bijective. They used microcollisions to find collisions of 2-branch FORK-256 and collisions of full FORK-256 with complexity of \( 2^{126.6} \) [169]. Four parallel branches of FORK-256 are operating on the same initial state and using the same blocks of messages but in a different order. This seems to be the strength of FORK-256 since the reported efforts to break it was limited to two of the four branches.
Table 5.1: Constants used in FORK-256.

<table>
<thead>
<tr>
<th>k</th>
<th>(\alpha_{1,k})</th>
<th>(\beta_{1,k})</th>
<th>(\alpha_{2,k})</th>
<th>(\beta_{2,k})</th>
<th>(\alpha_{3,k})</th>
<th>(\beta_{3,k})</th>
<th>(\alpha_{4,k})</th>
<th>(\beta_{4,k})</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(\delta_0)</td>
<td>(\delta_1)</td>
<td>(\delta_2)</td>
<td>(\delta_3)</td>
<td>(\delta_4)</td>
<td>(\delta_5)</td>
<td>(\delta_6)</td>
<td>(\delta_7)</td>
</tr>
<tr>
<td>1</td>
<td>(\delta_8)</td>
<td>(\delta_9)</td>
<td>(\delta_{10})</td>
<td>(\delta_{11})</td>
<td>(\delta_{12})</td>
<td>(\delta_{13})</td>
<td>(\delta_{14})</td>
<td>(\delta_{15})</td>
</tr>
<tr>
<td>2</td>
<td>(\delta_{16})</td>
<td>(\delta_{17})</td>
<td>(\delta_{18})</td>
<td>(\delta_{19})</td>
<td>(\delta_{20})</td>
<td>(\delta_{21})</td>
<td>(\delta_{22})</td>
<td>(\delta_{23})</td>
</tr>
<tr>
<td>3</td>
<td>(\delta_{24})</td>
<td>(\delta_{25})</td>
<td>(\delta_{26})</td>
<td>(\delta_{27})</td>
<td>(\delta_{28})</td>
<td>(\delta_{29})</td>
<td>(\delta_{30})</td>
<td>(\delta_{31})</td>
</tr>
<tr>
<td>4</td>
<td>(\delta_{32})</td>
<td>(\delta_{33})</td>
<td>(\delta_{34})</td>
<td>(\delta_{35})</td>
<td>(\delta_{36})</td>
<td>(\delta_{37})</td>
<td>(\delta_{38})</td>
<td>(\delta_{39})</td>
</tr>
<tr>
<td>5</td>
<td>(\delta_{40})</td>
<td>(\delta_{41})</td>
<td>(\delta_{42})</td>
<td>(\delta_{43})</td>
<td>(\delta_{44})</td>
<td>(\delta_{45})</td>
<td>(\delta_{46})</td>
<td>(\delta_{47})</td>
</tr>
<tr>
<td>6</td>
<td>(\delta_{48})</td>
<td>(\delta_{49})</td>
<td>(\delta_{50})</td>
<td>(\delta_{51})</td>
<td>(\delta_{52})</td>
<td>(\delta_{53})</td>
<td>(\delta_{54})</td>
<td>(\delta_{55})</td>
</tr>
<tr>
<td>7</td>
<td>(\delta_{56})</td>
<td>(\delta_{57})</td>
<td>(\delta_{58})</td>
<td>(\delta_{59})</td>
<td>(\delta_{60})</td>
<td>(\delta_{61})</td>
<td>(\delta_{62})</td>
<td>(\delta_{63})</td>
</tr>
</tbody>
</table>

Mendel, Lano, and Preneel [170] published the collision-finding attack on 2-branch FORK-256 using microcollisions and raised the possibility of its expansion. In other words, the main difficulty in cryptanalysis of FORK-256 comes from the fact that the same message blocks are input in each of the four branches in a permuted fashion. Thus, while one or maybe two branches may be easily dealt with, the effect of the difference is difficult to cancel in the remaining
branches. Matusiewicz et al. published another attack which finds a collision with complexity of $2^{108}$ at FSE 2007 [188].

In response to these attacks the authors of FORK-256 proposed in 2007 a new version of FORK-256 [103], which is supposedly resistant to all before mentioned attacks. The structure of NewFORK-256 is similar to FORK-256. The compression function of NewFORK-256 also consists of 4 parallel branch functions and each branch function consists of 8 sequential step functions. Message permutation, constants and their permutation used in NewFORK-256 are similar to FORK-256. Each step function has two different simple functions $f$ and $g$ with 32-bit inputs and outputs. In NewFORK-256, $f$ and $g$ are modified as follows,

$$f(x) = x \oplus (x \lll 15 \oplus x \lll 27),$$
$$g(x) = x \oplus (x \lll 7 + x \lll 25).$$

Especially, the function $f$ is changed from nonbijective to bijective. This change eliminates microcollisions in the step transformation, which have been crucial points of the attacks on FORK-256 in [169, 170]. Moreover, $f$ and $g$ propagate the difference of a message word to the chaining variables. Authors of NewFORK-256 [103] also modify the step function slightly. They have removed two additions and two XORs. They modified 4 shift rotations values used at step operations.

At Indocrypt’07 [171], Markku-Juhani O. Saarinen presented collision attack against NewFORK-256 using meet-in-the-middle technique. The complexity of this collision attack is $2^{112.8}$. Because of similarity in structure this attack is also applicable for FORK-256.

Both the hash functions from FORK-family were built on Merkle-Damgård construction. Although they are using parallel branch structure like RIPEMD-family hash functions and provide good resistance to dedicated attacks but they are still susceptible to generic attacks due to weak construction method. So we designed the hash function MNF-256 on dither construction which is good alternative to MD construction, for better security as discussed in the previous chapter and reduced the number of branches from four to three for better performance of the algorithm.
5.2 Design Choices for MNF-256 Hash Function

Modified NewFORK-256, named as MNF-256 is based on the design principle of NewFORK-256. We optimized the number of branches in NewFORK-256 to further improve the efficiency of existing FORK-family based hash functions. The compression function of MNF-256 consists of three parallel branches. The parallel structure makes differential attack complicated. Structure of three parallel branches of MNF-256 compresses three input parameters: 512-bit message block, 256-bit chaining variables and 512-bit dither values into a 256-bit hash value. Each branch of MNF-256 consists of eight step operations. The design rationale behind hash function MNF-256 is as follows:

(i) Mode of Operation

FORK and NewFORK-256 are based on Merkle-Damgård construction method. Both are vulnerable to generic attacks. We have observed in Chapter 4 that dither construction provides strong resistance to generic attacks, so we have built MNF-256 too on a dither construction. Dither construction assures strong platform to iterate the compression function. It is difficult to find fixed points [46] for compression function when it is iterated through dither construction because compression function includes an additional random dither input to compress the message to hash value. This dither input has also shown good impact on compression process.

(ii) Pre-processing

Pre-processing stage contains three steps: message padding, message parsing and initialization of eight chaining variables. Padding and parsing procedure of the algorithm is exactly the same as that of existing MD and SHA-family based hash functions. The purpose of the message padding is to make the total length of a padded message a multiple of 512. The message $M$ is padded with one bit equal to 1 next to the least significant bit of the message followed by a variable number of zero bits and then appends to the message the 64-bit original message length modulo $2^{64}$, so that the total length of the padded message is the exact multiple of 512.

In the parsing step the message is divided into $N$ blocks of 512-bit, and the $i^{th}$ block of 512-bit is a concatenation of sixteen 32-bit words. The 256-bit chaining variables are used to hold intermediate and final results of the hash function.
There are eight chaining variables $\{A, B, C, D, E, F, G, H\}$. The initial values of variables these chaining variables are exactly the same as the eight initial variables used in NewFORK-256 [103]. The eight chaining variables in each branch are initialized to the following hexadecimal values.

\[
\begin{align*}
A_0 &= 0x6A09E667 & B_0 &= 0xBB67AE85 \\
C_0 &= 0x3C6EF372 & D_0 &= 0xA54FF53A \\
E_0 &= 0x510E527F & F_0 &= 0x9B05688C \\
G_0 &= 0x1F83D9AB & H_0 &= 0x5BE0CD19
\end{align*}
\]

(iii) **Dither Input**

For each message block $M_i$, there is a sequence of 16 different dither values, $D_i \in d_0, d_1, \ldots, d_{15}$. These dither values are applied to each branch with a different order. Dither value ordering for different branches is shown in Table 5.5. These values are generated from Park-Miller algorithm [191]. The Park-Miller algorithm is an efficient and fast algorithm for generating good random sequences. It is based on congruential form: $S_{n+1} = aS_n \mod (2^{31} - 1)$. The steps of the algorithm are described as follows:

(i) initialize the input seed and parameters,

\[
a = 16807, m = 2147483647, q = 127773, r = 2836
\]

(ii) compute the value of $hi = seed \div q; lo = seed \mod q$.

(iii) then compute the corresponding test value: $test = a*lo - r*hi$.

(iv) save the new seed value. If $test > 0$, save test as new seed value, otherwise save

\[
test + m.
\]

(v) output the new seed.

(vi) iterate, and let the output seed be the new input seed.

(iv) **Message permutation**

Hash functions either use message permutation or message expansion. As we have seen in Chapter 3 it is easy to establish attack on hash function that uses message expansion methods. Also same message ordering in different branches are susceptible to attacks for example, RIPEMD, which consists of two branches and follows the same message ordering in both
branches, was fully attacked. On the other hand, in case of RIPEMD-160, there is no attack result because RIPEMD-160 has different message-ordering in branches. So we have used different message ordering in different branches of MNF-256. Message ordering used in three branches is similar to that of message ordering used in first three branches of NewFORK-256. MNF-256 can be implemented efficiently because message ordering is simpler than the message expansion such that of SHA-0/1/2.

(v) **Boolean Function**

Nonlinear functions $f$ and $g$ output one word with one input word. Almost dedicated hash functions use boolean functions which output one word with three words at least. The boolean functions can make it easy to control the output one word by adjusting the input several words. The attacks on MD-family, SHA-family and HAVAL are based on this weakness of boolean functions. Additionally, the output words of $f$ and $g$ functions propagate high diffusion to chaining variables. They update other chaining variables whereas in MD or SHA design based output words of boolean functions are used to update only one chaining variable. Functions $f$ and $g$ are used in MNF-256 are same as that of used in NewFORK-256. MNF-256 uses one bijective functions in the computation of its step operations. The functions $f$ and $g$ of MNF-256 output one word on the input of one word, and their outputs are XORed or added modulo $2^{32}$ to the multiple words in the chaining variables.

They are defined as:

$$f(x) = x \oplus (x <<< 15 \oplus x <<< 27),$$  
$$g(x) = x \oplus (x <<< 7 + x <<< 25).$$

Due to this property, $f$ and $g$ propagate the difference of a message word to the chaining variables. Among these two functions $f$ is completely bijective. The function $f$ eliminates microcollisions in the step transformation, which is observed as crucial points of the attacks on FORK-256. Shift rotations have selected in such way so that the rank of the linearized step function is maximal.

(vi) **Branch Constants**

Each branch uses sixteen different constants represent the first thirty-two bits of the fractional
parts of the cube roots of the first sixteen prime numbers. These constant values are similar to the constants used in NewFORK-256. By using constants we achieve the goal to disturb the attacker who tries to find a good differential characteristic with a relatively high probability. Each step uses two constant values. Values for constants are given in Table 5.6. These constants are used in each $BRANCH_j$ with different order. Ordering of constants is shown in Table 5.7.

(vii) 3-Branch Parallel Structure

MNF-256 uses parallel branch structure like FORK-256 and NewFORK-256. FORK-256 and NewFORK-256 use four branches. MNF-256 consists of three branches. By reducing one redundant computation of branches used in compression function we make it more efficient than its parent algorithms. Since branch 2 is used here in left as well right step structure, so the reduction in branch does not affect over all security of the compression function.

Here, Each $BRANCH_j$, for $1 \leq j \leq 3$, is computed as follows:

(i) The chaining variable $CV_i$ is copied to initial variable $V_{j,0}$ for $j^{th}$ branch.

(ii) At $k^{th}$ step of each branch for $0 \leq k \leq 7$, the step function $STEP_{j,k}$ is computed as follows:

$$V_{j,k+1} = STEP_{j,k}(V_{j,k}, M_{\sigma_j(2k)}, M_{\sigma_j(2k+1)}, \alpha_{j,k}, \beta_{j,k}, d_{j,k}),$$

Where, $\alpha_{j,k}$ and $\beta_{j,k}$ are constants and $d_{j,k}$ is the dither input.

(viii) Compression Function

The compression function of MNF-256 compresses 512-bit input message block, 256-bit chaining variables and 512-bit dither values to a 256-bit hash value. Each message block $M_i$ is divided into sixteen 32 bit words $m_0, m_1, \ldots, m_{15}$ and compressed according to Figure 5.1, where $\Sigma_j(M_j) = (m_{\sigma_j(0)}, \ldots, m_{\sigma_j(15)})$, for $1 \leq j \leq 3$, is the permutation for the message words, selected from Table 5.4.

The chaining variable $CV_i$ is updated to $CV_{i+1}$ according to following relation:

$$CV_{i+1} = CV_i + \{[BRANCH_1(CV_i, M_i, D_i) + BRANCH_2(CV_i, M_i, D_i)] \oplus [BRANCH_2(CV_i, M_i, D_i) + BRANCH_3(CV_i, M_i, D_i)]\},$$

... (5.3)
(ix) **Step Function Processing**

The input $V_{j,k}$ of $STEP_{j,k}$ is divided as follows: $V_{j,k} = (A_{j,k}, B_{j,k}, C_{j,k}, D_{j,k}, E_{j,k}, F_{j,k}, G_{j,k}, H_{j,k})$

$STEP_{j,k}$ uses $(V_{j,k}, m_{\sigma, (2k)}, m_{\sigma, (2k+1)}, \alpha_{j,k}, \beta_{j,k}, d_{j,k})$ as inputs and generates the following output:

Let,

\[
\begin{align*}
    p_1 &= f(A_{j,k} + m_{\sigma, (2k)}) \\
    p_2 &= g(A_{j,k} + m_{\sigma, (2k)} + \alpha_{j,k}) \\
    p_3 &= g(E_{j,k} + m_{\sigma, (2k+1)}) \\
    p_4 &= f(E_{j,k} + m_{\sigma, (2k+1)} + \beta_{j,k})
\end{align*}
\]

Then,
\[
\begin{align*}
A_{j,k+1} &= H_{j,k} \oplus (p_4 \ll 8) \oplus d_{j,k}, \\
B_{j,k+1} &= A_{j,k} + m_{\sigma_j(2k)} + \alpha_{j,k} \oplus d_{j,k}, \\
C_{j,k+1} &= B_{j,k} + p_1 \oplus d_{j,k}, \\
D_{j,k+1} &= C_{j,k} + (p_1 \ll 13) \oplus p_2 \oplus d_{j,k}, \\
E_{j,k+1} &= D_{j,k} \oplus (p_2 \ll 17) \oplus d_{j,k}, \\
F_{j,k+1} &= E_{j,k} + m_{\sigma_j(2k+1)} + \beta_{j,k} \oplus d_{j,k}, \\
G_{j,k+1} &= F_{j,k} + p_3 \oplus d_{j,k}, \\
H_{j,k+1} &= G_{j,k} + (p_3 \ll 3) \oplus p_4 \oplus d_{j,k}.
\end{align*}
\] \hspace{1cm} \ldots (5.4)

One MNF-256 step operation is illustrated in Figure 5.2.

Left shift for a 32-bit string by $s$ bit is denoted as $X \ll s$. In Figure 5.2 four shift values 13, 17, 3 and 8 of a 32-bit string is represented as: $S1$, $S2$, $S3$ and $S4$ respectively.

\textbf{Figure 5.2:} Step operation of MNF-256.
Table 5.4: Message permutation used in MNF-256.

<table>
<thead>
<tr>
<th>$t$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sigma_t$</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>$\sigma_{\tau}(t)$</td>
<td>14</td>
<td>15</td>
<td>11</td>
<td>9</td>
<td>8</td>
<td>10</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>13</td>
<td>0</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>$\sigma_{\tau}(t)$</td>
<td>7</td>
<td>6</td>
<td>10</td>
<td>14</td>
<td>13</td>
<td>2</td>
<td>9</td>
<td>12</td>
<td>11</td>
<td>4</td>
<td>15</td>
<td>8</td>
<td>5</td>
<td>0</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 5.5: Dither input permutation used in MNF-256.

<table>
<thead>
<tr>
<th>$k$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d_{1,k}$</td>
<td>$d_0$</td>
<td>$d_2$</td>
<td>$d_4$</td>
<td>$d_6$</td>
<td>$d_8$</td>
<td>$d_{10}$</td>
<td>$d_{12}$</td>
<td>$d_{14}$</td>
</tr>
<tr>
<td>$d_{2,k}$</td>
<td>$d_{15}$</td>
<td>$d_{13}$</td>
<td>$d_{11}$</td>
<td>$d_9$</td>
<td>$d_7$</td>
<td>$d_5$</td>
<td>$d_3$</td>
<td>$d_1$</td>
</tr>
<tr>
<td>$d_{3,k}$</td>
<td>$d_{14}$</td>
<td>$d_{12}$</td>
<td>$d_{10}$</td>
<td>$d_8$</td>
<td>$d_6$</td>
<td>$d_4$</td>
<td>$d_2$</td>
<td>$d_0$</td>
</tr>
</tbody>
</table>

Table 5.6: Constants used in MNF-256.

<table>
<thead>
<tr>
<th>$\delta_0$ = 0x428A2F98</th>
<th>$\delta_4$ = 0x3956C25B</th>
<th>$\delta_8$ = 0xD807AA98</th>
<th>$\delta_{12}$ = 0x72BE5D74</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\delta_1$ = 0x71374491</td>
<td>$\delta_5$ = 0x59F111F1</td>
<td>$\delta_9$ = 0x12835B01</td>
<td>$\delta_{13}$ = 0x80DEB1FE</td>
</tr>
<tr>
<td>$\delta_2$ = 0xB5C0FBCF</td>
<td>$\delta_6$ = 0x923F82A4</td>
<td>$\delta_{10}$ = 0x243185BE</td>
<td>$\delta_{14}$ = 0x9BDC06A7</td>
</tr>
<tr>
<td>$\delta_3$ = 0xE9B5DBA5</td>
<td>$\delta_7$ = 0xAB1C5ED5</td>
<td>$\delta_{11}$ = 0x550C7DC3</td>
<td>$\delta_{15}$ = 0xC19BF174</td>
</tr>
</tbody>
</table>

Table 5.7: Constants permutation used in MNF-256.

<table>
<thead>
<tr>
<th>$k$</th>
<th>$\alpha_{\delta,k}$</th>
<th>$\beta_{\delta,k}$</th>
<th>$\alpha_{\delta,k}$</th>
<th>$\beta_{\delta,k}$</th>
<th>$\alpha_{\delta,k}$</th>
<th>$\beta_{\delta,k}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$\delta_0$</td>
<td>$\delta_1$</td>
<td>$\delta_{13}$</td>
<td>$\delta_{14}$</td>
<td>$\delta_1$</td>
<td>$\delta_9$</td>
</tr>
<tr>
<td>1</td>
<td>$\delta_2$</td>
<td>$\delta_3$</td>
<td>$\delta_{13}$</td>
<td>$\delta_{12}$</td>
<td>$\delta_5$</td>
<td>$\delta_2$</td>
</tr>
<tr>
<td>2</td>
<td>$\delta_4$</td>
<td>$\delta_5$</td>
<td>$\delta_{14}$</td>
<td>$\delta_{10}$</td>
<td>$\delta_5$</td>
<td>$\delta_4$</td>
</tr>
<tr>
<td>3</td>
<td>$\delta_6$</td>
<td>$\delta_7$</td>
<td>$\delta_9$</td>
<td>$\delta_8$</td>
<td>$\delta_7$</td>
<td>$\delta_6$</td>
</tr>
<tr>
<td>4</td>
<td>$\delta_8$</td>
<td>$\delta_9$</td>
<td>$\delta_7$</td>
<td>$\delta_6$</td>
<td>$\delta_9$</td>
<td>$\delta_8$</td>
</tr>
<tr>
<td>5</td>
<td>$\delta_{10}$</td>
<td>$\delta_{11}$</td>
<td>$\delta_5$</td>
<td>$\delta_4$</td>
<td>$\delta_{11}$</td>
<td>$\delta_{10}$</td>
</tr>
<tr>
<td>6</td>
<td>$\delta_{12}$</td>
<td>$\delta_{13}$</td>
<td>$\delta_1$</td>
<td>$\delta_2$</td>
<td>$\delta_{13}$</td>
<td>$\delta_{12}$</td>
</tr>
<tr>
<td>7</td>
<td>$\delta_{14}$</td>
<td>$\delta_{15}$</td>
<td>$\delta_0$</td>
<td>$\delta_5$</td>
<td>$\delta_{14}$</td>
<td>$\delta_{14}$</td>
</tr>
</tbody>
</table>
5.3 Description of MNF-256 Hash Function

MNF-256 generates hash by iterating a compression function \( f : \{0,1\}^e \times \{0,1\}^m \times \{0,1\}^d \rightarrow \{0,1\}^n \) according to following algorithmic steps:

1. Pad and split a message \( M \) into \( N \) blocks \( (M_1, \ldots, M_N) \) of \( m \) bits each.
2. Set \( H_0 \) to the initialization value \( IV \).
3. For each message block \( M_i \) compute \( H_i = f(H_{i-1}, M_i, D_i) \).
4. Output \( h_f(M) = H_N \).

Step 1: Message Padding and Parsing

Append the bit 1 to the end of the \( l \)-bit message \( M \), followed by \( k \) zero bits, where \( k \) is the smallest, non-negative solution to the equation \( l + 1 + k = 448 \mod 512 \). Then append the 64-bit block represents length of original message. The length of the padded message is now a multiple of 512 bits.

Step 2: Setting the Initial Chaining Variables

\[
\begin{align*}
A_0 &= 0x6A09E667 & B_0 &= 0xBB67AE85 \\
C_0 &= 0x3C6EF372 & D_0 &= 0xA54FF53A \\
E_0 &= 0x510E527F & F_0 &= 0x9B05688C \\
G_0 &= 0x1F83D9AB & H_0 &= 0x5BE0CD19
\end{align*}
\]

Step 3: Processing

\[
\begin{align*}
A_{j,k+1} &= H_{j,k} \oplus (p_4 \ll 8) \oplus d_{j,k}, & B_{j,k+1} &= A_{j,k} + m_{\sigma_1(2k)} + \alpha_{j,k} \oplus d_{j,k}, \\
C_{j,k+1} &= B_{j,k} + p_1 \oplus d_{j,k}, & D_{j,k+1} &= C_{j,k} + (p_1 \ll 13) \oplus p_2 \oplus d_{j,k}, \\
E_{j,k+1} &= D_{j,k} \oplus (p_2 \ll 17) \oplus d_{j,k}, & F_{j,k+1} &= E_{j,k} + m_{\sigma_j(2k+1)} + \beta_{j,k} \oplus d_{j,k}, \\
G_{j,k+1} &= F_{j,k} + p_3 \oplus d_{j,k}, & H_{j,k+1} &= G_{j,k} + (p_3 \ll 3) \oplus p_4 \oplus d_{j,k}.
\end{align*}
\]
Where,
\[ p_1 = f(A_{j,k} + m_{\sigma_j(2k)}) \]
\[ p_2 = g(A_{j,k} + m_{\sigma_j(2k)} + \alpha_{j,k}) \]
\[ p_3 = g(E_{j,k} + m_{\sigma_j(2k+1)}) \]
\[ p_4 = f(E_{j,k} + m_{\sigma_j(2k+1)} + \beta_{j,k}) \]

**Step 4: Output**

After computing outputs of parallel branches following computation is performed to update \( CV_i \) to \( CV_{i+1} \):
\[ CV_{i+1} = CV_i + [(h_1 + h_2) \oplus (h_2 + h_3)], \text{ for } 1 \leq j \leq 3 \]
where \( h_j = \text{BRANCH}_j(CV_j, \Sigma_j M, \Sigma_j D) \), \( \Sigma_j M \) re-ordering of message words and \( \Sigma_j D \) reordering of dither words.

### 5.4 Strength of 3-Branch Structure Against Collision Attack Strategies

Collision finding attacks on single branch of MNF-256 can be considered in two individual scenarios. The first one is a chosen \( IV \) collision attack and the second one is an ordinary collision attack. Chosen \( IV \) collision finding attack is an attack which is worth considering on each single branch of MNF-256. In this attack, finding compatible \( IVs \) together with appropriate message differences can be led to collision. Here, this type of attack is not applicable on one branch of MNF-256. Since gaining collision in one branch is only possible by finding a nonzero XOR difference for some message words and preserving the other message word differences zero. This attack has a complexity not less than what it is in birthday attack.

Ordinary collision attack on single branch of MNF-256 can be successful if someone can insert a differential characteristic through one branch leading to zero differences in the last step. To this aim, the attacker should follow one of the following two strategies: first the attacker inserts one or more non zero difference message words in the first step and expects to meet zero difference words at the end of the last step of one branch. Second, the attacker constructs two individual characteristics for two semi-branches, using meet in the middle technique. In this scenario, the
attacker wishes that constructed characteristics for the first four steps and the last four steps in opposite direction meet each others at the end of fourth step.

Let us consider these strategies. Suppose that the attacker inserts one or more nonzero difference message words as input to the first step. Looking at the structure of each step reveals that the two message words along with a random dither value are involved in each step and changing in a message word at the beginning of the step causes the attackers decision for altering messages gaining to collision too complex. By the way, in this case the entire message must be changed to frustrate the effects of assigned non zero message words differences at the output of the branch. This makes the job of the attacker too hard due to arisen complexity in simultaneously equations. This complexity is not less than what it is in birthday attack \(2^{128}\) due to existence of some good properties of functions \(f\) and \(g\).

The second scenario is more complex than the first. In this strategy, he should find two individual and depended characteristics which collide with another in the middle of the branch. So, forced conditions resulted in more simultaneously equations than the first strategy will grow. Moreover, if an attacker inserts the message difference to find a collision in 3-branch then, he expects the following:

\[
(\Delta_1 + \Delta_2) \oplus (\Delta_2 + \Delta_3) = 0
\]

Where, \(\Delta_i\) is the output difference of the \(BRANCH_i\). Such a differential pattern an attacker can obtain in following way:

(i) To construct a differential characteristic with a high probability for a branch function, say \(BRANCH_i\) and then expects that, the operation of the output differences in the other branch \(\Delta_i\) is equal to \(\Delta_i\).

Considering the structure of a branch of MNF-256, it can be inferred that using functions with good properties, high diffusion structure, and different permutation of input message words for each branch causes the outputs of a branch to be randomized. So it can be expected that finding a collision costs at least \(2^{256}\).

(ii) To construct two different differential characteristics such that:
\[(\Delta_1 + \Delta_2) = -(\Delta_2 + \Delta_3).\]  \hspace{1cm} \ldots(5.6)

This can be generated for cancelling the first and second chaining values to obtain the difference between the chaining values as zero, the required condition for generating an attack.

To find an attack using this strategy an attacker has to construct such a differential pattern of the message words. But, for any message words it is computationally hard to find such sequences.

(iii) To insert the message difference which yields same message difference pattern in all the three branches and expect that, same differential characteristics occur simultaneously in three branches.

This strategy is relatively easy for an attacker. However, using the message word reordering this can be avoided just as in the case of FORK and NewFORK-256. Since the same message word reordering is used in the proposed hash functions same security level can be expected for it against this strategy. Moreover, using different operators (e.g. + and \(\oplus\)) highly complicates the computation of good differential paths. Addition of message words, parallel mixing structure, rotation of registers and addition of dither value made compression function stronger against differential attacks.

5.5 MNF-256 and Attack of Saarinen

Markku-Juhani O. Saarinen presented collision attack against NewFORK-256 using meet-in-the-middle technique [171]. For this, researcher used a method for finding messages that hash into a significantly smaller subset of possible hash values. The complexity of this collision attack is \(2^{112.8}\). This attack is also applicable for FORK-256. For MNF-256 this attack does not reduce the collision search, from the \(2^{128}\) to \(2^{112.8}\) number of hash calls, this is due to the fact that compression function of MNF-256 algorithm along with message block and chaining variables also requires dither input.

For the observation of compression function of MNF-256 against this attack we have not considered the input of dither value initially. Here, each branch of the compression function uses each message word \(m_0, m_1, \ldots, m_{15}\) exactly once. Due to the diffusion properties of the step
function, message words that are scheduled for the last steps do not affect all output words. Consider the sixth output word of each branch, $F_{j,8}$. The last step is defined as:

$$F_{j,8} = E_{j,7} + m_{\sigma_j(15)} + \beta_{j,7} \quad \ldots (5.7)$$

Furthermore $E_{j,7}$ in the previous step of it is defined as:

$$E_{j,7} = D_{j,6} \oplus (p_2 \ll 17), \text{ where } p_2 = g(A_{j,k} + m_{\sigma_j(2k)} + \alpha_{j,k}) \quad \ldots (5.8)$$

$$E_{j,7} = D_{j,6} \oplus (g(A_{j,6} + m_{\sigma_j(12)} + \alpha_{j,6})) \ll 17 \quad \ldots (5.9)$$

Ignoring the round constants $\alpha_{j,k}$ and $\beta_{j,k}$, it is observed that the only message words in steps 6 and 7 affecting $F_{j,8}$ are $m_{\sigma_j(12)}$ and $m_{\sigma_j(13)}$ the latter having a linear effect. Constants $\alpha_{j,k}$ and $\beta_{j,k}$ have no effect on the computation of this word. Thus by inspecting the step function and the message word schedule in Table 5.4, it is easy to verify that $F_j$ satisfies the following properties:

Branch 1: $F_{1,8}$ is independent of $m_{l_4}$,

Branch 2: $F_{2,8}$ is dependent on $m_1$ and $m_{l_4}$,

Branch 3: $F_{3,8}$ is independent of $m_1$.

The main strategy of this attack is to use a fast method for finding messages that hash into a significantly smaller subset of possible hash values by forcing the sixth word of the compression function to remain constant over the hash function iteration, i.e. $F_0 = F_i = 0x9B05688C$, thereby generating hash values in a subset of size $2^{224}$. Assuming uniform distribution, a full collision can be expected after $\sqrt{2\ln 2 \times 2^{224}} \approx 2^{112.2}$ hashes in the small subset have been found. This follows from the birthday attack.

The value of $F_i$ is combined from the three branches and the initialization vector is as follows:

$$F_i = F_0 + \{(F_{1,8} + F_{2,8}) \oplus (F_{2,8} + F_{3,8})\}. \quad \ldots (5.10)$$

By substituting $F_0 = F_i$ and regrouping branches we obtain the following necessary and sufficient conditions:
\[(F_{2,8} - F_{3,8}) = (F_{1,8} - F_{2,8}). \] 

...(5.11)

This attack is based on choosing two message words \(m_i\) and \(m_{14}\) in a specific way to satisfy \(F_0 = F_1\), which is possible due to the above observations. The values of the other message words are arbitrary and can be chosen at random as long as they remain constant through the two phases of the attack. The two phases can be repeated any number of times to produce sufficient hashes in the subset by the following procedure [171]:

**Phase I**

Set \(m_i = 0\) and loop over \(m_{14} = 0, 1, 2, \ldots, 2^{32} - 1\). Compute branches 2 and 3 for each \(m_{14}\) to obtain \(x = (F_{2,8} - F_{3,8})\). Place \(x\) and \(m_{14}\) into a look-up table so that the value of \(m_{14}\) can be immediately retrieved based on the corresponding \(x\) value i.e. \(m_{14}\) is indexed by \(x\).

**Phase II**

Loop over the \(2^{32}\) values of \(m_i\) for more than one value of \(m_{14}\). Compute branches 1 and 2 for each \(m_i\) to obtain \(y = (F_{1,8} - F_{2,8})\). In each step, perform a look-up.

If any matches \(x = y\) are found, the necessary and sufficient condition is satisfied and we have found a message or rather, a pair of \(m_i\) and \(m_{14}\) values that produce one or more hashes that satisfy \(F_0 = F_1\). Each loop step in the second phase can produces one match in the lookup table on average. This is due to the fact that there are a total of \(2^{32}\) \(m_{14}\) entries in the table. Hence, approximately \(2^{32}\) hashes with the property are produced in the later phase. Since computation of only two branches out of three is needed, the computational effort in the first and second phases is equal to \(2^{32}\) full hash computations each. If the full eight words in first phase are not stored, branches 2 and 3 need to be computed again to reproduce a full hash, bringing the total number to \(3 \times 2^{32}\). The average cost of producing a hash is therefore \(3/2\) hash function invocations. In addition to this since the step function of MNF-256 also takes 32-bit dither input in each step function to update the chaining variable, thus it requires to produce hash in the subset of \(2^{256}\).
instead of $2^{224}$. So for obtaining a favorable cost for each hash with the desired property $F_0 = F_1$ when dither inputs are considered is therefore equivalent to $\sqrt{2\ln 2 \times 2^{256}} \approx 2^{128}$.

5.6 Conclusion

In this proposal we have designed a 256-bit hash function MNF-256. It generates hash result of 256-bit. It is built on dither construction. Additional random third input to the compression function due to dither construction makes efforts for finding generic attacks such as multicollision, fixed points, long message second preimage attack more complex. MNF-256 compresses three inputs in a structure of 3 parallel branches. These parallel branches resist well different collision finding dedicated attacks. It is shown that collision attack against MNF-256 using meet-in-the-middle technique will require not less than $2^{128}$ operations. MNF-256 has lesser number of branches than its parent algorithms ensures fast and efficient computation of hash value. Performance evaluation of MNF-256 is evaluated in Chapter 7.