

# Cryptanalysis of FORK-256

Krystian Matusiewicz<sup>1</sup>, Thomas Peyrin<sup>2</sup>, Olivier Billet<sup>2</sup>, Scott Contini<sup>1</sup>,  
and Josef Pieprzyk<sup>1</sup>

<sup>1</sup> Centre for Advanced Computing, Algorithms and Cryptography,  
Department of Computing, Macquarie University

{kmatus,scontini,josef}@ics.mq.edu.au

<sup>2</sup> France Telecom Research and Development

Network and Services Security Lab

{thomas.peyrin,olivier.billet}@orange-ftgroup.com

**Abstract.** In this paper we present a cryptanalysis of a new 256-bit hash function, FORK-256, proposed by Hong *et al.* at FSE 2006. This cryptanalysis is based on some unexpected differentials existing for the step transformation. We show their possible uses in different attack scenarios by giving a 1-bit (resp. 2-bit) near collision attack against the full compression function of FORK-256 running with complexity of  $2^{125}$  (resp.  $2^{120}$ ) and with negligible memory, and by exhibiting a 22-bit near pseudo-collision. We also show that we can find collisions for the full compression function with a small amount of memory with complexity not exceeding  $2^{126.6}$  hash evaluations. We further show how to reduce this complexity to  $2^{109.6}$  hash computations by using  $2^{73}$  memory words. Finally, we show that this attack can be extended with no additional cost to find collisions for the full hash function, i.e. with the predefined IV.

**Keywords:** hash functions, cryptanalysis, FORK-256, micro-collisions.

## 1 Introduction

Most of the dedicated hash functions published in the last 15 years follow more or less closely ideas used by R. Rivest in the design MD4 [13,14] and MD5 [15]. Using terminology from [16], their step transformations are all based on source-heavy Unbalanced Feistel Networks (UFN) and employ bitwise boolean functions. Apart from MD4 and MD5 other examples include RIPEMD [12], HAVAL [21], SHA-1 [10] and also SHA-256 [11]. A very nice feature of all these designs is that they are very fast in software implementations on modern 32-bit processors and only use a small set of basic instructions executed by modern processors in constant-time like additions, rotations, and boolean functions [4].

However, traditional wisdom says that monoculture is dangerous, and this proved to be also true in the world of hash functions. Ground-breaking attacks on MD4, MD5 by X. Wang *et al.* [19,17] were later refined and applied to attack SHA-0 [20] and SHA-1 [18] as well as some other hash functions. Since source-heavy UFNs with Boolean functions seem to be susceptible to attacks similar to Wang's because only one register is changed after each step and the attacker can

manipulate it to a certain extent, one could try designing a hash function using the other flavour of UFNs, namely target-heavy UFNs where changes in one register influence many others. This is the case with the hash function Tiger [1] tailored for 64 bit platforms and designed in 1995, and a recently proposed FORK-256 [3] which is the focus of this paper.

The paper is organized as follows. In the next section, we briefly describe FORK-256. Then, in Section 3, we discuss some properties of the step transformation of the compression function. In Section 4 we investigate a special kind of rather pathological differentials in the step transformation. We analyse those differentials in details and derive an efficient necessary and sufficient condition for their existence. Effectiveness of this test allows a fast research of suitable configurations. Section 5 studies simple paths using those differentials and shows how to use them to efficiently find near-collisions for the compression function. In Section 6, we then show how to exploit local differentials studied in Section 4 to construct a high-level differential path for the full function as well as for its various simplified variants. Finally, in Section 7 we present two algorithms for finding collisions against FORK-256's compression function, and show in Section 8 how this method can be extended to find collisions for the full hash function.

**Notation.** Throughout the paper we use the following notations. Unless stated otherwise, all words are 32-bit words and are sometimes thought of as elements of  $\mathbb{Z}_{2^{32}}$  or  $\mathbb{Z}_2^{32}$ .

$x + y$	addition in $\mathbb{Z}$ or $\mathbb{Z}_{2^{32}}$ depending on the context,
$x - y$	subtraction in $\mathbb{Z}$ or $\mathbb{Z}_{2^{32}}$ ,
$x \oplus y$	bitwise xor of two words,
$x \lll a$	rotation of bits of the word $x$ by $a$ positions to the left,
$R_{j,i}$	value of register $R \in \{A, \dots, H\}$ in branch $j = 1, \dots, 4$ at step $i$ ,
$h_w(x)$	Hamming weight of word $x$ .

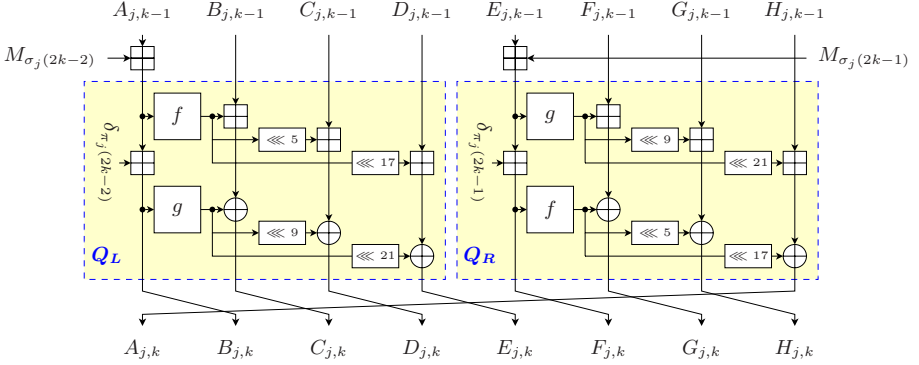
## 2 Description of FORK-256

FORK-256 is a new dedicated hash function proposed by Hong *et al.* [3,2]. 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  $M$  to 256 bits of a new state  $CV_{n+1}$ . For the complete description we refer to [3].

The compression function uses a set  $\{\text{BRANCH}_j\}_{j=1,2,3,4}$  of four branches running in parallel, each one of them using a different scheduling of sixteen 32 bit message blocks  $M_i$ ,  $i = 0, \dots, 15$  by permuting them through  $\sigma_j$ . The same set of chaining variables  $CV = (A_0, B_0, C_0, D_0, E_0, F_0, G_0, H_0)$  is used in the four branches. After computing outputs of parallel branches  $h_j = \text{BRANCH}_j(CV, M)$  the compression function updates the set of chaining variables according to the formula

$$CV := CV + [(h_1 + h_2) \oplus (h_3 + h_4)] ,$$

where ‘+’ and ‘ $\oplus$ ’ are performed word-wise. This construction can be seen as further extension of the design principle of two parallel lines used in RIPEMD [12].



**Fig. 1.** Step transformation of branch  $j$  of FORK-256.  $Q$ -structures are marked with frames.

Each branch function  $\text{BRANCH}_j$ ,  $j = 1, 2, 3, 4$  consists of eight steps. In each step  $k = 1, \dots, 8$  the branch function updates its own copy of eight chaining variables using the step transformation depicted in Fig. 1.  $R_{j,i}$  denotes the value of the register  $R \in \{A, \dots, H\}$  in  $j$ -th branch after step  $i$  and all  $A_{j,0}, \dots, H_{j,0}$  are initialised with corresponding values of eight chaining variables  $A_0, \dots, H_0$ .

The functions  $f$  and  $g$  are defined as  $f(x) = x + (x \lll 7 \oplus x \lll 22)$  and  $g(x) = x \oplus (x \lll 13 + x \lll 27)$ , respectively. Finally, the constants  $\delta_0, \dots, \delta_{15}$  are given in Table 6 and permutations  $\sigma_j$  and  $\pi_j$  are defined in Table 7 of Appendix A.

### 3 Preliminary Observations on FORK-256

As seen in the previous section, FORK-256 uses four parallel branches 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 first reported efforts to break it was limited to two of the four branches [8]. In other words, the main difficulty in cryptanalysing 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. There are however some specific differential characteristics of interest.

The first one, as noted by [9] and [8], overcomes the issue by applying the same *modular* difference  $d$  to every message block. Hence, just after the fourth step has been completed, if the internal state has the same difference  $d$  on all of its eight 32-bit words, there is a collision after the eighth step. This behavior, summarized in Table 1, renders the use of four branches with message reordering as a mean to protect against differential analysis ineffective since the same difference is applied to every message block and the same differential pattern is occurring

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

step	$\Delta A$	$\Delta B$	$\Delta C$	$\Delta D$	$\Delta E$	$\Delta F$	$\Delta G$	$\Delta H$	$\Delta M_L$	$\Delta M_R$	Prob.
in	$d$	$d$	$d$	$d$	$d$	$d$	$d$	$d$	$-d$	$-d$	
1	$d$	0	$d$	$d$	$d$	0	$d$	$d$	$-d$	$-d$	$P_d^6$
2	$d$	0	0	$d$	$d$	0	0	$d$	$-d$	$-d$	$P_d^4$
3	$d$	0	0	0	$d$	0	0	0	$-d$	$-d$	$P_d^2$
out	0	0	0	0	0	0	0	0			1

simultaneously in the four branches. The probability  $P_d$  is the probability that the difference  $d$  propagates without modification in one step. This comes from the fact that the modular difference has to pass through a ‘ $\oplus$ ’. Indeed, modular 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). This probability can be computed exactly for any given difference  $d$ , and this computation is given in Appendix B. Note that it does propagate without modification when it enters the step in the register  $A$  or the register  $E$  of the internal state, but with probability  $P_d$  otherwise. The overall probability for the differential pattern of Table 1 to occur is thus  $P_d^{12}$  for each branch.

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

step	$\Delta A$	$\Delta B$	$\Delta C$	$\Delta D$	$\Delta E$	$\Delta F$	$\Delta G$	$\Delta H$	$\Delta M_L$	$\Delta M_R$	Prob.
in	0	$d$	0	0	0	0	0	0	0	0	
1	0	0	$d$	0	0	0	0	0	0	0	$P_d$
2	0	0	0	$d$	0	0	0	0	0	0	$P_d$
3	0	0	0	0	$d$	0	0	0	0	0	$P_d$
4	0	0	0	0	0	$d$	0	0	0	0	$P_d$
5	0	0	0	0	0	0	$d$	0	0	0	$P_d$
6	0	0	0	0	0	0	0	$d$	0	0	$P_d$
out	$d$	0	0	0	0	0	0	0			$P_d$

Another way to deal with the four branches simultaneously is to apply a difference on the IV instead of the message  $M$ . This type of collisions is called a pseudo-collision. For the compression function  $h$  of any iterated hash function, a pseudo-collision can be expressed as  $h(IV, M) = h(IV', M')$ , 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 states 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 step. 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 IV, the difference propagates without spreading during three steps. Note that it propagates without being modified with probability  $P_d$  only, just as for the first differential pattern. During the fourth step however, the difference most likely spreads to the three internal

registers  $F$ ,  $G$ , and  $H$  in all four branches. However, we show in the next section that there is a way to prevent the spread of the difference from registers  $A$  and  $E$ .

## 4 Micro-collisions in $Q_L$ and $Q_R$

The step transformation described in Section 2 can be logically split into three parts: addition of message words, two parallel mixing structures  $Q_L$  and  $Q_R$  and a final permutation of registers (see Fig. 1). The key role is played by the two structures  $Q_L$  and  $Q_R$  as they are the main source of diffusion in the compression function.

In the next paragraphs, we describe a way of finding differentials of the form  $(\Delta A, 0, 0, 0) \rightarrow (\Delta A, 0, 0, 0)$  in  $Q_L$  and show that it works for  $Q_R$  as well. The idea is to look for pairs of inputs to the register  $A$  and appropriate input values of registers  $B$ ,  $C$  and  $D$  such that the output differences in registers  $B$ ,  $C$ ,  $D$  are equal to zero in spite of non-zero differences at the outputs of functions  $f$  and  $g$ . Such situation is possible if we have three simultaneous *micro-collisions* i.e. differences in  $g$  cancel out differences from  $f$  in all three registers  $B$ ,  $C$ ,  $D$ .

### 4.1 Necessary and Sufficient Condition for Micro-collisions

Let us denote  $y = f(x)$ ,  $y' = f(x')$  and  $z = g(x + \delta)$ ,  $z' = g(x' + \delta)$ . We have a micro-collision in the first line if the equation

$$(y + B) \oplus z = (y' + B) \oplus z' \quad (1)$$

is satisfied for given  $y$ ,  $y'$ ,  $z$ ,  $z'$  and some constant  $B$ . Our aim is to find the set of all constants  $B$  for which (1) is satisfied. Let us first introduce three different representations of differences between two numbers  $x$  and  $x'$  of  $\mathbb{Z}_{2^{32}}$ .

- The first representation is the xor difference. We treat it as a vector of  $\mathbb{Z}_2^{32}$  representing bits of  $x \oplus x'$  and denote it as  $\Delta^\oplus(x, x') \in \{0, 1\}^{32}$ .
- The second one is the integer difference between the two numbers  $x$  and  $x'$ , which we denote by  $\partial x := x - x'$ . Note that  $-2^{32} < \partial x < 2^{32}$ .
- The third one is the signed binary representation which uses digits from the set  $\{-1, 0, 1\}$ . A pair  $(x, x')$  has signed binary representation  $\Delta^\pm(x, x') = (x_0 - x'_0, x_1 - x'_1, \dots, x_{31} - x'_{31})$ , i.e. the  $i$ -th component is the result of the subtraction of corresponding  $i$ -th bits of  $x$  and  $x'$ .

A simple but important observation is that a difference with signed representation  $(r_0, r_1, \dots, r_{31})$  has a xor difference of  $(|r_0|, |r_1|, \dots, |r_{31}|)$ , that is the xor difference has ones in those places where the signed difference has a non-zero digit, either  $-1$  or  $1$ . The relationship between integer and signed binary representations is more interesting. An integer difference  $\partial x$  corresponds to a signed binary representation  $(r_0, \dots, r_{31})$  if  $\partial x = \sum_{i=0}^{31} 2^i \cdot r_i$  where  $r_i \in \{-1, 0, 1\}$ . Of course this correspondence is one-to-many because of the value-preserving transformations of signed representations  $(*, 0, 1, *) \leftrightarrow (*, 1, -1, *)$  and  $(*, 0, -1, *) \leftrightarrow$

$(*, -1, 1, *)$  that can stretch or shrink chunks of ones. Consider an example: assume we work with 4-bit words and let  $\Delta^\pm(11, 2) = (1, 0, 0, 1)$ ,  $\Delta^\pm(14, 5) = (1, 0, 1, -1)$ , and  $\Delta^\pm(12, 3) = (1, 1, -1, -1)$ . All these binary signed representations correspond to the integer difference  $\partial x = 9$ . Note that we can go from one pair of values to another by adding an appropriate constant, e.g.  $(12, 3) = (11 + 1, 2 + 1)$ . This addition preserves the integer difference but can modify the signed binary representation.

We are now equipped with the necessary tools and go back to our initial problem. Rewriting (1) as  $(y + B) \oplus (y' + B) = z \oplus z'$ , we can easily see that the signed difference  $\Delta^\pm(y + B, y' + B)$  can have non-zero digits only in those places where the xor difference  $\Delta^\oplus(z, z')$  has ones. This narrows down the set of possible signed binary representations that can “fit” into the xor difference of a particular form to  $2^{h_w(\Delta^\oplus(z, z'))}$ . But since a single signed binary representation corresponds to a unique integer difference, there are also only  $2^{h_w(\Delta^\oplus(z, z'))}$  integer differences  $\partial y$  that “fit” into the given xor difference  $\Delta^\oplus(z, z')$  and what is important, integer differences are preserved when adding a constant  $B$ .

Thus, to check whether a particular difference  $\partial y = y - y'$  may “fit” into xor difference we need to solve the following problem: given  $\partial y = y - y'$ ,  $-2^{32} < \partial y < 2^{32}$  and a set of positions  $I = \{k_0, k_1, \dots, k_m\} \subset \{0, \dots, 31\}$  (that is determined by non-zero bits of  $\Delta^\oplus(z, z')$ ), decide whether it is possible to find a binary signed representation  $r = (r_0, \dots, r_{31})$  corresponding to  $\partial y$  such that:

$$\partial y = \sum_{i=0}^m 2^{k_i} \cdot r_{k_i} \quad \text{where } r_{k_i} \in \{-1, 1\} . \quad (2)$$

Replacing  $t_i$  by  $(r_{k_i} + 1)/2$ , this equation can be rewritten in the equivalent form:

$$\partial y + \sum_{i=0}^m 2^{k_i} = 2^{k_0+1}t_0 + 2^{k_1+1}t_1 + \dots + 2^{k_m+1}t_m , \quad (3)$$

where  $t_i \in \{0, 1\}$ . Deciding if there are numbers  $t_i$  that satisfy (3) is an instance of the knapsack problem and since it is superincreasing—weights are powers of two, we can do this very efficiently. This gives us a computationally efficient necessary condition for a micro-collision in a line: if  $\partial y = y - y'$  cannot be represented as (2), no constant  $B$  exists and there is no solution to (1). Moreover, we can show that this condition is also sufficient: if we can find a solution of (2), then there exists a constant  $B$  that modifies the signed difference so that it “fits” the prescribed xor pattern.

Observe that since the solution of the superincreasing knapsack problem (3) is unique, so is the solution of (2). This means that we know the unique signed representation  $\Delta^\pm(u, u + \partial y) = (r_0, \dots, r_{31})$  that is compatible with the xor difference  $\Delta^\oplus(z, z')$  and yields the integer difference  $\partial y$ . However, a unique signed representation corresponds to a number of concrete pairs  $(u, u + \partial y)$ . If at a particular position  $j \in I$  we have  $r_j = -1$ , we know that in this position the value of  $j$ -th bit of  $u$  has to change from 1 to 0. Similarly, if we have  $r_j = 1$ , the  $j$ -th bit of  $u$  should change from 0 to 1. The rest of the bits of  $u$  (corresponding

to positions with zeros in  $\Delta^\pm(u, u + \partial y)$ ) can have arbitrary values. That way, we can easily determine the set  $\mathcal{U}$  of all such values  $u$ . It is clear that  $\mathcal{U}$  always contains at least one element.

Now, since  $u = y + B$  for all  $u \in \mathcal{U}$ , the set  $\mathcal{B}$  of all constants  $B$  satisfying (1) is simply  $\mathcal{B} = \{u - y : u \in \mathcal{U}\}$ . This reasoning also shows that if we can have a micro-collision in a line, there are  $|\mathcal{B}| = 2^{32-h_w(z \oplus z')}$  constants that yield the micro-collision if the most significant bit of  $z \oplus z'$  is zero and  $|\mathcal{B}| = 2^{32-h_w(z \oplus z')+1}$  if the MSB of  $z \oplus z'$  is one. The difference is caused by the fact that if  $31 \in I$ , we do not need to change  $u_{31}$  in a particular way (i.e. either  $1 \rightarrow 0$  or  $0 \rightarrow 1$ ) as any change is fine because we do not introduce carries.

Finally, since we didn't use any properties of functions  $f$  and  $g$ , the same argument applies not only to micro-collisions in  $Q_R$  but also to the same structure with any functions in place of  $f$  and  $g$ .

## 5 A First Attempt with a Simple Differential Path

In Section 3 we have seen that the seven step differential pattern of Table 2 with a modular difference  $d$  happens simultaneously in the four branches with probability  $P_d^{12}$  as soon as we have a micro-collision in each branch. For this, we need the registers  $(E, F, G, H)$  to reach a prescribed value at the fourth step in the four branches, which can be easily computed thanks to the method given in Section 3. Note that  $P_d$  is the probability that the modular difference is unchanged when a ' $\oplus$ ' is involved. Also, we do not care about the difference after the fourth step, because it never spreads again. So the exponent 12 comes from the fact that the difference has to go unchanged through the first three steps in the four branches. We show here how to force these registers to take on prescribed values.

### 5.1 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 branch to their required values.

To do this, we study the relationships between message blocks and IV blocks with this last quadruplet. Before getting into details of the attack, let us emphasize the following fact that simplifies the study of these relationships in the branch  $j$ . Forcing the value of the quadruplet  $(E_{j,3}, F_{j,3}, G_{j,3}, H_{j,3})$  is equivalent to setting 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 backwards in the threads of the FORK-256's step transformation, which can be translated in the following sequence of equations:

$$\begin{aligned} F_{j,2} &= (G_{j,3} \oplus f(F_{j,3})) - g(F_{j,3} - \delta_{\pi_j(5)}), \\ G_{j,2} &= (H_{j,3} \oplus f(F_{j,3}) \lll 5) - g(F_{j,3} - \delta_{\pi_j(5)}) \lll 9, \\ F_{j,1} &= (G_{j,2} \oplus f(F_{j,2})) - g(F_{j,2} - \delta_{\pi_j(3)}). \end{aligned} \tag{4}$$

**Table 3.** Relationship between the words of the quadruplets in each branch and the message blocks and IV. The symbols ‘\*’ and ‘x’ denote a degree of freedom in setting 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										*										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																

Table 3 summarizes those relationships. In the left column, there are words of the quadruplets that we would like to force to some predetermined values, and each row shows the dependence of one word on the message blocks and the IV registers.

Considering 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 branches take on the prescribed values. We can refine our algorithm to help the difference to propagate without a change. For instance, we can force the input of the function  $g$  to zero in order to be sure that the difference is unchanged.

1. *Initialize.* Choose  $A_0$  randomly.
2. *Adjust.* Do the following four assignments:  $M_0 := -(A_0 + \delta_{\pi_1(0)})$ ,  $M_{14} := -(A_0 + \delta_{\pi_2(0)})$ ,  $M_7 := -(A_0 + \delta_{\pi_3(0)})$ ,  $M_5 := -(A_0 + \delta_{\pi_4(0)})$ .
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  $F_{4,2}$  get their correct values, respectively.
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 step of FORK-256 in the branches 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  $F_{4,1}$  get their correct values, respectively.
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  $F_{3,3}$  get their correct values, respectively.



Now,  $F_{1,3}$  is assigned a random value. If this value is the correct one, continue to the next step, otherwise, return to the 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 a 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, go to Step 7, else go back to Step 6.
7. *Force word  $E_{3,3}$ .* (If this step has been run  $2^{32}$  times, go to Step 1.) Choose  $M_{13}$  at random. 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 the 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  $d$  has to propagate unmodified up to step 3. Since the probability to propagate in one branch is  $P_d$ , and taking into account the fact that we took care of it in the first step (Step 2 of the algorithm) and in two of the four branches of the second step (Step 4 of the algorithm), the overall probability is  $P_d^6$ .

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  $F_0$  to take any of the  $2^{32}$  possible values and the result remains valid. That is, our algorithm outputs  $2^{32}$  pairs  $\{(M, IV), (M, IV')\}$  such that after the seventh step differences only appear in registers  $A_{j,7}$  in all four branches.

Finally, our algorithm outputs  $2^{32}$  solutions with a complexity equivalent to  $2^{30} \cdot P_d^{-6}$  hash computations, and the average cost of computing a solution pair is thus about  $1/4 \cdot P_d^{-6}$ .

## 5.2 Choosing the Difference

In the two previous paragraphs, we saw that a useful difference has to fulfill two constraints. The first one is that micro-collisions must happen in all four branches of FORK-256 in the fourth step. The second one is that the probability  $P_d$  of propagating without modification should be as high as possible. Since differences with small Hamming weight yield bigger probabilities  $P_d$ , we checked all differences with Hamming weights one and two, and we finally chose the difference  $d = 0x00000404$ . For this choice of difference we have  $P_d$  of about  $2^{-3}$ , and a possible set of target values for each branch is:

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

### 5.3 Near-Collisions for FORK-256's Compression Function

**Seven step reduced version.** We focus on a seven step reduced version of FORK-256: the two additions, the xor, and the feed-forward are kept but the eighth step is removed except for the final permutation of registers. It may appear that we can find a collision against this seven steps reduced version of FORK-256, but this is not true. Indeed, we have seen that a difference remains in the internal registers  $A_{j,7}$ . Those differences have their lowest bit set to 1 exactly at the same position as the lowest bit of the difference  $d$  initially introduced, in our case the third lowest significant bit. These bits are shifted to the left by the addition in the first two branches and the last two branches, and the xor cancels them. However, a differential bit reappears at the previous position due to the feed-forward, and we can not 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 step (i.e. the values  $A_{j,6}$ ,  $B_{j,6}$ ,  $C_{j,6}$ , and  $D_{j,6}$  in each of the four branches), 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  $2^{30}/P_d^6 = 2^{49}$  hash computations, the complexity to find a set of  $2^{17}$  distinct 1-bit near-collisions is about  $2^{49}$  hash computations.

**Near-Collision for the full compression function.** The algorithm studied in the previous paragraphs outputs  $2^{32}$  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 one bit of difference (at a fixed position) in the second word and three 32-bit words to cancel. The probability of a 1-bit near collision on the second word was experimentally found to be  $2^{-15}$ , and cancelling any of the three remaining 32-bit words was experimentally found to require an average of  $2^{31}$  trials for  $d = 0x0000404$ . Hence 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 such a collision is about  $2^{49+93+10-32} = 2^{120}$ .

**Experimental results.** We exhibit a 22-bit near collision on FORK-256's compression function that was obtained by running our algorithm given in Section 5.1 together with the difference and the set of targets specified in Sect. 5.2. (Note that this also leads to a 2-bit near collision for the seven steps reduced version.)

```

CVn: 0x8406e290 0x5988c6af 0x76a1d478 0x0eb60cea 0xf5c5d865 0x458b2dd1 0x528590bf 0xc3bf98a1
CV'n: 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
CVn+1: 0x9897c70a 0x4e18862d 0xb4725ac1 0xcfc9f92c 0x9aa0637d 0xae772570 0x74dd4af1 0xcd444dd7
CV'n+1: 0x9897c70a 0x4e1880f9 0x1e677302 0x4c650966 0xf4792bf4 0xae772570 0x74dd4af1 0xcd444dd7

```

## 6 Finding High-Level Differential Paths in FORK-256

In this section we return to the question of finding differential paths in four branches of FORK-256 to present a general solution to that problem. If we can avoid mixing introduced by the structures  $Q_L$  and  $Q_R$  (i.e. we know how to find micro-collisions) and we can assume that differences in registers  $B, C, D$  and  $F, G, H$  remain unchanged ( $P_d = 1$ ), the only places where differences can change are registers  $A$  and  $E$ , after the addition of a message word difference. Thus, the values of registers in steps can be simply seen as linear functions of registers of the initial vector  $(A_0, \dots, H_0)$  and message words  $M_0, \dots, M_{15}$ .

If we consider the most general case and assume (very optimistically) that any two differences can cancel each other, we are in fact working over  $\mathbb{F}_2$  and differences in all registers are  $\mathbb{F}_2$ -linear combinations of differences  $\Delta A_0, \dots, \Delta H_0$  and  $\Delta M_0, \dots, \Delta M_{15}$  (which are now seen as elements of  $\mathbb{F}_2$ ). Now, output differences  $(\Delta A, \dots, \Delta H)$  of the whole compression function (with feed-forward) are also linear combinations of differences from  $S = (\Delta A_0, \dots, \Delta H_0, \Delta M_0, \dots, \Delta M_{15})$  and this can be represented by an  $\mathbb{F}_2$ -linear mapping  $(\Delta A, \dots, \Delta H) = L_{out}(S)$ . This means we can find the set  $\mathcal{S}_c$  of all vectors  $S$  of input differences that yield zero output differences at the end of the function simply as the kernel of this map,  $\mathcal{S}_c = \ker(L_{out})$ .

To minimise the complexity of the attack, we want to find high-level paths as short as possible. Since each register difference in each step is a linear function of differences  $\Delta A_0, \dots, \Delta H_0, \Delta M_0, \dots, \Delta M_{15}$  and there are only  $2^{24}$  of them, the straightforward approach is to enumerate them all and for any desirable subset of registers (e.g. for collisions in two or three branches) count the number of registers containing non-zero differences and pick those input differences  $S$  that give the smallest one. Using simple algebra and coding theory techniques we can make this process very efficient. Details can be found in [7].

Differences in registers other than  $A$  and  $E$  do not contribute to the complexity of the attack that much because they do not require finding micro-collisions. The measure based on the number of differences in registers  $A$  and  $E$  only corresponds to the number of “difficult” differentials we need to handle that require finding micro-collisions. Experiments show that there is a close correlation between the number of required micro-collisions and the overall length (number of all registers containing differences) of the differential path so it seems sufficient to use the measure based on differences in  $A$  and  $E$  only. Results of a search for such paths are presented in more details in [7], here we want to discuss an extension of this method.

### 6.1 More General Variant of Path Finding

We can generalize this approach further. Depending on whether we force a micro-collision to happen in a particular line or not, we have eight different models for each  $Q$ -structure. Using the linear model that assumes that all differences cancel each other, we can express output differences of each  $Q_L$ -structure as

$$\begin{aligned} \Delta A_{i+1} &= \Delta A_i , & \Delta C_{i+1} &= \Delta C_i + q_C \cdot \Delta A_i , \\ \Delta B_{i+1} &= \Delta B_i + q_B \cdot \Delta A_i , & \Delta D_{i+1} &= \Delta D_i + q_D \cdot \Delta A_i . \end{aligned}$$

where  $q_B, q_C, q_D \in \mathbb{F}_2$  are fixed coefficients characterizing the  $Q_L$ -structure. The same is true for  $Q_R$ -structures. This means that we have  $8^{64}$  possible linear models of FORK-256 when we allow such varied micro-collisions to happen. Allowing for micro-collisions in only selected lines decreases the number of active  $Q$ -structures, however, at the expense of additional conditions required to cancel differences coming from different parts of the structure.

Results of our search for such paths are summarized in Table 4. They show that by introducing such an extended model of  $Q$ -structures we can significantly decrease the number of necessary micro-collisions compared to the case when we require micro-collisions in all three lines simultaneously. Of special interest is the result showing that under favourable conditions, collisions can be achieved by using a single difference in  $M_{12}$  with six micro-collisions in the path. We show how to use this scenario to generate near-collisions but also collisions for the full compression function in Section 7.

**Table 4.** Minimal numbers  $m$  of  $Q$ -structures with micro-collisions for different scenarios of finding generalized high-level differential paths.  $Q$ -structures are numbered from 1 to 64 where 1 corresponds to  $Q_L$  in the first step of branch 1 and 64 to  $Q_R$  in the last step of branch 4. Notation N:110 means that in  $Q$ -structure number N input difference to  $A$  (resp.  $E$ ) propagates to the second and third register but not to fourth (e.g. to  $B, C$  or  $F, G$  resp.) For example, differential path from Fig. 2 is encoded as 13:110, 31:111, 40:000, 47:111, 50:000, 57:000.

Scenario	Branches	$m$	Differences in	active $Q$ -structures
Pseudo-collisions	1,2,3,4	5	$H_0, M_2, M_{11}$	12:000, 25:000, 35:001, 41:001, 51:010
Collisions	1,2,3,4	6	$M_{12}$	13:000, 31:001, 40:000, 47:100, 50:000, 57:000
Pseudo-collisions	1,2,3	2	$B_0, M_{12}$	8:100, 24:0
	1,2,4	3	$H_0, M_{11}$	3:000, 51:010, 60:000
	1,3,4	3	$H_0, M_2$	35:001, 44:000, 51:000
	2,3,4	3	$D_0, M_9$	36:010, 43:000, 52:000
Collisions	1,2,3	3	$M_0, M_3, M_9$	1:001, 20:010, 39:100
	1,2,4	4	$M_1, M_2$	2:001, 9:000, 25:100, 51:000
	1,3,4	5	$M_9$	10:000, 39:001, 42:001, 43:010, 59:000
	2,3,4	5	$M_3, M_9$	20:010, 27:000, 39:000, 57:000, 59:010

## 7 Collisions for the Full Compression Function

In this section we show how to use a high-level path with differences in  $M_{12}$  only presented in Section 6 in order to find very low weight output differences of the FORK-256's compression function. We then show two different strategies to find full collisions faster than the bound given by the birthday paradox.

The key observation is that if we introduce a difference in  $M_{12}$  only and are able to find micro-collisions in the first and fifth step of the fourth branch as

well as in the fourth step of the third branch, and prevent the propagation of the difference from  $A_{1,6}$  to  $E_{1,7}$  in the first branch, then the output difference is confined to registers  $B$ ,  $C$ ,  $D$ , and  $E$  of the output, i.e. to at most 128 bits. This behavior is illustrated in Fig. 2. The number of affected bits can be further decreased by a careful selection of the modular difference i.e. differences that are set on few most significant bits guarantee that the difference in output register  $B$  is confined to those most significant bits as well.

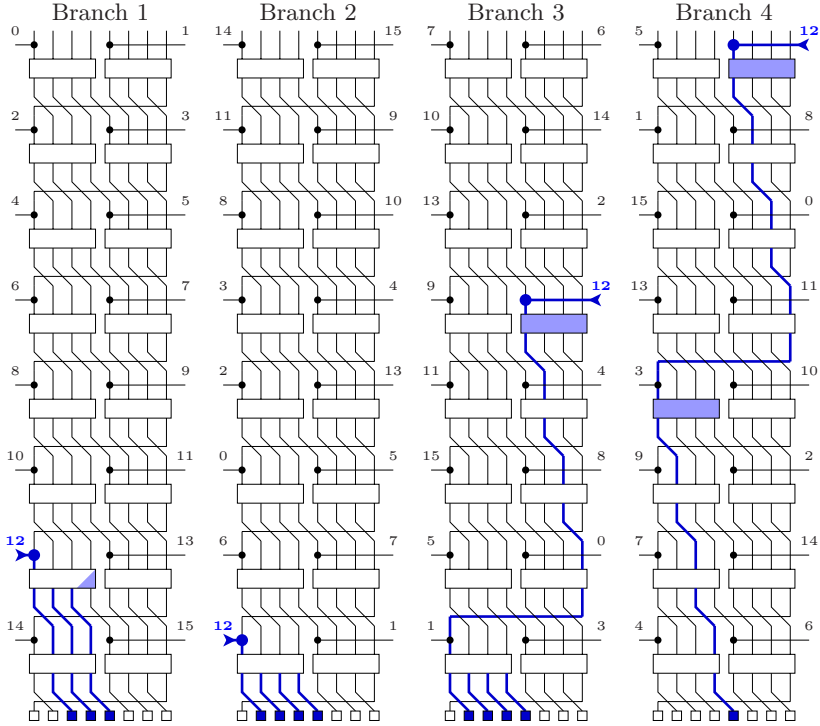
In the next paragraph, we develop our first strategy which does not require large memory. We show that pairs of messages satisfying the aforementioned constraints can be efficiently found and thus, assuming that the output differences closely follow the uniform distribution, we can expect to find very low weight differences and ultimately a collision. Finally, in the second paragraph, we use another strategy relying on precomputed tables to speed up the process of finding collisions.

## 7.1 Finding Collisions with Low Memory Requirements

The attack consists of two phases. During the first one, we find simultaneous micro-collisions at the first and fifth steps of the fourth branch as well as at the fourth step of the third branch for a modular difference injected in  $M_{12}$ . In the second phase we use free message words  $M_4$  and  $M_9$  that do not interfere with already fixed messages and micro-collisions found in the third and fourth branches in order to find messages yielding no difference in the register  $E_{1,7}$ . This is a reduced micro-collision in the single thread  $D_{1,6} \rightarrow E_{1,7}$  during the seventh step of the first branch. The description below is brief – for more details, see our implementation from [6].

**Finding micro-collisions in third and fourth branches.** Here we assume that a suitable modular difference  $d$  has already been chosen. We proceed as follows:

1. *Fourth branch, first step.* Pick  $x_1$  s.t. the pair  $(x_1, x_1 + d)$  gives simultaneous micro-collisions in  $Q_R$  at the first step of the branch four, set  $M_{12} := x_1 - E_0$  and assign the correct values to  $F_0$ ,  $G_0$ , and  $H_0$  for this micro-collision to happen.
2. *Fourth branch, fifth step.* Assign random values to  $M_5$ ,  $M_1$ ,  $M_8$ ,  $M_{15}$ ,  $M_0$ ,  $M_{13}$ , and  $M_{11}$ . Then compute the first half of the branch, up to the fifth step and find a pair of values  $(x_2, x_2 + d^*)$  (where  $d^*$  is the modular difference in register  $A_{4,4}$ ) yielding simultaneous micro-collisions in  $Q_L$ . Compute the corresponding constants  $\rho_1$ ,  $\rho_2$ , and  $\rho_3$ . If no solution exists, repeat this step, otherwise]
  - Set  $M_3 := x_2 - A_{4,4}$ .
  - Fix  $M_{13} := \rho_1 - A_{4,3} - \delta_8$  so that  $B_{4,4}$  gets its correct value  $\rho_1$ .
  - Fix  $M_{15} := [\rho_2 \oplus g(B_{4,4})] - f(B_{4,4} - \delta_8) - A_{4,2} - \delta_{10}$  so that  $C_{4,4} = \rho_2$ .
  - Similarly, fix  $M_1$  so that  $D_{4,4} = \rho_3$ .



**Fig. 2.** High level path used to find collisions for FORK-256. Thick lines show the propagation of differences.  $Q$ -structures requiring micro-collisions are greyed out. Numbers indicate message ordering.

Adjustments need to be made to  $M_0$  and  $M_{11}$  to compensate for the changes in  $M_1$  and  $M_{15}$ .

3. *Third branch.* Find a pair of values  $(x_3, x_3 + d)$  that causes simultaneous micro-collisions in  $Q_R$  in the fourth step and find the corresponding constants  $\lambda_1, \lambda_2$ , and  $\lambda_3$ . Similar to above, fix  $M_2$  so that  $F_{3,3} = \lambda_1$ ,  $M_{14}$  so that  $G_{3,3} = \lambda_2$ , and  $M_6$  so that  $H_{3,3} = \lambda_3$ . An adjustment needs to be made to  $M_{10}$  to compensate for the change in  $M_6$ . Similarly, we have to compensate for the change in  $M_{14}$  but we cannot change  $M_{13}$  since it is set in the branch four. Instead, we change  $B_0$  so that  $E_{3,3}$  gets the correct value  $x_3$ .
4. *Fourth branch.* The only modification made in the third branch that can spoil the arrangement of the fourth branch is  $B_0$ . It can only change the value  $A_{4,4}$ . Thus, adjusting  $M_{11}$  (again) is enough to put everything back into order.

At the end of this procedure, we have obtained a differential path in the third and fourth branches presented in Fig. 2. For the remaining part of the attack,

the key fact is that the values of message words  $M_4$  and  $M_9$  do not alter this path. These 64 bits of freedom are used to perform the second part of the attack.

**Single micro-collision in the first branch.** We are left with taking care of the first and second branches. Fortunately, the second branch actually does not require any attention:  $M_{12}$  appears in the very last step and so only induces differences in the output registers  $B$ ,  $C$ ,  $D$ , and  $E$ . The first branch however requires a single micro-collision in the thread  $D_{1,6} \rightarrow E_{1,7}$  during the seventh step, and it seems to us that there is no better way of finding messages causing that micro-collision than by randomly testing message words  $M_4$  and  $M_9$ . The success probability of this search heavily depends on the modular difference being used. The two best modular differences we found are displayed in Table 5.

**Table 5.** Best modular differences  $d$  we could find and their probabilities of inducing a single micro-collision in thread  $D_{1,6} \rightarrow E_{1,7}$  during the seventh step in the first branch. The number of input values  $A_{1,6}$  that may result in the micro-collision is denoted by  $\eta$ .

difference $d$	$\eta$	observed probability
0xdd080000	$2^{21.7}$	$2^{-24.6}$
0x22f80000	$2^{21.7}$	$2^{-24.6}$

Let us analyse the computational complexity of finding this single micro-collision in terms of numbers of full FORK-256 evaluations. Let  $\eta$  denote the number of allowable values for the chosen modular difference in use. By allowable value we mean an input  $x$  for which there exist three constants that cause a micro-collision for the pair  $(x, x + d)$ . For the modular difference  $d = 22f80000$  we have  $\eta = 2^{21.7}$  allowable input values.

Our algorithm to arrange the first branch correctly runs as follows:

1. *Initialize.* Fix  $M_4$  to zero.
2. *Pre-compute table.* Compute all the internal registers up to the seventh step. Then, for each allowable value  $x$ , set  $A_{1,6} = x$  and go one step backwards to get the corresponding  $H_{1,5}$ , and store the result into a hash table  $T$ .
3. *Search for  $M_9$ .* For every possible value of  $M_9$  compute the corresponding value of  $H_{1,5}$  and look for a match in  $T$ . If there is a match, go to Step 4. When all  $M_9$  are exhausted, increment  $M_4$  and go back to Step 2.
4. *Check.* If current value of  $M_9$  leads to a single micro-collision in thread  $D_{1,6} \rightarrow E_{1,7}$  then output the pair  $(M_4, M_9)$ . Continue Step 3.

Step 2 requires  $1/64$  of a full FORK-256 computation for each of the  $\eta$  allowable values. The complexity of this step is thus  $\eta/64 = 2^{15.7}$  FORK-256 evaluations. Step 3 requires  $1/64$  of full FORK-256 computation for each of the  $2^{32}$  values for  $M_9$ . The complexity of this step is thus  $2^{26}$  FORK-256 evaluations. Since Step 4 succeeds with probability  $2^{-24.6}$  (see Table 5), we get  $2^{7.4}$  solutions for a work effort of  $2^{26}$ . Hence the cost of finding a single solution with our algorithm is about  $2^{18.6}$  FORK-256 evaluations.

**Experiments.** Our C implementation of the algorithm is available for download from [6]. We conducted experiments and verified that for the difference  $d = 0\text{xdd}080000$ , the distribution of output differences on 108 affected bits (there are 109 bits that may contain differences, but we know that the differences in bit 19 of register  $B$  will always cancel out) is very close to uniform [7]. Moreover, after a few days of computations on a Pentium 4 running at 2.8 GHz, we were able to find an output difference of weight 28 [6].

**Complexity of the attack.** Since at most 108 bits are affected and the distribution of differences is close to uniform, we expect to find a collision after generating  $2^{108}$  pairs. With a work factor of  $2^{18.6}$  FORK-256 computations per pair, the total complexity required to find a collision is thus  $2^{108} \cdot 2^{18.6} = 2^{126.6}$ , which is better than the bound given by the birthday paradox. Additionally, this attack only requires about  $2 \cdot 2^{22}$  32-bit words of memory for storing precomputed inputs for micro-collisions and a hash table of similar size. It also parallelizes perfectly on many computers, each one performing independent computations starting with different seed.

The above complexity estimate is rather conservative, because if we multiply empirical probabilities of single bit differences being zero we get the value of  $2^{106.4}$  rather than  $2^{108}$  and thus also a lower complexity of the attack of  $2^{125}$  but one has to be cautious as there is no guarantee that the bits are uncorrelated enough to make this figure accurate. We refer to [7] for details.

## 7.2 Finding Collisions Faster with Precomputed Tables

In this paragraph we show how to speed up the collision search with the use of precomputed tables. To this end, let us study the spreading process when no difference is involved. During the step transformation, the eight registers are split into two subsets, namely  $(A, B, C, D)$  and  $(E, F, G, H)$ . If we restrict our attention to one of them, let us say  $(E, F, G, H)$ , we immediately see that the message block  $M$  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 registers, 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. This comes from the observation that for a fixed value of  $\delta$  and  $G$ , the function  $\psi_G : y \mapsto (g(y)^{\lll 9} + G) \oplus f(y + \delta)^{\lll 5}$  is very often a bijection. 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 pre-computation 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 pre-computation is now  $2^{64}$  both in time and memory, but access time is comparable to a single operation. Obviously, as already seen in the previous



paragraph, such a table and the freedom given by the incoming message block can be used to fix the value of one of the thread  $F \rightarrow G$ ,  $G \rightarrow H$ , and  $H \rightarrow A$  only.

In the following attack, we use a number of such tables. The first one,  $T_{10}$ , is used to control the thread  $C_{3,1} \rightarrow D_{3,2}$  through  $M_{10}$ , that is  $M_{10} = T_{10}(C_{3,1}, D_{3,2})$ . Another family of tables,  $T_{9,a}$ , is used to determine what value of  $M_9$  produces the expected transition  $E_{1,4} \rightarrow A_{1,6}$  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. (There are 36 such values for which the probability of a single micro-collision is  $2^{-8}$ , 1236 values with probability  $2^{-9}$  and many more with smaller probabilities.)

As in the previous attack, our goal is to use the high level path of Figure 2 by injecting a modular difference in  $M_{12}$  only, and to cause micro-collisions in grayed areas of this figure. To this end, we construct a sequencing allowing to set the message blocks fitting these constraints, but contrary to what is done in the previous attack, we choose the three micro-collisions of the branch three and four in advance. But now, we must ensure that the modular difference in the register  $A_{4,4}$  is the same as the one injected in  $M_{12}$ . Additionally, we note that for the difference  $d = 0\text{xdd}080000$  we are going to use, we consider around  $2^9$  values of  $a$  for which the difference  $d$  does not spread from  $A_{1,6}$  to  $E_{1,7}$  with highest probability, i.e. a single micro-collision is most likely to happen.

1. *Initialize.* Set  $M_{12}$ ,  $F_0$ ,  $G_0$ , and  $H_0$  in order to get a micro-collision in the first step of the fourth branch.
2. *Fourth branch.* Set  $M_1$  to fix  $B_{4,2}$  to its correct value. Choose a random  $M_5$ . Adjust  $M_8$  so that difference  $d$  propagates unchanged. Set  $M_{15}$  to fix  $B_{4,3}$  to its correct value. Adjust  $M_0$  so that difference  $d$  propagates unchanged. Set  $M_{13}$  to fix  $B_{4,4}$  to its correct value. Adjust  $M_{11}$  so that difference  $d$  propagates unchanged, and set  $M_3$  to fix  $A_{4,4}$  to its correct value.
3. *Third branch.* 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 because  $M_5$ ,  $M_7$ ,  $M_{13}$ , and  $M_{14}$  are already fixed.) Set  $M_2$  to fix  $F_{3,3}$  to its correct value.
4. *First branch.* Choose  $M_4$  randomly. Using the hash table  $T_{9,a}$  for some value of  $a$ , decide which value  $M_9$  will lead to the value of  $A_{1,6}$  equal to  $a$ . This value prevents the difference of  $M_{12}$  from spreading into  $E_{1,7}$  with probability at least  $2^{-9}$ . If the difference spreads into  $E_{1,7}$ , restart Step 4 with another value of  $a$ . After testing around  $2^9$  such values, difference in the first branch does not spread to  $E_{1,7}$  with a high probability.

The complexity of this algorithm is close to  $2^{1.6}$  FORK-256 evaluations if we assume access to tables in a single processor operation, with a pre-computation step of complexity about  $2^{64}$  in time and  $2^{73}$  words of memory. Since at most 108 bits of the output differ for the modular difference  $0\text{xdd}080000$ , the algorithm finds a collision in about  $2^{109.6}$  FORK-256 computations.

## 8 Compression Function's Collisions Turned into Hash Ones

Here we show that the last algorithm can be turned into collision finding algorithm for the full hash function, i.e. with a given IV. Our algorithm indeed relies on the fact that three values of IV—namely  $F_0$ ,  $G_0$ ,  $H_0$ —have specific values. By prepending a well chosen 512-bit message block to the colliding inputs for the compression function we get the expected result for the whole hash function.

Now since the targeted values are three 32-bit words, the probability to reach these value by prepending a random 512-bit message block is  $2^{-96}$ , so we need around  $2^{96}$  FORK-256 computations. This can be done after the execution of our algorithm and thus the overall complexity is dominated by  $2^{109.6}$  of the FORK-256 evaluations.

## 9 Conclusion

In this paper we exposed a number of weaknesses of the compression function of FORK-256. We studied in detail the properties of  $Q$ -structures and described very efficient algorithms to finding micro-collisions for them. We further showed how this can be exploited to mount various attacks against FORK-256's compression function. Finally, we showed that the chosen-IV collision-finding attack for the compression function can be extended to find collisions for the full hash function, i.e. with a given IV. We expect that more computational power would allow to investigate slight variations of the attacks we presented, and might improve them significantly.

Although we are intrigued by the design of FORK-256, we think it should not be used in applications that require a high level of security against collision attacks.

## Acknowledgements

The second and third authors would like to thank Sebastien Kunz-Jacques for his kind lending of cpu time as well as Gilles Macario-Rat for his help with precomputed tables in paragraph 7.2.

The project was supported by ARC grant DP0663452.

## References

1. Anderson, R., Biham, E.: Tiger: A fast new hash function. In: Gollmann, D. (ed.) FSE'96. LNCS, vol. 1039, pp. 121–144. Springer, Heidelberg (1996)
2. Hong, D., Chang, D., Sung, J., Lee, S., Hong, S., Lee, J., Moon, D., Chee, S.: A New Dedicated 256-bit Hash Function: FORK-256. In: Robshaw, M. (ed.) FSE 2006. LNCS, vol. 4047, pp. 195–209. Springer, Heidelberg (2006)
3. Hong, D., Sung, J., Hong, S., Lee, S., Moon, D.: A new dedicated 256-bit hash function: FORK-256. In: First NIST Workshop on Hash Functions (2005)

4. Intel Corporation. Intel 64 and IA-32 architectures optimization reference manual (2006) Appendix C, Instruction latency and throughput. Available from <http://developer.intel.com/design/processor/manuals/248966.pdf>
5. Lipmaa, H., Walln, J., Dumas, P.: On the additive differential probability of exclusive-or. In: Fast Software Encryption – FSE '04. LNCS, vol. 3017, pp. 317–331. Springer, Heidelberg (2004)
6. Matusiewicz, K., Contini, S., Pieprzyk, J.: Cryptanalysis of FORK-256. Web page, <http://www.ics.mq.edu.au/~kmatu/FORK/>
7. Matusiewicz, K., Contini, S., Pieprzyk, J.: Weaknesses of the compression function of FORK-256. IACR e-print Archive, report 2006/317, available from <http://eprint.iacr.org/2006/317>
8. Mendel, F., Lano, J., Preneel, B.: Cryptanalysis of reduced variants of the FORK-256 hash function. In: Abe, M. (ed.) CT-RSA 2007. LNCS, vol. 4377, pp. 85–100. Springer, Heidelberg (2006)
9. Muller, F.: Personal communication (2006)
10. National Institute of Standards and Technology. Secure hash standard (SHS). FIPS 180-1 (April 1995) Replaced by [11].
11. National Institute of Standards and Technology. Secure hash standard (SHS). FIPS 180-2 (August 2002)
12. Preneel, B., Bosselaers, A., Dobbertin, H.: RIPEMD-160: A strengthened Version of RIPEMD. In: Gollmann, D. (ed.) FSE'96. LNCS, vol. 1039, pp. 71–82. Springer, Heidelberg (1996)
13. Rivest, R.L.: The MD4 Message Digest Algorithm. In: Menezes, A.J., Vanstone, S.A. (eds.) CRYPTO 1990. LNCS, vol. 537, pp. 303–311. Springer, Heidelberg (1991)
14. Rivest, R.L.: The MD4 Message Digest Algorithm. RFC 1320, IETF (April 1992)
15. Rivest, R.L.: The MD5 Message Digest Algorithm. RFC 1321, IETF (April 1992)
16. Schneier, B., Kesley, J.: Unbalanced Feistel networks and block cipher design. In: Gollmann, D. (ed.) FSE'96. LNCS, vol. 1039, pp. 121–144. Springer, Heidelberg (1996)
17. Wang, X., Lai, X., Feng, D., Chen, H., Yu, X.: Cryptanalysis of the Hash Functions MD4 and RIPEMD. In: Cramer, R.J.F. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 1–18. Springer, Heidelberg (2005)
18. Wang, X., Yin, Y.L., Yu, H.: Finding collisions in the full SHA-1. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 17–36. Springer, Heidelberg (2005)
19. Wang, X., Yu, H.: How to break MD5 and other hash functions. In: Cramer, R.J.F. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 19–35. Springer, Heidelberg (2005)
20. Wang, X., Yu, H., Yin, Y.L.: Efficient collision search attacks on SHA-0. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 1–16. Springer, Heidelberg (2005)
21. Zheng, Y., Pieprzyk, J., Seberry, J.: HAVAL – A One-Way Hashing Algorithm with Variable Length of Output. In: Zheng, Y., Seberry, J. (eds.) AUSCRYPT 1992. LNCS, vol. 718, pp. 83–104. Springer, Heidelberg (1993)

## A Additional Details of the Specification of FORK-256

**Table 6.** Constants  $\delta_0, \dots, \delta_{15}$  used in FORK-256. They are defined as the first 32 bits of fractional parts of binary expansions of cube roots of the first 16 primes.

$\delta$	0	1	2	3	4	5	6	7
0	428a2f98	71374491	b5c0fbcf	e9b5dba5	3956c25b	59f111f1	923f82a4	ab1c5ed5
8	d807aa98	12835b01	243185be	550c7dc3	72be5d74	80deb1fe	9bdc06a7	c19bf174

**Table 7.** Message and constant permutations used in four branches of FORK-256

$j$	message permutation $\sigma_j$	permutation of constants, $\pi_j$
1	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
2	14 15 11 9 8 10 3 4 2 13 0 5 6 7 12 1	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
3	7 6 10 14 13 2 9 12 11 4 15 8 5 0 1 3	1 0 3 2 5 4 7 6 9 8 11 10 13 12 15 14
4	5 12 1 8 15 0 13 11 3 10 9 2 7 14 4 6	14 15 12 13 10 11 8 9 6 7 4 5 2 3 0 1

## B Propagation of Modular Differences Through ‘ $\oplus$ ’

When studying the internal step transformation of FORK-256, the problem appears of computing the probability that a given modular difference  $d$  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 [5]. Here we give a much weaker version of their result that fits our needs:

*Property 1.* Given any 32-bit word  $d$ , the probability

$$P_d = \Pr_{x,y} [((x + d) \oplus y) = (x \oplus y) + d]$$

where elements  $x$  and  $y$  are 32-bit words can be expressed as the following matrix product:

$$P_d = L \times M_{d_{31}} \times M_{d_{30}} \times \dots \times M_{d_0} \times C,$$

where  $d_i$  denotes the  $i$ -th bit of  $d$  and  $L$ ,  $C$ ,  $M_0$ , and  $M_1$  are defined as:

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

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