A Meet-in-the-Middle Collision Attack Against the New FORK-256

Markku-Juhani O. Saarinen

Information Security Group
Royal Holloway, University of London
Egham, Surrey TW20 0EX, UK.
m.saarinen@rhul.ac.uk

Abstract. We show that a $2^{112.9}$ collision attack exists against the FORK-256 Hash Function. The attack is surprisingly simple compared to existing published FORK-256 cryptanalysis work, yet is the best known result against the new, tweaked version of the hash. The attack is based on "splitting" the message schedule and compression function into two halves in a meet-in-the-middle attack. This in turn reduces the space of possible hash function results, which leads to significantly faster collision search. The attack strategy is also applicable to the original version of FORK-256 published in FSE 2006.

Keywords: FORK-256, Hash Function Cryptanalysis, Meet-in-the-middle Attack.

1 Introduction

FORK-256 is a dedicated hash function that produces a 256-bit hash from a message of arbitrary size. The original version of FORK-256 was presented in the first NIST hash workshop and at FSE 2006 [1]. Several attacks have been outlined against this original version, namely:

– Matusiewicz, Contini, and Pieprzyk attacked FORK-256 by using the fact that the functions f and g in the step function were not bijective in the original version. They used microcollisions to find collisions of 2-branch FORK-256 and collisions of full FORK-256 with complexity of $2^{126.6}$ in [3].
– Independently, Mendel, Lano, and Preneel published the collision-finding attack on 2-branch FORK-256 using microcollisions and raised possibility of its expansion [5].
– At FSE 2007 [4], Matusiewicz et al. published the result of [3] and another attack which finds a collision with complexity of 2^{108} and memory of 2^{64}.

In response to these attacks the authors of FORK-256 have recently proposed a new, tweaked version of FORK-256 [2], which is supposedly resistant to all before-mentioned attacks. We will present a simple attack which is the best currently known against the new version of FORK-256, and also applicable to the previous version.

2 Description of New FORK-256

New FORK-256 (hereafter FORK-256) is a Merkle-Damgård hash with a 256-bit (8-word) internal state and a 512-bit (16-word) message block. Padding and chaining details are similar to those of the SHA and the MD families of hash functions.

FORK-256 is entirely built on shift, exlusive-or, and addition operations on 32-bit words. In this paper we use the following notation for these operations:

$x \oplus y$	Bitwise exclusive-or between x and y.
$x \boxplus y$	Equal to $(x + y) \mod 2^{32}$.
$x \boxminus y$	Equal to $(x - y) \mod 2^{32}$.
$x \lll y$	Circular left shift of 32-bit word x by y bits.

The compression function of FORK-256 consists of four independent "branches". Each one these branches takes in the 256-bit (8-word) chaining value and a 512-bit (16-word) message block to produce a 256-bit result. These four branch results are combined with the chaining value to produce the final compression function result. Figure 1 illustrates the branch structure.

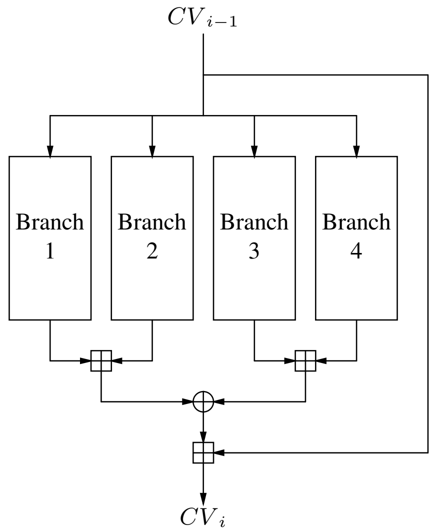

Fig. 1. Overall structure of four branches of FORK-256. Note that the lines are 256 bits wide; the addition symbols represent eight 32-bit modular additions in parallel.

The four branches are structurally equivalent, but differ in scheduling of the message words and round constants. Each branch is computed in eight steps, $0 \leq s \leq 7$. Each step utilizes two message words and two round constants.

The scheduling of the message block words $M[0 \ldots 15]$ in each branch is given in Table 1. Round constants $\delta[0 \ldots 15]$ are given in Table 2 and their schedule in Table 3. The original description uses auxiliary tables σ and ρ; for convenience we use a ("left word"), b ("right word"), α ("left constant"), and β ("right constant") in this description as follows:

$$
\begin{aligned}
a_j^{(s)} &= M[\sigma_j(2s)] \\
b_j^{(s)} &= M[\sigma_j(2s+1)] \\
\alpha_j^{(s)} &= \delta[\rho_j(2s)] \\
\beta_j^{(s)} &= \delta[\rho_j(2s+1)]
\end{aligned}
$$

FORK-256 uses two 32-bit Boolean functions f and g, which were redefined for the New FORK-256 to avoid microcollisions.

$$
\begin{aligned}
f(x) &= x \oplus (x \lll 15) \oplus (x \lll 27) \\
g(x) &= x \oplus ((x \lll 7) \boxplus (x \lll 25)).
\end{aligned}
$$

Following the convention of the FORK-256 specification, let $CV_i[0..7]$ be the result of the compression function iteration i and $CV_0[0..7]$ the Initialization Vector, given in Table 4.

Each branch j processes eight input words $R_j^{(0)}[t] = CV_i[t]$ to eight output words $R_j^{(8)}[t]$, $0 \le t \le 7$. Figure 2 illustrates the step function. For $0 \le s \le 7$:

$$
\begin{aligned}
t_1 &= f(R_j^{(s)}[0] \boxplus a_j^{(s)}) \\
t_2 &= g(R_j^{(s)}[0] \boxplus a_j^{(s)} \boxplus \alpha_j^{(s)}) \\
t_3 &= g(R_j^{(s)}[4] \boxplus b_j^{(s)}) \\
t_4 &= f(R_j^{(s)}[4] \boxplus b_j^{(s)} \boxplus \beta_j^{(s)}) \\
R_j^{(s+1)}[0] &= R_j^{(s)}[7] \oplus (t_4 \lll 8) \\
R_j^{(s+1)}[1] &= R_j^{(s)}[0] \boxplus a_j^{(s)} \boxplus \alpha_j^{(s)} \\
R_j^{(s+1)}[2] &= R_j^{(s)}[1] \boxplus t_1 \\
R_j^{(s+1)}[3] &= (R_j^{(s)}[2] \boxplus (t_1 \lll 13)) \oplus t_2 \\
R_j^{(s+1)}[4] &= R_j^{(s)}[3] \oplus (t_2 \lll 17) \\
R_j^{(s+1)}[5] &= R_j^{(s)}[4] \boxplus b_j^{(s)} \boxplus \beta_j^{(s)} \\
R_j^{(s+1)}[6] &= R_j^{(s)}[5] \boxplus t_3 \\
R_j^{(s+1)}[7] &= (R_j^{(s)}[6] \boxplus (t_3 \lll 3)) \oplus t_4
\end{aligned}
$$

The final result of the compression function for each word $0 \le t \le 7$ is

$$
CV_{i+1}[t] = CV_i[t] \boxplus ((R_1^{(8)} t \boxplus R_2^{(8)}[t]) \oplus (R_3^{(8)}[t] \boxplus R_4^{(8)}[t])).
$$

If i is the final iteration, CV_{i+1} is the final hash value.

3 Observations

Each branch of the compression function uses each message word $M[0\dots15]$ exactly once. Due to diffusion properties of the step function, message words that are scheduled for the last steps do not affect all output words.

Consider the sixth output word of each branch, $R_j^{(8)}[5]$. The last step is defined as:

$$R_j^{(8)}[5] = R_j^{(7)}[4] \boxplus b_j^{(7)} \boxplus \beta_j^{(7)}.$$

Furthermore we "open up" $R_j^{(7)}[4]$ in the previous step:

$$R_j^{(7)}[4] = R_j^{(6)}[3] \oplus (g(R_j^{(6)}[0] \boxplus a_j^{(6)} \boxplus \beta_j^{(6)}) \lll 17).$$

Ignoring the round constants $\alpha_j^{(s)}$ and $\beta_j^{(s)}$, we can observe that the only message words in steps 6 and 7 affecting $R_j^{(8)}[5]$ are $a_j^{(6)}$ and $b_j^{(7)}$, the latter having a linear effect. Constants $b_j^{(6)}$ and $a_j^{(7)}$ have no effect in the computation of this word.

By thus inspecting the step function and the message word schedule in Table 1, it is easy to verify that $R_j[5]$ satisfies the following properties:

Branch 1: $R_1^{(8)}[5]$ is independent of $M[14] = a_1^{(7)}$.
Branch 2: $R_2^{(8)}[5]$ is linearly dependent on $M[1] = b_2^{(7)}$.
Branch 3: $R_3^{(8)}[5]$ is independent of $M[1] = a_3^{(7)}$.
Branch 4: $R_4^{(8)}[5]$ is independent of $M[14] = b_4^{(6)}$.

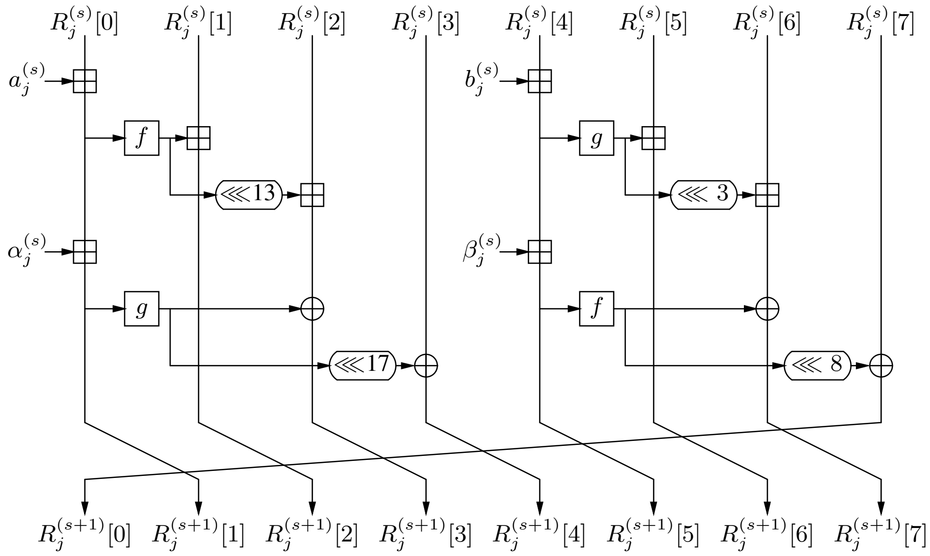

Fig. 2. The new FORK-256 step iteration.

We shall use these simple observations to construct an attack against FORK-256. We note that due to the fact the message word schedule is shared between the old and new versions of FORK-256, the same four observations – and the same general attack – apply to both versions, although there are important technical differences between the old and the new version. The complexity of the attack is the same for both.

4 A Collision Attack

The main strategy of the attack is to use a fast method for finding messages that hash into a significantly smaller subset of possible hash values. We do this by forcing the sixth word of the compression function to remain constant over the hash function iteration, $CV_1[5] = CV_0[5]$, thereby generating hashes in a subset of size 2^{224}. Assuming uniform distribution, a full collision can be expected after $\sqrt{\frac{\pi}{2}} \times 2^{\frac{224}{2}} \approx 2^{112.3}$ hashes in the small subset have been found.

The value of $CV_1[5]$ is combined from the four branches and the initialization vector as follows:

$$CV_1[5] = CV_0[5] \boxplus ((R_1^{(8)}[5] \boxplus R_2^{(8)}[5]) \oplus (R_3^{(8)}[5] \boxplus R_4^{(8)}[5])).$$

By substituting $CV_1[5] = CV_0[5]$ and regrouping branches 2 and 3 on the left side and branches 1 and 4 on the right side, we obtain the following necessary and sufficient condition for $CV_1[5] = CV_0[5]$:

$$R_2^{(8)}[5] \boxminus R_3^{(8)}[5] = R_1^{(8)}[5] \boxminus R_4^{(8)}[5].$$

Our attack is based on choosing two message words $M[1]$ and $M[14]$ in a specific way to satisfy $CV_1[5] = CV_0[5]$, which is possible due to the observations given in the previous section. The values of the fourteen other message words are arbitrary and can be chosen at random (as long as they remain constant through the two phases of the attack). The two phases can be repeated any number of times to produce sufficient amount of hashes in the subset.

4.1 First Phase

Set $M[1] = 0$ and loop over $M[14] = 0, 1, 2, \cdots, 2^{32} - 1$. Compute branches 2 and 3 for each $M[14]$ to obtain $x = R_2^{(8)}[5] \boxminus R_3^{(8)}[5]$. Place x and $M[14]$ into a look-up table so that the value of $M[14]$ can be immediately retrieved based on the corresponding x value (i.e. $M[14]$ is indexed by x).

Note that since the mapping from $M[14]$ to x is not surjective, about $1/e \approx 36.8\%$ of the values of x will never occur (when the mapping is modeled as random). On the other hand, many x can be obtained with more than one value of $M[14]$. Using a straightforward lookup cannot handle the latter situation, but simple data structures with negligible expansion exist that can be used for these cases. The table does not need to be larger than 16 gigabytes (32 bits $\times 2^{32}$ entries).

4.2 Second Phase

Loop over the 2^{32} values of $M[1]$. Compute branches 1 and 4 for each $M[1]$ to obtain $y = R_1^{(8)}[5] \boxminus R_4^{(8)}[5] \boxplus M[1]$. The $M[1]$ term is included due to the linear dependence of $R_2^{(8)}[5]$ on it (this is also why $M[1]$ is set to zero in the first phase).

In each step, perform a look-up. If a match or matches $x = y$ are found, the necessary and sufficient condition is satisfied and we have found a message (or rather, a pair of $M[1]$ and $M[14]$ values) that produces one or more hashes that satisfy $CV_1[5] = CV_0[5]$.

4.3 Runtime Analysis

Each loop step in the second phase produces one match in the lookup table on average. This is due to the fact that even though the mapping is not surjective, there is a total of 2^{32} $M[14]$ entries in the table. Hence approximately 2^{32} hashes with the property are produced in the second phase.

Since computation of only two branches out of four are needed, the computational effort in the first and second phases is roughly equivalent to 2^{31} full hash computations each, or 2^{32} total. If the full 8 words in phase 1 are not stored, branches 2 and 3 need to be computed again to reproduce a full hash, bringing the total number to $3 * 2^{31}$. The average cost of producing a hash in the 2^{224} subset therefore is $\frac{3}{2}$ hash function invocations.

Unfortunately we have been unable to come up with a method of utilizing "memoryless" random-walk collision search methods such as those discussed in [6]. This is due to the fact that the algorithm outlined above only works in "batches" of 2^{32} to obtain a favorable average cost for each hash with the desired property $CV_1[5] = CV_0[5]$. The memory requirement is therefore equivalent to running time requirement, $\frac{3}{2}\sqrt{\frac{\pi}{2}} \times 2^{\frac{224}{2}} = 2^{112.9}$.

5 Further Work

The same observations about the effects of $M[1]$ and $M[14]$ on the final hash can be easily be adopted into a pre-image attack that recovers the values of these two message words with 2^{32} effort, rather than 2^{64} as expected in a brute-force search.

It may be possible to "fix" more than 32 bits by using additional words of keying material besides $M[1]$ and $M[14]$ in the attack. This would naturally lead to a more effective overall collision attack. Terms $M[0]$ and $M[5]$ appear to be good candidates as they are only used in steps 5 and 6 of branches 2 and 3, respectively, and are therefore not fully diffused at the end of step 7.

6 Conclusion

We have presented a $2^{112.9}$ collision attack against the new, improved version of the hash function FORK-256. This represents a speed improvement of factor $2^{15.4}$ over a straightforward collision search. The attack strategy is surprisingly simple, and can also be applied against the original version of FORK-256 in slightly modified form.

7 Acknowledgements

The author would thank Keith Martin and the INDOCRYPT Program Committee members for essential quality control and helpful comments. Financial support for this work was provided by Nixu Ltd. and Academy of Finland.

References

1. D. Hong, D. Chang, J. Sung, S. Lee, S. Hong, J. Lee, D. Moon, and S. Chee. "A New Dedicated 256-Bit Hash Function: FORK-256." FSE 2006, LNCS 4047, Springer-Verlag, pp. 195 – 209, 2006.
2. D. Hong, D. Chang, J. Sung, S. Lee, S. Hong, J. Lee, D. Moon, and S. Chee. "New FORK-256." Cryptology ePrint Archive 2007/185, Jul., 2007.
3. K. Matusiewicz, S. Contini, and J. Pieprzyk. "Weaknesses of the FORK-256 Compression Function." Cryptology ePrint Archive 2006/317 (Second version), Nov., 2006.
4. K. Matusiewicz, T. Peyrin, O. Billet, S. Contini, and J. Pieprzyk. "Cryptanalysis of FORK-256." Preproceeding of FSE 2007, 2007.
5. F. Mendel, J. Lano, and B. Preneel. "Cryptanalysis of Reduced Variants of the FORK-256 Hash Function." CT-RSA, LNCS 4377, Springer-Verlag, pp. 85 – 100, 2007.
6. P. van Oorschot and M. Wiener. "Parallel collision search with cryptanalytic applications." Journal of Cryptology, 12 (1999), pp. 1 – 28, 1999.

Step	Branch 1		Branch 2		Branch 3		Branch 4	
s	$a_1^{(s)}$	$b_1^{(s)}$	$a_2^{(s)}$	$b_2^{(s)}$	$a_3^{(s)}$	$b_3^{(s)}$	$a_4^{(s)}$	$b_4^{(s)}$
0	$M[0]$	$M[1]$	$M[14]$	$M[15]$	$M[7]$	$M[6]$	$M[5]$	$M[12]$
1	$M[2]$	$M[3]$	$M[11]$	$M[9]$	$M[10]$	$M[14]$	$M[1]$	$M[8]$
2	$M[4]$	$M[5]$	$M[8]$	$M[10]$	$M[13]$	$M[2]$	$M[15]$	$M[0]$
3	$M[6]$	$M[7]$	$M[3]$	$M[4]$	$M[9]$	$M[12]$	$M[13]$	$M[11]$
4	$M[8]$	$M[9]$	$M[2]$	$M[13]$	$M[11]$	$M[4]$	$M[3]$	$M[10]$
5	$M[10]$	$M[11]$	$M[0]$	$M[5]$	$M[15]$	$M[8]$	$M[9]$	$M[2]$
6	$M[12]$	$M[13]$	$M[6]$	$M[7]$	$M[5]$	$M[0]$	$M[7]$	$M[14]$
7	$M[14]$	$M[15]$	$M[12]$	$M[1]$	$M[1]$	$M[3]$	$M[4]$	$M[6]$

Table 1. Message word schedule for FORK-256. It is easy to observe that in branch 2 and branch 3, $M[1]$ only affects the result in the last step. $M[14]$ is used in the last and next-to-last steps in branches 1 and 4, correspondingly. These observations are used in the attack.

$\delta[0]$ = `0x428a2f98`	$\delta[1]$ = `0x71374491`
$\delta[2]$ = `0xb5c0fbcf`	$\delta[3]$ = `0xe9b5dba5`
$\delta[4]$ = `0x3956c25b`	$\delta[5]$ = `0x59f111f1`
$\delta[6]$ = `0x923f82a4`	$\delta[7]$ = `0xab1c5ed5`
$\delta[8]$ = `0xd807aa98`	$\delta[9]$ = `0x12835b01`
$\delta[10]$ = `0x243185be`	$\delta[11]$ = `0x550c7dc3`
$\delta[12]$ = `0x72be5d74`	$\delta[13]$ = `0x80deb1fe`
$\delta[14]$ = `0x9bdc06a7`	$\delta[15]$ = `0xc19bf174`

Table 2. Round constants.

Step	Branch 1		Branch 2		Branch 3		Branch 4	
s	$\alpha_1^{(s)}$	$\beta_1^{(s)}$	$\alpha_2^{(s)}$	$\beta_2^{(s)}$	$\alpha_3^{(s)}$	$\beta_3^{(s)}$	$\alpha_4^{(s)}$	$\beta_4^{(s)}$
0	$\delta[0]$	$\delta[1]$	$\delta[15]$	$\delta[14]$	$\delta[1]$	$\delta[0]$	$\delta[14]$	$\delta[15]$
1	$\delta[2]$	$\delta[3]$	$\delta[13]$	$\delta[12]$	$\delta[3]$	$\delta[2]$	$\delta[12]$	$\delta[13]$
2	$\delta[4]$	$\delta[5]$	$\delta[11]$	$\delta[10]$	$\delta[5]$	$\delta[4]$	$\delta[10]$	$\delta[11]$
3	$\delta[6]$	$\delta[7]$	$\delta[9]$	$\delta[8]$	$\delta[7]$	$\delta[6]$	$\delta[8]$	$\delta[9]$
4	$\delta[8]$	$\delta[9]$	$\delta[7]$	$\delta[6]$	$\delta[9]$	$\delta[8]$	$\delta[6]$	$\delta[7]$
5	$\delta[10]$	$\delta[11]$	$\delta[5]$	$\delta[4]$	$\delta[11]$	$\delta[10]$	$\delta[4]$	$\delta[5]$
6	$\delta[12]$	$\delta[13]$	$\delta[3]$	$\delta[2]$	$\delta[13]$	$\delta[12]$	$\delta[2]$	$\delta[3]$
7	$\delta[14]$	$\delta[15]$	$\delta[1]$	$\delta[0]$	$\delta[15]$	$\delta[14]$	$\delta[0]$	$\delta[1]$

Table 3. Round constant schedule.

$CV_0[0]$ = `0x6a09e667`	$CV_0[1]$ = `0xbb67ae85`
$CV_0[2]$ = `0x3c6ef372`	$CV_0[3]$ = `0xa54ff53a`
$CV_0[4]$ = `0x510e527f`	$CV_0[5]$ = `0x9b05688c`
$CV_0[6]$ = `0x1f83d9ab`	$CV_0[7]$ = `0x5be0cd19`

Table 4. Initialization Vector.