Extending FORK-256 Attack to the Full Hash Function

Scott Contini, Krystian Matusiewicz, and Josef Pieprzyk

Centre for Advanced Computing, Algorithms and Cryptography,
Department of Computing, Macquarie University
{scontini,kmatus,josef}@ics.mq.edu.au

Abstract. In a paper published in FSE 2007, a way of obtaining near-collisions and in theory also collisions for the FORK-256 hash function was presented [8]. The paper contained examples of near-collisions for the compression function, but in practice the attack could not be extended to the full function due to large memory requirements and computation time. In this paper we improve the attack and show that it is possible to find near-collisions in practice for any given value of IV. In particular, this means that the full hash function with the prespecified IV is vulnerable in practice, not just in theory. We exhibit an example near-collision for the complete hash function.

1 Introduction

Recent spectacular attacks on many established hash functions endangered most commonly used dedicated hash functions and cast some doubts on the remaining ones. This rekindled the interest in designing more secure yet still efficient alternatives. While most of the dedicated hash functions used source-heavy unbalanced Feistel networks [11], some alternatives were proposed that utilise the other option, target-heavy UFNs. One of the examples is the hash function Tiger [1] and a recent design FORK-256, proposed by Hong et al. [5,6].

Soon after FORK-256 was presented, works [9,7] showed that the step transformation has a particular weakness that may threaten the function. Indeed, soon after those ideas were refined and the attack on the full compression function was presented [8], including example near-collisions [3]. Section 8 of the paper [8] briefly mentions how to extend the result to the full compression function, but there is a mistake in the description (see Section 3 of this paper). Additionally, a cost based analysis [2] was never considered and from this viewpoint the attack suffers due to the large memory requirements. In fact, the combination of large memory and long running time preclude the idea from being implemented to find near-collisions in practice.

This paper. In this paper we correct our mistake from [8] and give an improved method for finding near-collisions (and full collisions) for any given IV. Our method modifies the algorithm from [8] in order to keep the memory usage low and improve the efficiency of one phase of the attack. Consequently, we are able to actually implement the algorithm to produce near-collisions for the full FORK-256 with the real IV. We give an example near-collision.

In Section 2 we define some notations and give a brief description of FORK-256. In Section 3, we briefly recall the original attack from [8]. Section 4 contains our main contribution in which we explain the new version of the algorithm including a detailed analysis. Finally, we present an example of a near-collision with the IV specified by the designers and then we conclude our work.

2 Brief Description of FORK-256

FORK-256 is a dedicated hash function based on the classical Merkle-Damgård iterative construction with the compression function that maps 256 bits of state and 512 bits of message to 256 bits of a new state. Here we give a concise description – more details can be found in [5].

The compression function consists of four parallel branches BRANCH_j, $j = 1, 2, 3, 4$, each one of them using a different permutation σ_j of 16 message words M_i, $i = 0, \ldots, 15$.

The same set of eight chaining variables

$$\mathrm{CV}_\ell = (A_0, B_0, C_0, D_0, E_0, F_0, G_0, H_0)$$

is input to the four branches. After computing outputs of parallel branches

$$h_j = \text{BRANCH}_j(\mathrm{CV}_\ell, M), \quad j = 1, \ldots, 4,$$

the compression function updates the set of chaining variables according to the formula

$$\mathrm{CV}_{\ell+1} := \mathrm{CV}_\ell + [(h_1 + h_2) \oplus (h_3 + h_4)] \ ,$$

where modular and XOR additions are performed word-wise. Before the first application of the compression function registers $\mathrm{CV}_0 = (A_0,\ldots,H_0)$ are initialised by appropriate constants presented in Table 3.

Each branch function BRANCH_j, $j = 1, 2, 3, 4$ consists of eight steps. In each step $k = 1, \ldots, 8$ the branch function updates its own copy of eight chaining variables using the step transformation depicted in Fig. 1.

We will denote the value of register R in j-th branch after step i as $R_i^{(j)}$.

Before the computation of j-th branch, all $A_0^{(j)}, \ldots, H_0^{(j)}$ are initialised with corresponding values of eight chaining variables.

Note that the crucial role in the step transformation play two so-called Q-structures, marked in the picture with grey.

Functions f and g mapping 32-bit words to 32-bit words are defined as

$$f(x) = x + \left(ROL^7(x) \oplus ROL^{22}(x)\right) \ ,$$
$$g(x) = x \oplus \left(ROL^{13}(x) + ROL^{27}(x)\right) \ .$$

Constants $\delta_0, \ldots, \delta_{15}$ used in each step are defined as the first 32 bits of fractional parts of binary expansions of cube roots of the first 16 primes and are presented in Table 4. Finally, permutations σ_j of message words and permutations π_j of constants are shown in Table 1.

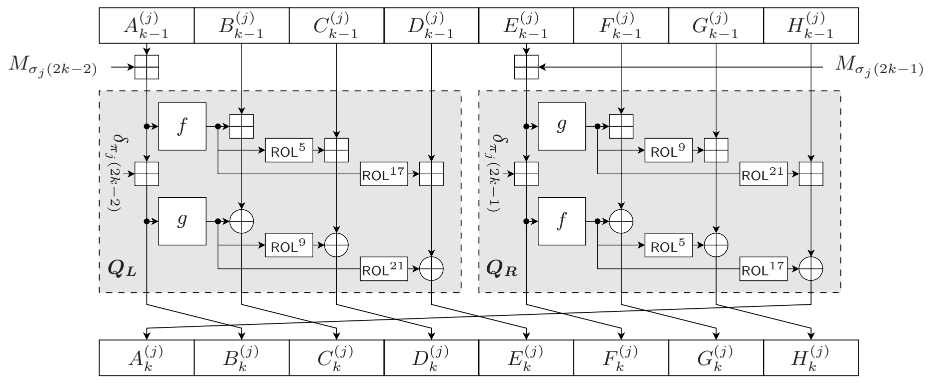

Fig. 1. Step transformation of a single branch of FORK-256. Q-structures are greyed out.

Table 1. Message and constant permutations used in four branches $j = 1, \ldots, 4$ of FORK-256

j	message permutation σ_j	permutation of constants, π_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

3 Attack on the Compression Function of FORK-256

In this section we recall the main points of the attack on the compression function of FORK-256 presented in [8] that our attack builds on.

The first essential fact is that it is possible to relatively easily find situations when non-zero differences in registers A and E do not spread to other registers during the step transformation. In other words, it is possible to obtain characteristics of the form $(\Delta A, 0, 0, 0, 0, 0, 0, 0) \rightarrow (0, \Delta B, 0, 0, 0, 0, 0, 0)$ and $(0, 0, 0, 0, \Delta E, 0, 0, 0) \rightarrow (0, 0, 0, 0, 0, \Delta F, 0, 0)$ without resorting to cancelling the difference by appropriate message word difference (cf. Fig. 1). Such characteristics are called micro-collisions and they are possible if the right difference is fed to the register A (or E) and appropriate corresponding "constants" B, C and D (F, G, H correspondingly) are set. Details on how those differences and constants can be found are presented in [8].

The second important ingredient is the possibility of using micro-collisions to find differential paths spanning the whole function that can be used to obtain collisions for the complete compression function. One such path, used in the original FORK-256 attack utilises difference in message word M_{12} only and is presented in Fig. 2.

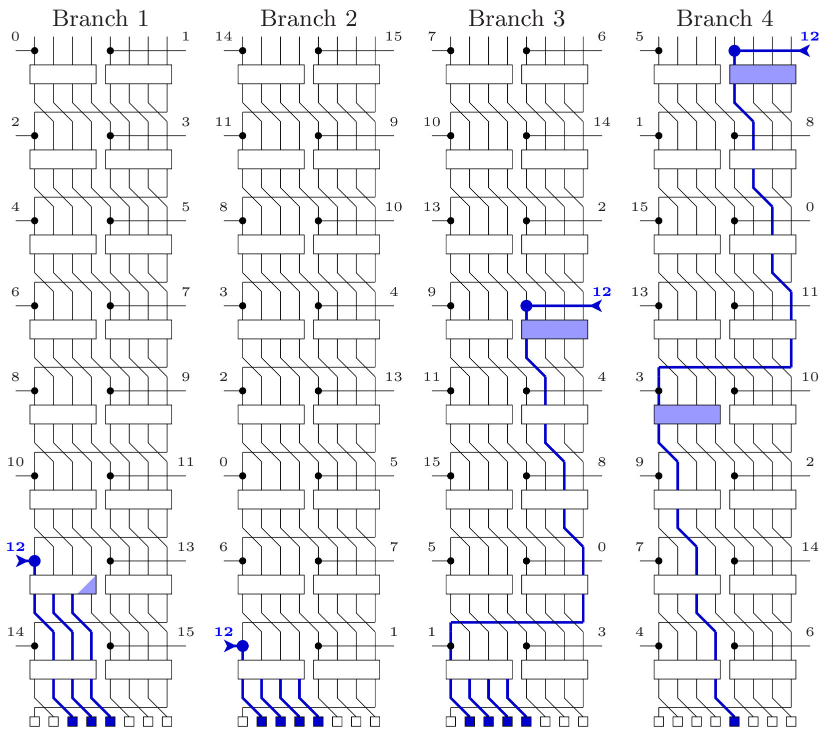

Fig. 2. High-level path used to find near-collisions in FORK-256. Thick lines show the propagation of differences. Indices of the message words that are fed into each step transformation are given in the left and right columns of each branch. Q-structures for which micro-collisions have to be found are greyed out.

The original attack from [8] first shows how to find chosen-IV near-collisions (or full collisions) and then briefly suggests a way of extending it to an attack on the full FORK-256. For now, we only focus on the low-memory version of the attack: The reason for this will be evident later.

The idea is to first choose an appropriate difference for M_{12}, then make branches 3 and 4 work (see Fig. 2) and then use free message words to get branch 1 and 2 to work. Initially all message words can be anything. To get branch 4 to work, one manipulates the values of registers F_0, G_0, H_0 and message words M_5, M_1, M_8, M_{15}, M_0, M_{13}, M_{11} in order to get the micro-collisions in the two Q-structures. Then to get branch 3 to work, message words M_6, M_{10}, M_{14}, and M_2 are manipulated along with register B_0. The change in B_0 upsets the branch 4, but by manipulating M_{11} again, the characteristic through both branches holds.

For the remaining part of the characteristic (branches 1 and 2), the main observation is that message words M_9 and M_4 can be freely changed without upsetting branches 3 and 4. This gives 2^{64} possibilities for satisfying the full characteristic. In fact, there are no requirements for branch 2, and only a single microcollision needs to happen for branch 1 at $D_6^{(1)} \rightarrow E_7^{(1)}$ (step 7). Satisfying this microcollision can be left to chance. With a precomputation trick and a good choice of difference in M_{12}, [8] finds pairs of outputs that differ in at most 108-bits (differences in registers C, D, F and part of B only) in time equivalent to $2^{18.6}$ FORK-256 operations. Such outputs can be called near-collisions. For $2^{108} \cdot 2^{18.6} = 2^{126.6}$ work, full collisions are expected. The result is faster than birthday attack.

However, it must be emphasised that these are *chosen-IV* near-collisions and collisions. It requires choosing values for B_0, F_0, G_0, and H_0. In Section 7.2 of the paper, a way to eliminate the need to choose B_0 is suggested, though it uses large precomputation tables – on the order of 2^{73} words of memory. Assuming the choice of B_0 can be eliminated, [8] argues in Section 8 that real collisions can be found by prepending a message block that yields the right values of F_0, G_0, and H_0 when that message block is sent through the compression function with the real IV defined in FORK-256. But it claims that finding this message block can be done *after* the execution of the algorithm that finds the chosen-IV collision. This is not correct. The characteristic depends upon *all* chaining variable regsisters. In other words, it is not only F_0, G_0, and H_0 that have to match the inputs to the chosen-IV collision, but also A_0 through E_0. This is easy to see: if one has a near-collision such as any one given from [8], you cannot change an input register value and still have the same near-collision because the difference propagates rapidly.

There is a simple fix for the error in Section 8. The requirements for F_0, G_0 and H_0 are dictated by the predetermined difference in M_{12}. Thus, one can process the prepended message block first: simply try random first messages blocks until allowable values for F_0, G_0, and H_0 are found. Then, one can execute the search algorithm to determine a second message block that yields a partial collision/full collision for the chaining variables determined from the prepended block. More details are in Section 4.2.

It would be nice to implement the attack to show that it works and can at least produce near-collisions, thus showing that there are real problems with FORK-256 (as opposed to attacks that are of theoretical interest only). Note that the low-memory version cannot be implemented because of the requirement on B_0, which would amplify the running time significantly (beyond what can be done in on a typical PC using a reasonable amount of CPU time). Neither can the large memory version since computers with 2^{73} of memory do not yet exist. Moreover it is claimed in [8] that finding the right values of F_0, G_0, and H_0 takes 2^{96} steps.

Our new contribution is to present a simplified and improved near-collision (and collision) search algorithm which does not use large memory and can be ran on a typical PC to produce near-collisions on the full FORK-256 with specified IV within a few days of run time. The simplified algorithm is a modification of the low-memory attack from [8]. We ran our new algorithm and found several near-collisions on the full FORK-256 with the real IV.

4 Improving the Attack

The obstacle for extending the low-memory attack from [8] to the full hash function is the requirement for particular values of four chaining values, B_0 required by branch 3 and F_0, G_0, H_0 required by branch 4. Nothing can be done about constants necessary to achieve micro-collision in the first step of branch 4. However, by careful modification of some steps of the procedure we can eliminate the need for choosing the value of the constant B_0.

4.1 The Algorithm

Instead of solving for branch 4, then branch 3, and later making a small adjustment to branch 4 again, the idea is to go through the first step of branch 4 only, then switch to branch 3, and finally return to solve for the rest of branch 4.

Let d denote the modular difference used in M_{12}. Recall that an allowable value x is a value fed to register A (or E) such that there exist constants B, C, D (or F, G, H) that cause simultaneous micro-collisions to happen in all three lines when $x, x + d$ are the values of register A (or E). The modified algorithm first precomputes for difference d all allowable values for step 5 of the left Q-structure of branch 4. Then, the steps are as follows:

Branch 4, step 1. We find x_1 such that $x_1, x_1 + d$ give simultaneous g-δ_{15}-f micro-collisions for step 1 of branch 4, compute corresponding constants τ_1, τ_2, τ_3 and assign $F_0 := \tau_1$, $G_0 := \tau_2$, $H_0 := \tau_3$. Set M_{12} to $x_1 - E_0$ and M'_{12} to $x_1 - E_0 + d$.

Branch 3. We choose values of M_7, M_6, M_{10}, M_{14}, M_{13} and M_2 appearing in the first three steps of branch 3 randomly and compute the function up to the beginning of step 4. We check if the value $E_4^{(3)} + M_{12}$ is an allowable value for the g-δ_6-f micro-collision in step 4, i.e. we test if there exist constants μ_0, μ_1, μ_2 such that the pair $E_4^{(3)} + M_{12}$, $E_4^{(3)} + M_{12} + d$ yields micro-collisions when those constants are set in registers $F_3^{(3)}$, $G_3^{(3)}$, $H_3^{(3)}$. If it is not, we pick fresh values of the message words and repeat the process. Once we get the right values (this needs around 2^{23} trials using the difference from [8]) we modify values of M_6, M_{10}, M_{14}, M_{13} and M_2 to adjust the values of $F_3^{(3)}$, $G_3^{(3)}$, $H_3^{(3)}$ to appropriate constants μ_0, μ_1, μ_2. This modification is similar to the original except here we are required to modify M_{13}, whereas the original algorithm avoided it because it was set in branch 4 (instead the original algorithm modified B_0). Now branch 3 is ready.

Branch 4, steps 2–4. We start with choosing random values for M_5, M_1, and M_{15}. Then values of M_8, M_0, and M_{11} are chosen to preserve the subtraction difference d through the first 4 steps of the characteristic. This is easy to do, for example, by setting the message blocks so that the input to the f function is zero (the output of the f function is the only thing that can change the subtraction difference). Then we compute up to the beginning of step 5. Next, we use our precomputed table to loop through all choices of M_3 that lead to allowable values and we test each one to see if any of them does not cause a difference propagation to $C_5^{(4)}$ for the current value of $B_4^{(4)}$ that is there. In other words, we are looking for a value of M_3 that actually induces a single micro-collision in line B and has the potential to cause simultaneous micro-collisions in the other two lines. This is illustrated in Fig. 3. If no solution is found, then we go back to solve for branch 3 again with new random values.[1]

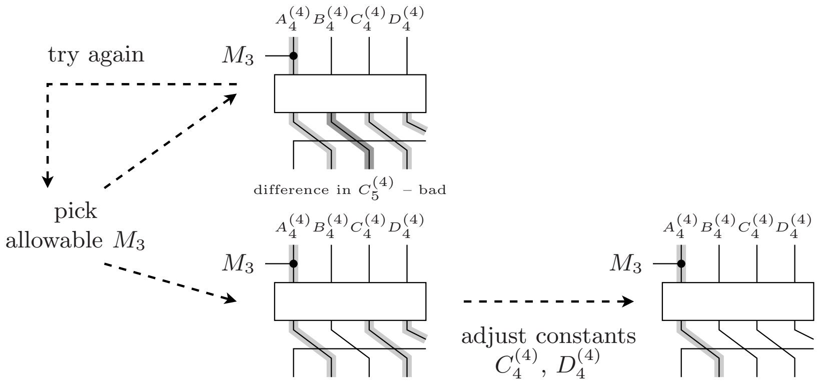
$A_4^{(4)}$ $B_4^{(4)}$ $C_4^{(4)}$ $D_4^{(4)}$
try again
M_3
difference in $C_5^{(4)}$ – bad
pick allowable M_3
$A_4^{(4)}$ $B_4^{(4)}$ $C_4^{(4)}$ $D_4^{(4)}$
M_3
adjust constants $C_4^{(4)}$, $D_4^{(4)}$
$A_4^{(4)}$ $B_4^{(4)}$ $C_4^{(4)}$ $D_4^{(4)}$
M_3

Fig. 3. Illustration of the procedure used in step 5 of branch 3. We want to get micro-collisions in all three lines without the need for modifying the value of $B_4^{(4)}$.

Once such a solution is found, we have to set the values of $C_4^{(4)}$ and $D_4^{(4)}$ to appropriate constants so that we obtain simultaneous micro-collisions for all three lines. We do this by adjusting the values of M_1, and M_{15} and appropriately compensating for these changes by adjusting M_0 and M_{11}. After this is done, branches 3 and 4 are ready.

Branches 1 and 2. The part of the algorithm that deals with branches 1 and 2 is identical to the one presented in [8] and it does not require any further explanations.

In the original attack [8], the search complexity for a near-collision is dominated by branches 1 and 2. The search through branches 1 and 2 involved 2^{64} potential characteristics for the cost of 2^{58} FORK-256 operations. Provided that the cost of our modified algorithm for branches 3 and 4 is less than this, the overall complexity is unchanged.

With the difference of $d =$`0x22f80000` the probability of passing step 4 of branch 3 is about 2^{-24} and the probability of passing step 5 in branch 4 is about 2^{-19}. The cost of a single check is about eight steps of FORK-256, so 2^{-3} full FORK-256 evaluations. Thus, passing branches 3 and 4 in our modified algorithm requires about 2^{40} FORK-256 evaluations. Hence, it does not influence the final complexity of the attack.

[1] We cannot repeat Branch 4 again since we will always end up with the same value for $B_4^{(4)}$.

4.2 Fixing Appropriate Chaining Values

So far we have removed the need for the fourth initial chaining value to be fixed. This leaves us with three 32-bit words, each one to be set to one of the possible constants required by simultaneous micro-collisions in step 1 of branch 4. This means that by prepending a random message block and computing the digest that in turn becomes the chaining value for the main part of the attack we have the probability of getting the right values of those registers at least 2^{-96}, less than $2^{126.6}$ required for the second phase. However, we can do much better when we use the fact that *any* of the possible constants will suffice in each of the three initial registers.

Let $\mathcal{A}$ be the set of allowable values for g - δ_{15} - f micro-collision in step 1 of branch 4 for a given difference d. For each allowable value $a \in \mathcal{A}$ we can compute sets $\mathcal{F}_a, \mathcal{G}_a, \mathcal{H}_a$ of constants that yield a micro-collision in the corresponding line. Then, the probability that a randomly selected triple constitute good constants for some allowable value a is

$$P = 1 - \prod_{a \in \mathcal{A}} \left(1 - \frac{|\mathcal{F}_a| \cdot |\mathcal{G}_a| \cdot |\mathcal{H}_a|}{2^{96}} \right)$$

This probability depends on the choice of the difference d. For both differences $d =$ `0xdd080000` and $d =$ `0x22f80000` used in [8] it is equal to $P = 2^{-64.8}$, but there are other differences with much higher values of P. Of course those differences may give worse performance in the main part of the attack because they are not tuned to yield optimal chance of passing requirements of branch 1. What really matters though is that original differences are suitable for the improved attack.

4.3 Experimental Results

We implemented this modified strategy and tested it. As an example, we present in Table 2 a pair of messages that give a near-collision of weight 42 of the full hash function FORK-256. Here we used difference $d =$`0x3f6bf009` since it has $P = 2^{-21.7}$ for the first phase of the attack.

Table 2. Example of a near-collision of weight 42 for the complete hash function FORK-256. The first block is used to obtain the desired values of chaining registers that enable the attack on the compression function.

M	2d4458a4	57976f57	3e44cfd9	1ab54cb2	7ec11870	173f6573	6141c261	7db20d3e
	2feeb74d	5fac87a6	61a73fa1	3454b23d	451d389b	78f061ec	7c32fb06	57ef1928
	79dcd071	39dc97f0	3a1bff42	031d364c	fef000e6	40873ef5	d0741256	649430cf
	97ef5538	3eab6a7e	b4f9cf72	9eba8257	<u>4e84d457</u>	5a6c49b6	ad1d9711	0f69afa2
M'	2d4458a4	57976f57	3e44cfd9	1ab54cb2	7ec11870	173f6573	6141c261	7db20d3e
	2feeb74d	5fac87a6	61a73fa1	3454b23d	451d389b	78f061ec	7c32fb06	57ef1928
	79dcd071	39dc97f0	3a1bff42	031d364c	fef000e6	40873ef5	d0741256	649430cf
	97ef5538	3eab6a7e	b4f9cf72	9eba8257	<u>8df0c460</u>	5a6c49b6	ad1d9711	0f69afa2
diff	00000000	83480012	32b4070c	681a1279	648600ad	00000000	00000000	00000000

5 Conclusions

In this paper we presented an attack that can find near-collisions and even collisions for the full hash function of FORK-256. We improved on previous results that used large memory and were too inefficient to implement in practice. This in a sense completes the attack and adds another result relevant to the analysis of FORK-256 and possibly also similar designs.

We remark that the authors of FORK-256 recently proposed a patched version of their function [4], largely due to [8]. Because of a change in functions f and g and a modified structure of the step transformation, the new FORK does not allow for finding micro-collisions. Despite this, Saarinen found an attack on the new FORK [10] faster than birthday paradox but requiring large memory. It would be interesting to see if either the time or memory requirements can be improved.

Acknowledgements. The authors were supported by Australian Research Council grant DP0663452.

References

1. Anderson, R., Biham, E.: Tiger: A fast new hash function. In: Gollmann, D. (ed.) Fast Software Encryption. LNCS, vol. 1039, pp. 121–144. Springer, Heidelberg (1996)
2. Bernstein, D.J.: What output size resists collisions in a xor of independent expansions? In: ECRYPT Hash Workshop (May 2007)
3. Contini, S., Matusiewicz, K., Pieprzyk, J.: Cryptanalysis of FORK-256. web page (2007), `http://www.ics.mq.edu.au/~kmatus/FORK/`
4. Hong, D., Chang, D., Sung, J., Lee, S., Hong, S., Lee, J., Moon, D., Chee, S.: New FORK-256. Cryptology ePrint Archive, Report, 2007/185 (2007), `http://eprint.iacr.org/2007/185`
5. 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)
6. Hong, D., Sung, J., Lee, S., Moon, D., Chee, S.: A new dedicated 256-bit hash function. In: Robshaw, M. (ed.) FSE 2006. LNCS, vol. 4047, Springer, Heidelberg (2006)
7. Matusiewicz, K., Contini, S., Pieprzyk, J.: Weaknesses of the FORK-256 compression function. IACR Cryptology e-print Archive, Report 2006/317 (2006), `http://eprint.iacr.org/2006/317`
8. Matusiewicz, K., Peyrin, T., Billet, O., Contini, S., Pieprzyk, J.: Cryptanalysis of FORK-256. In: FSE 2007. LNCS, vol. 4593, pp. 19–38. Springer, Heidelberg (2007)
9. 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)
10. Saarinen, M.-J.O.: A meet-in-the-middle collision attack against the new FORK-256. In: Proceedings of Indocrypt 2007. LNCS, Springer, Heidelberg (2007)
11. Schneier, B., Kesley, J.: Unbalanced Feistel networks and block cipher design. In: Gollmann, D. (ed.) FSE 1996. LNCS, vol. 1039, pp. 121–144. Springer, Heidelberg (1996)

A Constants

Table 3. Constants used to initialise chaining variables of FORK-256

A_0	B_0	C_0	D_0	E_0	F_0	G_0	H_0
6a09e667	bb67ae85	3c6ef372x	a54ff53a	510e527f	9b05688c	1f83d9ab	5be0cd19

Table 4. Step constants $\delta_0, \ldots, \delta_{15}$ used in FORK-256

δ	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