Cryptanalysis of FORK-256 ⋆

Thomas Peyrin and Olivier Billet

France Télécom R&D
38–40, rue du Général Leclerc
92794 Issy les Moulineaux Cedex 9 — France
{forname.lastname}@orange-ftgroup.com

Abstract. In this paper we expose a practical attack against a new hash function design, FORK-256, which was proposed by Hong *et al.* at FSE 2006. Our attack allows to find a collision against a 160-bit truncated version of the FORK-256 compression function with a complexity of 2^{49} hash computations and with negligible memory. This has to be compared with the theoretical complexity 2^{80} hash computations given by the birthday paradox. Additionally, we expose a 1 bit (resp. 2-bit) near-collision attack against the full version of FORK-256 running with a complexity of 2^{125} (resp. 2^{120}) and with negligible memory, and exhibit a 22-bit near collision. Finally, we discuss very recent independent results about FORK-256, and show how our attack strategy can be used to improve upon these results to yield a collision against the complete version of FORK-256 with a complexity of 2^{106} hash computations and about 2^{64} memory.

1 Introduction

Cryptographic hash functions are essential primitives for the secure deployment of many protocols and applications in the digital world. Hash functions map binary strings of arbitrary length to a binary string of fixed length n. For a hash function H to be cryptographically secure, three properties have to be fulfilled, respectively called preimage resistance, second preimage resistance, and collision resistance. Preimage resistance is the computational difficulty, given any hash value y, to find an input string x such that $H(x) = y$. Second preimage resistance is the computational difficulty, given a hash value y and an input string x such that $H(x) = y$, to find another input string z for which $H(z) = y$. Finally, collision resistance is the computational difficulty to find two distinct inputs mapping to the same output through H. By computational difficulty, we mean that there is no attack better than the generic ones: to find a collision requires at least $2^{n/2}$ hash computations, and to find a preimage or a second preimage requires at least 2^n hash computations, where the length of H's output is n bits. A common way to build a hash function is to use the Merkle and Damgård iteration paradigm [4, 19]: the hash function H consists in iterating a compression function h that maps a chaining variable of size n bits together with a block of size m bits of the message to be hashed to the next value of the chaining variable. The hash output is the value of the chaining variable at the end of the iteration process. It can be proven that the hash function H is then at least as resistant as its underlying compression function to collision, but recent cryptanalytic work has shown the limits of this iteration scheme for more complex security notions [11, 9].

⋆ This work is supported in part by the french government through the SAPHIR project.

Three main strategies can be identified in the design of cryptographic hash functions: hash functions based on block ciphers, hash functions whose security relates to some hard problem, and dedicated designs. There has been several proposals from this last category through the years: MD4 [24] and MD5 [25], RIPEMD [7], and SHA-0, SHA-1 [21] to name a few, of which MD5 and SHA-1 may be the most vastly deployed. However, some recent results [27, 28, 2, 1, 5] have shown that these functions are far from behaving like ideal hash functions, since there exist attacks much faster than the generic ones. As a consequence, new hash function proposals have been made to prevent this new type of attacks. This is the case of FORK-256 [8], which was proposed at FSE 2006 and at the first NIST Workshop on hash functions.

The paper is organized as follows. In the next section, we quickly describe the FORK-256 hash function proposal. Then, we expose the technical elements that we use throughout our attacks. The description of our main algorithm follows, that allows to cancel 160 predetermined output difference bits of the FORK-256 hash function. We also expose two ways of using this result to attack FORK-256. Finally, we discuss the results that appeared very recently in [15] and improve upon them to compute a collision against the complete version of FORK-256 in 2^{106} hash computations and 2^{64} memory.

2 Description of FORK-256

This section briefly describes the design of the FORK-256 hash function as proposed at FSE 2006 by Hong *et al.* in [8]. For further details, we refer the reader to the original article. We hereafter use '$\oplus$' to denote bitwise exclusive or, '$+$' to denote integer addition modulo 2^{32}, and $w^{\lll k}$ to denote the word w cyclically rotated by k bit positions to the left. The hash function FORK-256 follows the iteration principle proposed by Merkle [19] and Damgård [4], and its compression function hashes a 512-bit message block M at each iteration and uses a 256-bit chaining variable cv_n. The name of FORK-256 comes from the fact that the internal state is modified simultaneously in four parallel streams, and the four corresponding outputs $h_1, \ldots, h_4$ are recombined via $h' = (h_1 + h_2) \oplus (h_3 + h_4)$ just before applying a feed-forward which produces the output $\mathrm{cv}_{n+1} = h' + \mathrm{cv}_n$ of the compression function, as shown in Fig. 2 in App. B. To process the 512-bit message M with the chaining variable cv_n, M is first subdivided into sixteen 32-bit words M_0, M_1, $\ldots$, M_{15}. The processing applied in each of the four streams is the same: it consists of eight iterations of a step transformation on an internal state. The internal state consists of eight 32-bit words denoted by (A, B, C, D, E, F, G, H), and the step transformation involves several parameters such as varying constants $\alpha_{j,r}$ and $\beta_{j,r}$ (defined in Table 5 of App. B), and two 32-bit words M_i and M_j from M. These words M_i and M_j are chosen depending on the stream number and the round number according to the rules of Table 4: basically, a permutation σ_j is applied in stream j to select the sub-blocks $M_{\sigma_j(2i)}$ and $M_{\sigma_j(2i+1)}$ at round i. The step transformation itself is pictured in Fig. 1. It operates on the internal state that was initially set to the value of the chaining variable. The internals are built around the two following non-linear 32-bit word to 32-bit word functions f and g:

$$f(x) = x + (x^{\lll 7} \oplus x^{\lll 22}), \qquad g(x) = x \oplus (x^{\lll 13} + x^{\lll 27}). \tag{1}$$

In the following, we denote by $A_{j,r}, \ldots, H_{j,r}$, words of the internal registers of stream j, after step r. The words $A_0, \ldots, H_0$, denote the common initial state of the registers, and there are eight rounds in each of the four streams.

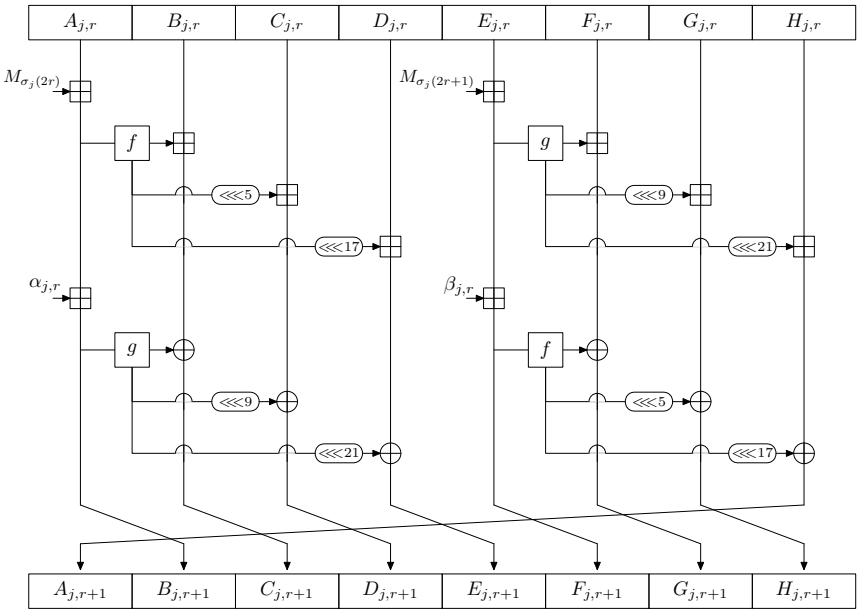

Fig. 1. Internals of FORK-256's step function for stream j, $j = 1, \ldots, 4$. The message blocks used at round r are $M_{\sigma_j(2r-1)}$ and $M_{\sigma_j(2r)}$. (The permutation σ is given in Table 4 while the additional values α and β are constants defined in Table 5 given in App. B.)

3 Preliminary properties of FORK-256

As seen in the previous section, FORK-256 uses four parallel streams operating on the same initial state (or cv_n) and using the same blocks of messages but in a different order. This seems to be the strength of FORK-256 since the two first reported efforts to break it were limited to two of the four streams [16, 15]. In the work of [16], a collision is exhibited for a reduced version of FORK-256 with two streams. In the study [15] another technique is used to provide a chosen IV collision for the same reduced version of FORK-256 with two streams. But [15] was later on expanded to include an attack against the full version of FORK-256.

In this paper, we present an independent analysis resulting in a 1-bit near-collision attack against a reduced version of FORK-256 keeping *all of its four streams*, but with seven rounds instead of the eight rounds of the original hash function. Moreover, we show in a later section how to use this result to attack the complete FORK-256 hash function. Along with those independent results, we also improve upon a very recent addition to [15] (which we show to contain flaws) in the last section.

3.1 Differential characteristics

As noted above, the main difficulty in cryptanalysing FORK-256 comes from the fact that the same message blocks are input in each of the four streams in a permuted fashion. Thus, while one or maybe two streams may be easily dealt with, the effect of the difference is difficult to cancel in the remaining streams. There are however at least two specific

differential characteristics of interest. The first one, as noted by [20] and [16], overcomes

Table 1. A four steps differential pattern to force an inner collision for FORK-256. The table shows the pattern in one stream and its probability to occur for each round.

step	ΔA	ΔB	ΔC	ΔD	ΔE	ΔF	ΔG	ΔH	ΔM_L	ΔM_R	Prob.
in	δ	δ	δ	δ	δ	δ	δ	δ	$-\delta$	$-\delta$	
1	δ	0	δ	δ	δ	0	δ	δ	$-\delta$	$-\delta$	P_δ^6
2	δ	0	0	δ	δ	0	0	δ	$-\delta$	$-\delta$	P_δ^4
3	δ	0	0	0	δ	0	0	0	$-\delta$	$-\delta$	P_δ^2
out	0	0	0	0	0	0	0	0			1

the issue by applying the same *additive* difference δ to every message block. Hence, just after the fourth step has been completed, if the internal state has the same difference δ on all of its eight 32-bit words, there is a collision after round eight. This behavior, summarized in Table 1, renders the use of four streams with message reordering as a means to protect against differential analysis ineffective since the same difference is applied to every message block and the same differential pattern is occurring simultaneously in the four streams. The probability P_δ is the probability that the difference δ propagates without modification in one step. (It does propagates without modification when it enters the round in the register A or the register E of the internal state, but with probability P_δ otherwise.) This probability can be computed exactly for any given difference δ, and this computation is given in App. A. The overall probability for the differential pattern of Table 1 to occur is thus P_δ^{12} for each stream.

Table 2. A seven steps differential pattern to get an inner near-collision for FORK-256.

step	ΔA	ΔB	ΔC	ΔD	ΔE	ΔF	ΔG	ΔH	ΔM_L	ΔM_R	Prob.
in	0	δ	0	0	0	0	0	0	0	0	
1	0	0	δ	0	0	0	0	0	0	0	P_δ
2	0	0	0	δ	0	0	0	0	0	0	P_δ
3	0	0	0	0	δ	0	0	0	0	0	P_δ
4	0	0	0	0	0	δ	0	0	0	0	P'
5	0	0	0	0	0	0	δ	0	0	0	P_δ
6	0	0	0	0	0	0	0	δ	0	0	P_δ
out	δ	0	0	0	0	0	0	0			P_δ

Another way to deal with the four streams simultaneously is to apply a difference on the IV instead of the message M. This specific type of collision is called a pseudo-collision, and can be expressed as follows for the compression function h of any iterated hash function: $h(IV, M) = h(IV', M')$ where $(IV', M') \neq (IV, M)$. In the case of FORK-256, differences in the words of the internal state register do not diffuse identically, see the description of the internals of FORK-256's step function in Fig. 1. More precisely, only the differences in the words A and E will spread to the other registers in the next round. The other differences (in the words B, C, D, F, G, H) only shift one word to the right. Hence, by applying a difference to the second word of the IV, the difference propagates without spreading during

three rounds[1]. (Note that it propagates without being modified with probability P_δ only, just as for the first differential pattern. This comes from the fact that it has to pass through a '$\oplus$'. Indeed, additive differences do not propagate without modification whenever '$\oplus$' are used in the design of the hash function—just as xor differences do not propagate without modification whenever '+' are used. Again, for the exact computation of probability P_δ for a given difference δ, we refer the interested reader to App. A.) During the fourth round however, the difference spreads *a priori* to the three internal registers F, G, and H in all four streams, with some probability $(1 - P')$. We show in paragraph 3.2 that there is a way to prevent such a spreading of the difference with probability 1. Then, the difference again propagates during three rounds without spreading. As a result, for a difference with a low Hamming weight, we obtain a near-collision at the seventh round of FORK-256. In Section 5, we show how to take advantage of this fact to attack the real FORK-256.

3.2 Getting through the fourth round or how to make $P' = 1$

The main obstacle to the realisation of the seven rounds pattern is to prevent the difference from spreading at the fourth round. Remember that the spreading can occur both at the left side or at the right side of FORK-256's internal step transformation. The technique to prevent this spreading is the same at the left or at the right side: only f and g are swapped, and some constants changed, see Fig. 1. In the following, we focus on the right side, but the same work can be done on the left side. To put the spreading problem into equation:

Problem 1 (Spreading Problem). *Given rotations values ρ_f and ρ_g together with a constant β, is there any tuple of values (x, x^*, y) such that $x \neq x^*$ and*

$$\big(g(x)^{\lll \rho_g} + y\big) \oplus \big(f(x+\beta)^{\lll \rho_f}\big) = \big(g(x^*)^{\lll \rho_g} + y\big) \oplus \big(f(x^*+\beta)^{\lll \rho_f}\big)\,? \tag{2}$$

This problem has been somehow studied in [16, 15] but here we give another strategy to solve it, especially suited to our attack. The authors of [15, 16] both suggest a first way that partially answer this problem. They look for input values x and x^* such that:

$$g(x) = g(x^*), \qquad f(x+\beta) = f(x^*+\beta). \tag{3}$$

Depending on the value of the constant β, such a pair (x, x^*) may or may not exist. The authors of [16] call the constants β such that there exists at least one pair fulfilling Eq. 3 *weak constants*. In our attack however, we are interested in weak constants involved in the fourth round, that is $\beta_{1,3}$, $\beta_{2,3}$, $\beta_{3,3}$, and $\beta_{4,3}$. Unfortunately, [16, 15] show that none of these are weak constants.

A restricted version of the initial problem of difference spreading is when the difference $\delta = x^* - x$ is fixed and the problem becomes to find a solution (x, y) to Eq. 2. A natural strategy is to perform an exhaustive search on x so that if $a = g(x)^{\lll \rho_g}$, $b = g(x+\delta)^{\lll \rho_g}$, $c = f(x+\beta)^{\lll \rho_f}$, and $d = f(x+\delta+\beta)^{\lll \rho_f}$, the problem amounts to finding y satisfying

$$(a + y) \oplus c = (b + y) \oplus d, \tag{4}$$

where a, b, c, and d are fixed values. This last equation is easily solved considering the following argument. This equation has a solution only if the lowest significant bits of each

[1] In our first kind of attacks, we only use a difference in the second word of the IV and no difference at all in the message blocks.

word satisfy: $a_0 \oplus c_0 = b_0 \oplus d_0$. (We use the notation w_i for the i-th lowest significant bit of word w.) The same reasoning can obviously be applied iteratively to the next lowest bits, but now a carry has to be taken into account. Let us call l_1 and r_1 the carry from the left member and the right member respectively. Again, Eq. 4 has a solution only if $a_1 \oplus c_1 \oplus l_1 = b_1 \oplus d_1 \oplus r_1$, where $l_1 = a_0 y_0$ and $r_1 = b_0 y_0$. Thus, at the i-th bit stage, Eq. 4 admits a solution only if $a_i \oplus c_i \oplus l_i = b_i \oplus d_i \oplus r_i$, where $l_i = a_{i-1} y_{i-1} \oplus a_{i-1} l_{i-1} \oplus l_{i-1} y_{i-1}$ and $r_i = b_{i-1} y_{i-1} \oplus b_{i-1} r_{i-1} \oplus r_{i-1} y_{i-1}$.

Now the solving algorithm is pretty straightforward: at each step i, check if the corresponding equation $a_i \oplus c_i \oplus l_i = b_i \oplus d_i \oplus r_i$ is consistent for all possible pairs of carries (l_i, r_i). If not, halt with the result "*there is no solution.*" Else, compute the set of all corresponding carries l_{i+1} and r_{i+1} for the next step for both possible values of y_i and for all pairs of valid carries (l_{i-1}, r_{i-1}) from the last step. Since at any step, the cardinality of the set of possible carries (l_i, r_i) is only four, there is only a small set of checks to do at each steps. If the algorithm arrives at the most significant bit level, there is at least a solution which is obtained by inspecting backward pairs of carries that have not been invalidated and choosing corresponding values of y_i.

Hence, there is an algorithm that solves the spreading problem for a fixed additive difference $\delta = x^* - x$ with time complexity 2^{32} and with constant memory, by testing all possible values of x. Actually, we are interested in obtaining a solution of the spreading problem for a fixed difference δ, but with each of the three pairs of rotations (ρ_f, ρ_g): $(0, 0)$, $(5, 9)$, and $(17, 21)$, together with each of the four constants $\beta_{1,3}$, $\beta_{2,3}$, $\beta_{3,3}$, and $\beta_{4,3}$ in turn. This corresponds to the three threads mapping (F, G, H) to (G, H, A) in the four streams of the fourth step of FORK-256, see Fig. 1. The algorithm goes exactly the same in this case: for every possible value of x, it looks for a solution in each thread one after the other to get a quadruplet (x, y_F, y_G, y_H) of values so that the spreading of the fixed difference δ is prevented in all threads simultaneously. We do this for the four streams independently.

4 Attacking a seven rounds reduced version of FORK-256

In the previous section we have seen that the seven rounds differential pattern of Table 2 with an additive difference δ happens simultaneously in the four streams with probability P_δ^{12} as soon as the registers (E, F, G, H) reach a prescribed value at the fourth round in the four streams. (Remember that P_δ is the probability that the additive difference goes unmodified in when a '$\oplus$' is involved. Also, we do not care about the difference being modified after the fourth round, because it never spreads again. So the factor 12 comes from the fact that the difference has to go unmodified through the three first rounds in all of the four streams.) We show here how to force these registers to take prescribed values. These prescribed values are the four quadruplets of values computed to solve the spreading problem at the fourth round of each of the four streams with the algorithm of Sec. 3.2.

4.1 Near collision at the seventh round

Our main tool here is a good scheduling in the determination of each message block so as to be able to force the four quadruplets of each stream to their required values.

To this aim, we study the relationships between message blocks and IV blocks with this last quadruplet. Before getting into deeper details of the attack, let us emphasize the following fact that simplifies the study of these relationships in stream j: forcing the value

of the quadruplet $(E_{j,3}, F_{j,3}, G_{j,3}, H_{j,3})$ is equivalent to forcing the value of the quadruplet $(E_{j,3}, F_{j,3}, F_{j,2}, F_{j,1})$. This fact can be easily checked by going backward in the threads of FORK-256's step transformation, which can be translated in the following sequence of equations:

$$\begin{aligned} F_{j,2} &= \big(G_{j,3} \oplus f(F_{j,3})\big) - g(F_{j,3} - \beta_{j,2}), \\ G_{j,2} &= \big(H_{j,3} \oplus f(F_{j,3})^{\lll 5}\big) - g(F_{j,3} - \beta_{j,2})^{\lll 9}, \\ F_{j,1} &= \big(G_{j,2} \oplus f(F_{j,2})\big) - g(F_{j,2} - \beta_{j,1}). \end{aligned} \tag{5}$$

Taking this remark into account, the Table 3 summarizes the relationships. In the left column are printed the words of the quadruplets that we would like to force to some predetermined value, and each row shows the dependence of one word in the message blocks and IV registers.

Table 3. Relationship between the words of the quadruplets in each stream and the message blocks and the IV. The symbols '`*`' and '`x`' both denote a degree of freedom to set the value of a word W by adjusting the corresponding parameter P when all the remaining parameters of the row have already been fixed. The '`x`' is used to emphasize that the parameter P can be used *directly* to set word W to its target value.

	IV								message block M_i															
	A	B	C	D	E	F	G	H	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$E_{4,3}$	*	x		*	*		*	*		*				*			*			x	*			*
$E_{3,3}$	*	x		*	*		*	*							*	*			*		x	*	*	
$E_{2,3}$	*	x		*	*		*	*					x				*	*		*			*	*
$E_{1,3}$	*	x		*	*		*	*	*	*	*	*	*			x								
$F_{4,3}$	*		x		*			*	x	*				*							*			
$F_{3,3}$	*		x		*			*			x				*	*			*					
$F_{2,3}$	*		x		*			*											x	*			*	*
$F_{1,3}$	*		x		*			*	*	*	*			x										
$G_{4,3} \leftrightarrow F_{4,2}$	*			x										*			x							
$G_{3,3} \leftrightarrow F_{3,2}$	*			x												*							*	
$G_{2,3} \leftrightarrow F_{2,2}$	*			x														x					*	
$G_{1,3} \leftrightarrow F_{1,2}$	*			x					*			x												
$H_{4,3} \leftrightarrow F_{4,1}$					x																x			
$H_{3,3} \leftrightarrow F_{3,1}$					x										x									
$H_{2,3} \leftrightarrow F_{2,1}$					x																			x
$H_{1,3} \leftrightarrow F_{1,1}$					x					x														

In view of Table 3, we propose the following algorithm to sequentially assign values to the message blocks and IV values so that the four quadruplets in all four streams actually take the prescribed values. Adjustments are refinements of the algorithm introduced to help the difference propagate without modification; for instance, by forcing the value entering function g to be zero, we ensure that the difference propagates unmodified.

1. *Initialize.* Choose A_0 randomly.
2. *Adjust.* Do the following four assignments: $M_0 = -(A_0 + \alpha_{1,0})$, $M_{14} = -(A_0 + \alpha_{1,1})$, $M_7 = -(A_0 + \alpha_{1,2})$, $M_5 = -(A_0 + \alpha_{1,3})$.
3. *Force words $G_{j,3}$.* Choose D_0 so that $F_{3,2}$ gets the correct value. Then, choose M_3, M_9, and M_8 in turn so that $F_{1,2}$, $F_{2,2}$, and $F_{4,2}$ respectively gets their correct value.

4. *Force words $H_{j,3}$.* (If this step has been run 2^{32} times, return to step 1.) Randomly choose H_0. Adjust M_{11} and M_1 to prevent the difference from being modified in the second round of FORK-256 in stream 2 and 4 respectively. Then, set the words E_0, M_{15}, M_6, and M_{12} so that $F_{1,1}$, $F_{2,1}$, $F_{3,1}$, and $F_{4,1}$ respectively gets their correct value.
5. *Force words $F_{j,3}$.* Set C_0 so that $F_{4,3}$ gets its correct value. Then, set M_{10} and M_2 in turn so that $F_{2,3}$ and $F_{3,3}$ respectively gets their correct value. Now, $F_{1,3}$ is assigned some random value. If this value is the correct one, continue to next step, otherwise, return to step 4.
6. *Force words $E_{1,3}$, $E_{2,3}$, and $E_{4,3}$.* (If this step has been run 2^{32} times, return to step 1.) Choose some random value for G_0. Fix B_0 so that $E_{4,3}$ takes the correct value. Fix M_4 so that $E_{2,3}$ takes the correct value. Check the random value taken by $E_{1,3}$: if this is the expected value, continue to step 7, else go back to step 6.
7. *Force word $E_{3,3}$.* (If this step has been run 2^{32} times, return to step 1.) Choose some random value for M_{13}. Check the random value taken by $E_{3,3}$: if this is the expected value, output all messages M_i and all IV blocks. Otherwise, go back to step 7.

Notice that this algorithm makes a few *independent* exhaustive searches in spaces of size 2^{32}. Almost no memory is required, and the average time complexity is 2^{32} applications of one fourth of FORK-256, that is about 2^{30} computations of the hash function. Now, for the attack to succeed, the difference δ has to propagate unmodified up to round 3. Since the probability to propagate in one stream is P_δ, and taking into account the fact that we took care of it in the first round (step 2 of the algorithm) and in two of the four streams of the second round (step 4 of the algorithm), the overall probability is P_δ^6.

We eventually remark that the word F_0 of the IV does not modify the four targeted quadruplets. Hence, in the output of our algorithm, we can make F_0 take any of the 2^{32} possible values and the result remains valid. That is, our algorithm actually outputs 2^{32} values (M, IV), so that the registers $B_{j,7}$, $C_{j,7}$, $D_{j,7}$, $E_{j,7}$, $F_{j,7}$, $G_{j,7}$, $H_{j,7}$ of the internal state reached in each of the four streams just before the eighth round is the same whether (M, IV) is processed or (M, IV') is processed, where IV' is equal to IV except for B_0 which is replaced by $B_0 + \delta$. Moreover, the registers $A_{j,7}$ carry an additive difference.

Finally, since our algorithm outputs 2^{32} solutions with a complexity of $2^{30} \cdot P_\delta^6$ hash computations, the average cost of computing a solution pair is about P_δ^6.

4.2 Choosing the difference

In the two previous paragraphs, we saw that a useful difference has to fulfill two constraints. The first one is that there must be a quadruplet solution to the spreading problem for the four streams of FORK-256 in the fourth round. The second one is that its probability P_δ of propagating without modification should be as high as possible. We checked all differences with Hamming weights one and two, and we finally chose the difference $\delta =$ `0x00000404`. For this choice of difference we have P_δ of about 2^{-3}, and a possible set of target values for each stream is:

$$
\begin{array}{llll}
E_{1,3} = \texttt{0x030e9c3f}, & E_{2,3} = \texttt{0x7e24de5c}, & E_{3,3} = \texttt{0x00fa4d1e}, & E_{4,3} = \texttt{0x20b7363f}, \\
F_{1,3} = \texttt{0xa4115fb0}, & F_{2,3} = \texttt{0x10276030}, & F_{3,3} = \texttt{0x35edee6e}, & F_{4,3} = \texttt{0xefc6172f}, \\
G_{1,3} = \texttt{0x22c18168}, & G_{2,3} = \texttt{0x4db27e00}, & G_{3,3} = \texttt{0xd81cdc6c}, & G_{4,3} = \texttt{0x8c2c7c00}, \\
H_{1,3} = \texttt{0x1816822c}, & H_{2,3} = \texttt{0x27e004db}, & H_{3,3} = \texttt{0xcdc6bd82}, & H_{4,3} = \texttt{0xc7bff8c3}.
\end{array}
$$

4.3 Near collision for a seven rounds reduced version of FORK-256

We focus on a seven rounds reduced version of FORK-256: the two additions, the xorings, and the feed-forward are kept but the eighth round is removed. It may appear that we can find a collision against this seven rounds reduced version of FORK-256. This is not exactly true because of the final operations: the additions of the first two streams and the last two streams, and the xoring of the result. Indeed, we have seen that a difference remains in the internal registers $A_{j,7}$. Those differences might not be all the same, but their lowest bit set to 1 exactly at the same position as the lowest bit of the difference δ that was initially introduced, in our case the third lowest significant bit. These bits are shifted to the left by the addition in the first two streams and the last two streams, and the xoring cancel them. However, a differential bit reappears at the previous position due to the feed-forward, and there is no way to get rid of it.

We thus seek 1 bit near collisions, the probability of which has been estimated as follows. We chose a random internal state before the seventh round (i.e. the values A_6, B_6, C_6, and D_6 in each of the four streams), and ran the seventh step transformation, plus the recombination mechanism. After 2^{32} experiments, there was, on the average, 8.96 non zero bits. The probability of a 1 bit near-collision has been evaluated to 2^{-15} (127665 outputs out of the 2^{32} experiments were 1 bit near-collisions). Since the algorithm given in the previous section outputs 2^{32} correct values in $2^{30}\,P_\delta^6 = 2^{49}$ hash computations, the complexity to find a set of 2^{17} distinct 1 bit near collisions is about 2^{49} hash computations.

5 Attacking the full version of FORK-256

Here we show how the previous attack can be extended to the complete FORK-256 hash function. Moreover, we show a practical attack against the full hash function FORK-256 in the setting of truncations for compatibility purposes.

5.1 Attacking the full FORK-256

Recall that the algorithm given in the previous section outputs exactly 2^{32} pairs for which FORK-256's outputs collide on four of the eight 32-bit words with a complexity of 2^{49} hash computations. It remains at least one bit of difference (at a fixed position) in the second word and three 32-bit words to cancel.

Recall that the probability of a 1-bit near collision on the second word was experimentally found to be 2^{-15}. Cancelling any of the three remaining 32-bit words was experimentally found to require an average of 2^{31} trials for $\delta = \texttt{0x0000404}$. Since our algorithm outputs a set of 2^{32} pairs for a work factor of 2^{49}, the overall complexity to find a 1-bit near collision is about $2^{49+93+15-32} = 2^{125}$. Similarly, the probability to find a 2-bit near collision was experimentally found to be less than 2^{-10} so that the overall complexity to find a 2-bit near collision is about $2^{49+93+10-32} = 2^{120}$.

5.2 Truncation proposals

In view of the recent attacks against hash functions, there have been several new proposals. Those new hash function designs adopt a common consensus on the minimum size of their output: 256 bits. The side effect is that there is a multitude of applications that will need

a compatibility mode with respect to the previously widely deployed designs like MD5 [25] (128 bits) or SHA-1 [21] (160 bits). This issue has recently received some attention: it is emphasized in [10], for instance, that current ECDSA/DSA key sizes, file formats and standardized protocols currently rely on 128-bit or 160-bit hashes.

A natural approach to solve this problem is obviously to truncate the hash functions having bigger outputs, and it is suggested in [10] that most people agree with this. This intuition is supported by the fact that it can be proved to be a good approach in the random oracle model. However, as it is pointed out in [10], there might be some concern when the hash functions are practical functions in use and not perfect ideal ones. We show in the next paragraph that natural truncations of FORK-256 would lead to huge security concerns.

5.3 Attacking a truncated FORK-256

In the case of a 128-bit or even a 160-bit truncation of FORK-256, our attack becomes practical. Indeed, our algorithm produces 2^{32} pairs of inputs colliding on four of the eight output blocks with a work factor of 2^{49}. Thus, a fifth word of the output can be forced to collide for the same time complexity. It results in a pseudo-collision against two truncations with a complexity of 2^{49} hash computations, which has to be compared to the theoretically expected complexity of 2^{80} hash computations given by the birthday paradox.

5.4 Experimental results

Here we show a 2-bit near collision on a seven rounds reduced version of FORK-256 that was obtained by running[2] our algorithm given in Section 4.1 together with the difference and the set of targets of Section 4.2:

$IV =$ 0x8406e290 0x5988c6af 0x76a1d478 0x0eb60cea 0xf5c5d865 0x00001b09 0x528590bf 0xc3bf98a1
$IV' =$ 0x8406e290 0x5988cab3 0x76a1d478 0x0eb60cea 0xf5c5d865 0x00001b09 0x528590bf 0xc3bf98a1
$M =$ 0x396eedd8 0x0e8c2a93 0xb961f8a4 0xf0a06fc6 0x9935952b 0xe01d16c9 0xddc60aa4 0x0ac1d8df
0xc6fef1d8 0x4c472ca6 0x58d9322d 0x2d087b65 0x7c8e1a26 0x71ba5da1 0xba5d2bfc 0x1988f929

which enables us to exhibit a corresponding 22-bit near collision against the complete version of the FORK-256 hash function:

$IV =$ 0x8406e290 0x5988c6af 0x76a1d478 0x0eb60cea 0xf5c5d865 0x458b2dd1 0x528590bf 0xc3bf98a1
$IV' =$ 0x8406e290 0x5988cab3 0x76a1d478 0x0eb60cea 0xf5c5d865 0x458b2dd1 0x528590bf 0xc3bf98a1
$M =$ 0x396eedd8 0x0e8c2a93 0xb961f8a4 0xf0a06fc6 0x9935952b 0xe01d16c9 0xddc60aa4 0x0ac1d8df
0xc6fef1d8 0x4c472ca6 0x58d9322d 0x2d087b65 0x7c8e1a26 0x71ba5da1 0xba5d2bfc 0x1988f929

6 Additional remarks

In this section we show that the chosen IV collision attack suggested in [15] is flawed, but that the original idea of [15] can however be extended by the same message sequencing strategy that we used for our previous attacks. As a result, we show how to produce collisions against the complete FORK-256 hash function with a complexity of 2^{106} hash computations.

[2] We ran our algorithm on eight dual core processors running at about 2.5 GHz for two weeks

6.1 The original idea

This paragraph sketches the original result of [15]; we refer the reader to this paper for further details. The idea of [15] is to put an additive difference to the message block M_{12} only. According to Table 4, this message block only appears in the seventh round of the first stream and the last round of the second stream. Hence, its effect will be local to four 32 bit blocks of the output. The aim of the attack described in [15] is thus to solve the spreading problem for the obtained differences in the third and fourth stream simultaneously. The very interesting fact about this attack is that there is no difference on the IV and hence, the possibility to find a real collision on FORK-256 remains open. Actually, the authors of [15] use four words of the IV to solve the spreading problems so that the attack will result in a collision with a chosen IV.

In order to solve the spreading problem they find a sequencing to set the message blocks in two steps: M_5, M_1, M_8, M_{15}, M_0, M_{13}, and M_3[3] first (which handles the case of the fourth stream), and then M_2, M_{14}, M_{13}, M_6, M_{10}, and B_0 or $IV[1]$ (which handles the case of the third stream). This second sequence of assignments tweaked at least the correct values of M_3 and M_{13}. But those two values were already set to solve the spreading problem in the fourth stream. If the issue can be easily corrected in the case of M_3 by adjusting M_{11} which has no influence during the four first rounds in the third stream, the case of M_{13} is much more problematic.

6.2 Correcting the M_{13} issue and other improvements

In this paragraph, we suggest several improvements to fix the M_{13} issue as well as to lower the overall complexity of the attack.

First, let us study the spreading process when no difference is involved. During the step transformation, the eight registers are dealt with four by four: (A, B, C, D) on the one hand, (E, F, G, H) on the other hand. If we restrict our attention to four of them, let us say (E, F, G, H), we immediately see that the block message M acting on the input register E allows to set the output register F to any value. But what about the action of this message block on one of the three other register, like say, the output register H? The answer is that, on the average, for any input register G, there exists a value of the message block such that the output register H takes any prescribed value. That is because for a fixed value of β, the function $\psi_G : y \mapsto \big(g(y)^{\lll 9} + G\big) \oplus f(y + \beta)^{\lll 5}$ is almost a bijection for all values of G. Hence, for any fixed value of the output register H a table T_H can be built that stores values (G, y) such that $\psi_G(y) = H$. This table can be built during a precomputation step in time 2^{32} with 2^{32} memory. By building 2^{32} such tables (one for every possible value of H), it is then possible, for any given pair (G, H), to find a message so that G is indeed transformed into H during one half of the step transformation. The cost of the precomputation is now 2^{64} both in time and memory, but the access is in constant time. Obviously, such a table can be used via the freedom given by the incoming message block, to fix the value of one of the thread $F \to G$, $G \to H$, $H \to A$ only.

In the following attack, we use several such tables. The first one, T_{10}, is used to control the thread $C_{3,1} \to D_{3,2}$ through M_{10}, that is $M_{10} = T_{10}(C_{3,1}, D_{3,2})$. Another familly of tables, $T_{9,a}$, is used to determine what value of M_9 produces the expected transition

[3] M_3 is actually dependent of $IV[1] = B_0$

$E_{1,4} \rightarrow A_{1,6}$ given a fixed M_{11}, that is $M_9 = T_{9,a}(E_{1,4}, M_{11})$ so that $A_{1,6} = a$, where a is some fixed value. (We will use 253 such tables.)

Our attack strategy has the same goal as [15]: inject an additive difference in M_{12} only. To this end, we construct a sequencing allowing to set the message blocks in a way that removes the above mentioned issues, and contrary to what is done in [15], we follow the same strategy as in our previous attack: we aim to force the values of the quadruplets entering the right side of the fourth round in third stream, the right side of the first round in fourth stream and the left side of the fifth round in fourth stream. Those three quadruplets are distinct ones, because of the varying constants and the alternating role of f and g. We assume that the additive difference in register $A_{4,4}$ is the same as the one injected in M_{12}. Additionally, we note that for the difference $\delta =$ `0xdd080000`, there are exactly 253 values of a so that the difference δ does not spread from $A_{1,6}$ to $E_{1,7}$. For instance, $a - M_{12} =$ `0xe8db2d4b` is such a value.

1. *Initialize.* Set M_{12}, F_0, G_0, H_0, to correct values for top quadruplet of stream 4.
2. *Fourth stream.* Set M_1 to fix $B_{4,2}$ to its correct value. Choose a random M_5. Adjust M_8 so that difference δ propagates unchanged. Set M_{15} to fix $B_{4,3}$ to its correct value. Adjust M_0 so that difference δ propagates unchanged. Set M_{13} to fix $B_{4,4}$ to its correct value. Adjust M_{11} so that difference δ propagates unchanged, and set M_3 to fix $A_{4,4}$ to its correct value. (Quadruplet of stream 4 is correct.)
3. *Third stream.* Set M_6 to fix $F_{3,1}$ to its correct value. Choose M_7 randomly. Set M_{14} to fix $F_{3,2}$ to its correct value. Use the hash table T_{10} to set M_{10} so that $E_{3,3}$ gets its correct value. (This is possible for M_5, M_7, M_{13}, and M_{14} are already fixed.) Set M_2 to fix $F_{3,3}$ to its correct value. (Quadruplet of stream 3 is correct.)
4. *First stream.* Choose M_4 randomly. Using hash table $T_{9,a}$ for some value of a, decide which value M_9 will lead to a value of $A_{1,6}$ equal to a. This value prevents the difference on M_{12} from spreading into $E_{1,7}$ with probability 2^{-8}. If the difference spreads into $E_{1,7}$, restart step 4 with another value of a. Since there are 253 such values, difference in stream 1 does not spread to $E_{1,7}$ with high probability.

The complexity of this algorithm is about one fourth of a FORK-256 computation, with a precomputation step of complexity about $O(2^{64})$ in time and memory. For the difference `0xdd080000`, the authors of [15] have shown that at most 108 bits may differ, so we conclude that our algorithm has to be run about 2^{108} times to get a collision, which amounts to about 2^{106} FORK-256 computations.

6.3 A collision on FORK-256

In contrast with [15], we note here that we do not have to fix the IV to a specific value to produce the collision. Indeed, in [15], four 32-bit words had to be chosen, while our attack only needs three 32-bit words to be chosen. Thus, the complexity to find a message M so that $h(IV_0, M) = IV$ where IV has the three expected 32-bit words and IV_0 is FORK-256's original initial vector is 2^{96} and can be done in a precomputation step, along with the choice of M_{12} and the computation of the tables. Our attack then finds a pair (M', M'') of 512-bit message so that in the end $H(IV_0, M||M') = H(IV_0, M||M'')$. The complexity of the whole attack is thus 2^{106}.

7 Conclusion

We presented several attacks against a new hash function proposal, FORK-256. The first one is a practical attack against a truncated version of FORK-256 that requires 2^{49} hash computations. The second one is a 2-bit near collision – with predetermined positions – against FORK-256 in 2^{120} hash computations. The third one improves upon [15] to construct a collision against FORK-256 in 2^{106} hash computations with 2^{64} memory instead of 2^{128} theoretically.

References

1. Eli Biham and Rafi Chen. Near-Collisions of SHA-0. In Matthew K. Franklin, editor, *Advances in Cryptology – CRYPTO 2004*, volume 3152 of *Lecture Notes in Computer Science*, pages 290–305. Springer-Verlag, 2004.
2. Eli Biham, Rafi Chen, Antoine Joux, Patrick Carribault, Christophe Lemuet, and William Jalby. Collisions of SHA-0 and Reduced SHA-1. In Ronald Cramer, editor, *Advances in Cryptology – EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 36–57. Springer, 2005.
3. Scott Contini, Arjen K. Lenstra, and Ron Steinfeld. VSH, an Efficient and Provable Collision-Resistant Hash Function. In Serge Vaudenay, editor, *Advances in Cryptology – EUROCRYPT 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 165–182. Springer, 2006.
4. Ivan Damgrd. A Design Principle for Hash Functions. In Gilles Brassard, editor, *Advances in Cryptology – CRYPTO '89*, volume 435 of *Lecture Notes in Computer Science*, pages 416–427. Springer-Verlag, 1989.
5. Christophe Debaert and Henri Gilbert. The RIPEMD and RIPEMD Improved Variants of MD4 Are Not Collision Free. In Mitsuru Matsui, editor, *Fast Software Encryption – FSE 2001*, volume 2355 of *Lecture Notes in Computer Science*, pages 52–65. Springer, 2002.
6. T. Dierks and C. Allen. RFC 1320: The TLS Protocol version 1.0, January 1999. `http://www.ietf.org/rfc/rfc2246.txt`.
7. Hans Dobbertin, Antoon Bosselaers, and Bart Preneel. RIPEMD-160: A Strengthened Version of RIPEMD. In Dieter Gollmann, editor, *Fast Software Encryption – FSE '96*, volume 1039 of *Lecture Notes in Computer Science*, pages 71–82. Springer, 1996.
8. D. Hong, J. Sung, S. Lee, D. Moon, and S. Chee. A New Dedicated 256-bit Hash Function. In Matthew J.B. Robshaw, editor, *Fast Software Encryption – FSE 2006*, volume 4047 of *Lecture Notes in Computer Science*. Springer-Verlag, 2006.
9. Antoine Joux. Multicollisions in Iterated Hash Functions. Application to Cascaded Constructions. In Matthew K. Franklin, editor, *Advances in Cryptology – CRYPTO 2004*, volume 3152 of *Lecture Notes in Computer Science*, pages 306–316. Springer-Verlag, 2004.
10. John Kelsey. SHA-160: A Truncation Mode for SHA-256 (and most other hashes). In *Halloween Hash Bash*, 2005.
11. John Kelsey and Bruce Schneier. Second Preimages on n-Bit Hash Functions for Much Less than 2^n Work. In Ronald Cramer, editor, *Advances in Cryptology – EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 474–490. Springer-Verlag, 2005.
12. Lars R. Knudsen. SMASH – A Cryptographic Hash Function. In Henri Gilbert and Helena Handschuh, editors, *Fast Software Encryption – FSE 2005*, volume 3557 of *Lecture Notes in Computer Science*, pages 228–242. Springer, 2005.
13. Xuejia Lai, Dengguo Feng, Hui Chen, Xiaoyun Wang, and Xiuyuan Yu. Cryptanalysis of the Hash Functions MD4 and RIPEMD. In Ronald Cramer, editor, *Advances in Cryptology – EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, 2005.
14. Helger Lipmaa, Johan Walln, and Philippe Dumas. On the additive differential probability of exclusive-or. In Willi Meier and Roy Bimal, editors, *Fast Software Encryption – FSE 2004*, volume 3017 of *Lecture Notes in Computer Science*, page 317. Springer-Verlag, 2004.
15. Krystian Matusiewicz, Scott Contini, and Josef Pieprzyk. Collisions for two branches of FORK-256. Cryptology ePrint Archive, Report 2006/317, 2006. Available from `http://eprint.iacr.org/`.
16. Florian Mendel, Joseph Lano, and Bart Preneel. Cryptanalysis of Reduced Variants of the FORK-256 Hash Function. In Masayuki Abe, editor, *CT-RSA 2007*, Lecture Notes in Computer Science. Springer-Verlag, 2007. *To appear*.

17. Florian Mendel, Norbert Pramstaller, Christian Rechberger, and Vincent Rijmen. On the Collision Resistance of RIPEMD-160. In Sokratis K. Katsikas, Javier Lopez, Michael Backes, Stefanos Gritzalis, and Bart Preneel, editors, *Information Security – ISC 2006*, volume 4176 of *Lecture Notes in Computer Science*, pages 101–116. Springer, 2006.
18. Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1996.
19. Ralph C. Merkle. One Way Hash Functions and DES. In Gilles Brassard, editor, *Advances in Cryptology – CRYPTO '89*, volume 435 of *Lecture Notes in Computer Science*, pages 428–446. Springer-Verlag, 1989.
20. Frédéric Muller. Personal communication, 2006.
21. National Institute of Standards and Technology. FIPS 180-2: Secure Hash Standard, August 2002. `http://csrc.nist.gov`.
22. Norbert Pramstaller, Christian Rechberger, and Vincent Rijmen. Breaking a New Hash Function Design Strategy Called SMASH. In Bart Preneel and Stafford E. Tavares, editors, *Selected Areas in Cryptography – SAC 2005*, volume 3897 of *Lecture Notes in Computer Science*, pages 233–244. Springer, 2006.
23. Bart Preneel. Ph.d. thesis. katholieke universiteit leuven, 1993.
24. Ronald L. Rivest. RFC 1320: The MD4 Message-Digest Algorithm, April 1992. `http://www.ietf.org/rfc/rfc1320.txt`.
25. Ronald L. Rivest. RFC 1321: The MD5 Message-Digest Algorithm, April 1992. `http://www.ietf.org/rfc/rfc1321.txt`.
26. Markku-Juhani Olavi Saarinen. Security of VSH in the Real World. In Rana Barua and Tanja Lange, editors, *Progress in Cryptology – INDOCRYPT 2006*, volume 4329 of *Lecture Notes in Computer Science*, pages 95–103. Springer, 2006.
27. Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding Collisions in the Full SHA-1. In Victor Shoup, editor, *Advances in Cryptology – CRYPTO 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 17–36. Springer-Verlag, 2005.
28. Xiaoyun Wang and Hongbo Yu. How to Break MD5 and Other Hash Functions. In Ronald Cramer, editor, *Advances in Cryptology – EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 19–35. Springer-Verlag, 2005.

A Propagation of additive differences through '$\oplus$'

When studying the internal step transformation of FORK-256, the problem appears of computing the probability that a given additive difference propagates through a '$\oplus$' without being modified. An even more general version of this problem has already been studied at FSE 2004 by Lipmaa, Wallén, and Dumas [14]. We give hereafter a much weaker version of their result that fits our needs:

Property 1. Given any 32-bit word δ, the probability $P_\delta = \Pr_{x,y}\left[\big((x+\delta)\oplus y\big) = \big(x \oplus y\big)\right]$ where elements x and y are 32-bit words can be expressed as the following matrix product:

$$P_\delta = L \times M_{\delta_{31}} \times M_{\delta_{30}} \times \cdots \times M_{\delta_0} \times C, \tag{6}$$

where δ_i denotes the i-th bit of δ and L, C, M_0, and M_1 are defined as:

$$M_0 = \frac{1}{4}\begin{pmatrix} 4&0&0&1&0&1&1&0\\ 0&0&0&1&0&1&0&0\\ 0&0&0&1&0&0&1&0\\ 0&0&0&1&0&0&0&0\\ 0&0&0&0&0&1&1&0\\ 0&0&0&0&0&1&0&0\\ 0&0&0&0&0&0&1&0\\ 0&0&0&0&0&0&0&0 \end{pmatrix}, \quad M_1 = \frac{1}{4}\begin{pmatrix} 1&0&0&0&0&0&0&0\\ 1&0&0&1&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&1&0&0&0&0\\ 1&0&0&0&0&0&1&0\\ 1&0&0&1&0&4&1&0\\ 0&0&0&0&0&0&1&0\\ 0&0&0&1&0&0&1&0 \end{pmatrix},$$

$$L = (\,1\;0\;0\;0\;0\;0\;0\;0\,), \quad {}^{T}C = (\,1\;1\;1\;1\;1\;1\;1\;1\,).$$

B Additional data for FORK-256's description

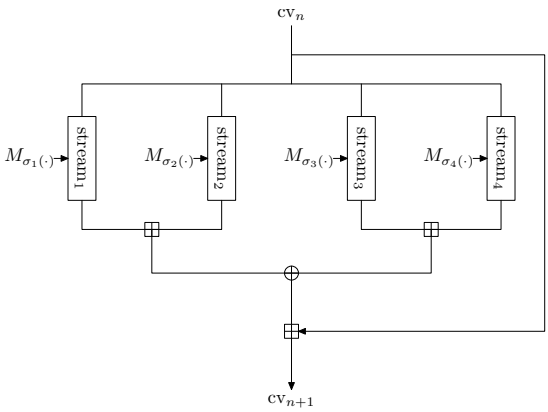

Fig. 2. FORK-256

Table 4.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\sigma_1(i)$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\sigma_2(i)$	14	15	11	9	8	10	3	4	2	13	0	5	6	7	12	1
$\sigma_3(i)$	7	6	10	14	13	2	9	12	11	4	15	8	5	0	1	3
$\sigma_4(i)$	5	12	1	8	15	0	13	11	3	10	9	2	7	14	4	6

Table 5. Constants involved in the step transformation.

step r	$\alpha_{1,r}$	$\beta_{1,r}$	$\alpha_{2,r}$	$\beta_{2,r}$	$\alpha_{3,r}$	$\beta_{3,r}$	$\alpha_{4,r}$	$\beta_{4,r}$
0	δ_0	δ_1	δ_{15}	δ_{14}	δ_1	δ_0	δ_{14}	δ_{15}
1	δ_2	δ_3	δ_{13}	δ_{12}	δ_3	δ_2	δ_{12}	δ_{13}
2	δ_4	δ_5	δ_{11}	δ_{10}	δ_5	δ_4	δ_{10}	δ_{11}
3	δ_6	δ_7	δ_9	δ_8	δ_7	δ_6	δ_8	δ_9
4	δ_8	δ_9	δ_7	δ_6	δ_9	δ_8	δ_6	δ_7
5	δ_{10}	δ_{11}	δ_5	δ_4	δ_{11}	δ_{10}	δ_4	δ_5
6	δ_{12}	δ_{13}	δ_3	δ_2	δ_{13}	δ_{12}	δ_2	δ_3
7	δ_{14}	δ_{15}	δ_1	δ_0	δ_{15}	δ_{14}	δ_0	δ_1

$\delta_0 =$ 0x428a2f98	$\delta_4 =$ 0x3956c25b	$\delta_8 =$ 0xd807aa98	$\delta_{12} =$ 0x72be5d74
$\delta_1 =$ 0x71374491	$\delta_5 =$ 0x59f111f1	$\delta_9 =$ 0x12835b01	$\delta_{13} =$ 0x80deb1fe
$\delta_2 =$ 0xb5c0fbcf	$\delta_6 =$ 0x923f82a4	$\delta_{10} =$ 0x243185be	$\delta_{14} =$ 0x9bdc06a7
$\delta_3 =$ 0xe9b5dba5	$\delta_7 =$ 0xab1c5ed5	$\delta_{11} =$ 0x550c7dc3	$\delta_{15} =$ 0xc19bf174