# Weaknesses of the FORK-256 compression function

Krystian Matusiewicz, Scott Contini, and Josef Pieprzyk

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

**Abstract.** This report presents analysis of the compression function of a recently proposed hash function, FORK-256. We exhibit some unexpected differentials existing for the step transformation and show their possible uses in collision-finding attacks on different variants of FORK-256. As a simple application of those observations we present a method of finding chosen IV collisions for a variant of FORK-256 reduced to two branches : either 1 and 2 or 3 and 4. Moreover, we present how those differentials can be used in the full FORK-256 to easily find messages with hashes differing by only a relatively small number of bits. We argue that this method allows for finding collisions in the full function with complexity not exceeding $2^{126.6}$ hash evaluations, better than birthday attack and additionally requiring only a small amount of memory.

Version: 2.1, last modified: November 29, 2006

## 1 Introduction

Most of dedicated hash functions published in the last 15 year follow more or less closely design ideas used by R. Rivest in his functions MD4 [10, 11] and MD5 [12]. Using terminology from [13], 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 [9], HAVAL [18], SHA-1 [7] and also SHA-256 [8]. A very nice feature of all these designs is that they all are very fast in software implementations on modern 32-bit processors and use only a small set of basic instructions executed in constant-time like additions, rotations and Boolean functions.

However, traditional wisdom says that monoculture is dangerous. This proved to be true also in the world of hash functions. Ground-breaking attacks on MD4, MD5 by X. Wang *et al.* [16, 14] were later refined and applied to attack SHA-0 [17] and SHA-1 [15] 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 designed in 1995 hash function Tiger [1] (tailored for 64-bit platforms) and a recently proposed FORK-256 [3] which is the focus of this paper.

We exhibit a flaw in the design of the step transformation of the compression function that allows for a special kind of rather pathological differentials to exist. We analyse those differentials in details in section 4 and derive an efficient necessary and sufficient condition for the existence of those differentials. Effectivness of this test allows us to search for suitable configurations extremely fast.

Then, in section 5 we show how to exploit the existence of such local differentials to construct a high-level differential path for the full function as well as for its various simplified variants.

As a concrete example we show how to easily find collisions for two branches of FORK-256 in section 6. Finally, in section 7 we present a method for finding hashes differing by only a small number of bits and show how this method is applicable to finding collisions for the full compression function of FORK-256 with complexity $2^{126.6}$, faster than by birthday paradox.

*Notation* Throughout the paper we will use the notation presented below. Unless stated otherwise, all words are 32-bit and can be seen as elements of $\mathbb{Z}_{2^{32}}$ or $\mathbb{Z}_2^{32}$.

| | |
|---|---|
| $X + Y$ | integer addition / addition modulo $2^{32}$ (depending on the context), |
| $X - Y$ | integer subtraction / modular subtraction of two words $X, Y$, |
| $X \oplus Y$ | bitwise XOR of two words $X$, $Y$, |
| $ROL^a(X)$ | rotation of bits of the word $X$ by $a$ positions left, |
| $R_i^{(j)}$ | the value of register $R \in A, \ldots, H$ in branch $j = 1, \ldots, 4$ after step $i$. |

## 1.1 A brief description of FORK-256

FORK-256 is a dedicated hash function recently proposed by Hong *et al.* [3, 4]. It is 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. For the complete description we refer interested readers to [3], here we only present an outline necessary to understand main ideas of the rest of this paper.

The compression function consists of four parallel branches $BRANCH_j$, $j = 1, 2, 3, 4$, each one of them using a different permutation of 16 message words $M_i$, $i = 0, \ldots, 15$ and the same set of chaining variables $CV = (A, B, C, D, E, F, G, H)$. The compression function updates the set of chaining variables according to the formula

$$\begin{aligned} CV_{i+1} = CV_i + \{&[BRANCH_1(CV_i, M) + BRANCH_2(CV_i, M)] \oplus \\ &[BRANCH_3(CV_i, M) + BRANCH_4(CV_i, M)]\} \quad , \end{aligned}$$

where modular and XOR additions are performed word-wise. This construction can be seen as a further extension of the design principle of two parallel lines used in RIPEMD [9].

Each branch function $BRANCH_j,\ j = 1, 2, 3, 4$ consists of eight steps. In each step $k = 1, \ldots, 8$ branch function updates its own copy of eight chaining variables according to the following formulae

$$\begin{aligned}
A_k^{(j)} &:= H_{k-1}^{(j)} + ROL^{21}(g(E_{k-1}^{(j)} + M_{\sigma_j(2k-1)}) \oplus ROL^{17}(f(E_{k-1}^{(j)} + M_{\sigma_j(2k-1)} + \delta_{\pi_j(2k-1)})),\\
B_k^{(j)} &:= A_{k-1}^{(j)} + M_{\sigma_j(2k-2)} + \delta_{\pi_j(2k-2)},\\
C_k^{(j)} &:= B_{k-1}^{(j)} + f(A_{k-1}^{(j)} + M_{\sigma_j(2k-2)}) \oplus g(A_{k-1}^{(j)} + M_{\sigma_j(2k-2)} + \delta_{\pi_j(2k-2)}),\\
D_k^{(j)} &:= C_{k-1}^{(j)} + ROL^{5}(f(A_{k-1}^{(j)} + M_{\sigma_j(2k-2)})) \oplus ROL^{9}(g(A_{k-1}^{(j)} + M_{\sigma_j(2k-2)} + \delta_{\pi_j(2k-2)})),\\
E_k^{(j)} &:= D_{k-1}^{(j)} + ROL^{17}(f(A_{k-1}^{(j)} + M_{\sigma_j(2k-2)})) \oplus ROL^{21}(g(A_{k-1}^{(j)} + M_{\sigma_j(2k-2)} + \delta_{\pi_j(2k-2)})),\\
F_k^{(j)} &:= E_{k-1}^{(j)} + M_{\sigma_j(2k-1)} + \delta_{\pi_j(2k-1)},\\
G_k^{(j)} &:= F_{k-1}^{(j)} + g(E_{k-1}^{(j)} + M_{\sigma_j(2k-1)}) \oplus f(E_{k-1}^{(j)} + M_{\sigma_j(2k-1)} + \delta_{\pi_j(2k-1)}),\\
H_k^{(j)} &:= G_{k-1}^{(j)} + ROL^{9}(g(E_{k-1}^{(j)} + M_{\sigma_j(2k-1)}) \oplus ROL^{5}(f(E_{k-1}^{(j)} + M_{\sigma_j(2k-1)} + \delta_{\pi_j(2k-1)})),
\end{aligned}$$

where $R_i^{(j)}$ denotes the value of the register $R$ in $j$-th branch after step $i$ and all $A_0^{(j)}, \ldots, H_0^{(j)}$ are initialized with corresponding values of eight chaining variables. Functions $f$ and $g$ are defined as

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

Constants $\delta_0, \ldots, \delta_{15}$ 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 1.

**Table 1.** Constants $\delta_0, \ldots, \delta_{15}$ used in FORK-256

| $\delta$ | 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 |

Finally, permutations $\sigma_j$ of message words and permutations $\pi_j$ of constants are shown in Table 2.

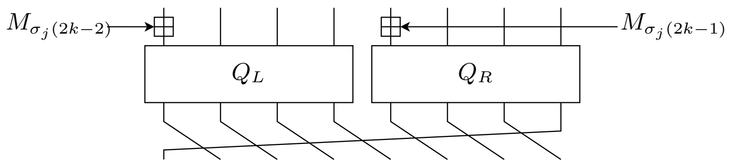
**Table 2.** Message and constant permutations used in four branches of FORK-256

| $j$ | 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 |

## 2 Analysis of step transformation of FORK-256

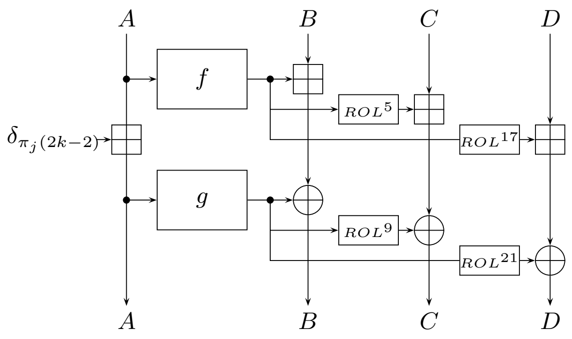
The step transformation described in the previous section can be logically split into three parts: addition of message words, two parallel mixing structures $Q_L$ and $Q_R$ and a final rotation of registers. This view is presented in Fig. 1. The key role is played by the two transformations of four words, $Q_L$ and $Q_R$ as they are the main source of diffusion in the compression function. It is clear that if we can find interesting differential characteristics for $Q_L$ and $Q_R$, we should be able to extend them to the whole branch and maybe also the whole function.

**Fig. 1.** A high-level structure of step transformation of FORK-256



Let us focus on $Q_L$, presented in Fig. 2, as $Q_R$ is very similar to $Q_L$ ($f$ and $g$ are swapped and rotation amounts are different) and the properties we are going to discover work for both of them.

**Fig. 2.** $Q_L$-structure of step transformation in FORK-256



Characteristics of the form $(0, \Delta B, \Delta C, \Delta D) \rightarrow (0, \Delta B, \Delta C, \Delta D)$ are not that difficult to get since in each step the difference in registers $B$, $C$, $D$ are modified by only one modular addition and one XOR operation. Whether we consider modular or XOR differences, there is only one incompatible operation to deal with.

We can combine such characteristics to get a straightforward differential for up to three steps for each branch.

The difficult part is characteristics of the form $(\Delta A, 0, 0, 0) \to (\Delta A, 0, 0, 0)$. As far as we could see, there are two ways of finding them. The first method of finding those desired characteristics is based on the fact that both $f$ and $g$ are not bijective so we can hope that we can find such inputs $x, x'$ that $f(x) = f(x')$ and $g(x + \delta) = g(x' + \delta)$. The second one is aimed at getting zero differences in registers $B$, $C$, $D$ in spite of non-zero differences at the outputs of $f$ and $g$. In next sections we describe both of them in detail.

## 3 Simultaneous collisions for $f$ and $g$

For given value $\delta$, we would like to find all $x$ and $x'$ such that $f(x) = f(x')$ and $g(x + \delta) = g(x' + \delta)$. A naive search would require computations of order $2^{64}$, which is well beyond our computing resources. A less naive method trades time for memory. Below we describe this tradeoff in a way that involves order $2^{32}$ computations and $2^{32}$ memory for the particular functions $f$ and $g$ used in FORK-256. Again, we focus on $Q_L$, i.e. $f$ is applied before $g$.

**Step 1:** We determine which inputs $x$ have more than one preimage. This is done by initializing an array of $2^{32}$ entries to zero, and then incrementing entries indexed by $f(x)$ in the array for all $2^{32}$ inputs $x$. The entries with their values at least 2 are output. There are about $2^{30}$ of these. In fact, this step is not necessary for our algorithm, but it may be helpful in practice since it reduces memory requirements for the next step.

**Step 2:** Read in the values of $f(x)$ from Step 1, i.e. the values that have more than one preimage. Then, for each input value, build a linked list of all preimages of that value. This is done in a similar way to Step 1: compute all $2^{32}$ values of $f(x)$, and for each value that matches one of the inputs from Step 1 (this can be checked quickly with a hash table), add it to the corresponding linked list. The longest linked list for $f$ has 12 preimages.

**Step 3:** Process the linked lists from Step 2. For each linked list, consider the set of values $x_1, \dots, x_k$ that map to the same image. See if there is any $x_i$ and $x_j$ in that list such that $g(x_i + \delta) = g(x_j + \delta)$, and if so, output the pair as a solution of simultaneous collisions of $f$ and $g$.

The running time of Step 3 depends upon the number of combinations of pairs of preimages that map to the same image. According to our computations, this is $2134351185 < 2^{31}$.

There are many potential tricks to reduce the search space and/or memory requirements further, but the above algorithm was sufficient for us to determine the following solutions:

$$x = \texttt{4b4d2a05},\ x' = \texttt{6ff2f3e9},\ \text{for}\ \delta_1 = \texttt{71374491},$$
$$x = \texttt{06def69a},\ x' = \texttt{aeb691e5},\ \text{for}\ \delta_2 = \texttt{b5c0fbcf},$$
$$x = \texttt{27a61343},\ x' = \texttt{67eac4d8},\ \text{for}\ \delta_3 = \texttt{e9b5dba5},$$
$$x = \texttt{04549cdc},\ x' = \texttt{20d331a5},\ \text{for}\ \delta_7 = \texttt{ab1c5ed5},$$

for $Q_L$ and

$$x = \texttt{445c5563},\ x' = \texttt{d73bc777},\ \text{ for } \delta_{10} = \texttt{243185be},$$
$$x = \texttt{be452586},\ x' = \texttt{edfd4d5b},\ \text{ for } \delta_{14} = \texttt{9bdc06a7}.$$

for $Q_R$.

## 4 Microcollisions in $Q_L$ and $Q_R$

In this section we concentrate on an alternative way of finding characteristics of the form $(\Delta A, 0, 0, 0) \to (\Delta A, 0, 0, 0)$ 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 $A$ such that 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 a situation is possible if we have three simultaneous *microcollisions* : differences in $g$ cancel out differences from $f$ in all three registers $B$, $C$, $D$ (cf. Fig. 2).

### 4.1 Necessary and sufficient condition for microcollisions

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

$$(y + B) \oplus z = (y' + B) \oplus z' \tag{1}$$

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 $x, x' \in \mathbb{Z}_{2^{32}}$. We will use certain relationships between them in our analysis.

- The first kind of representation useful for us is the usual XOR difference. We will treat it as a vector of 32 digits representing bits of $x \oplus x'$ and denote it $\Delta^{\oplus}(x, x') \in \{0, 1\}^{32}$.
- The second one is a plain integer difference. For two numbers $x$, $x'$, we define the integer difference $\partial x$ simply as the result of the subtraction of two operands, i.e. $\partial x = x - x'$, $-2^{32} < \partial x < 2^{32}$.
- The third kind of representation we will be using is the signed binary representation. It uses three digits, 1, 0, $-1$, and a pair $x$, $x'$ has signed binary representation $\Delta^{\pm}(x, x') = (x_0 - x'_0, x_1 - x'_1, \ldots, x_{31} - x'_{31})$, i.e. the $i$-th component is the result of the subtraction of corresponding bits of $x$ and $x'$ at position $i$.

A simple but important observation is that if a difference has signed representation $(r_0, r_1, \ldots, r_{31})$ than the corresponding XOR difference is $(|r_0|, |r_1|, \ldots, |r_{31}|)$, i.e. 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 $(r_0, \ldots, r_{31})$ 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. To see this better consider a small example. Let us assume words of 4 bits and consider $\Delta^{\pm}(11, 2) = (1, 0, 0, 1)$, $\Delta^{\pm}(14, 5) = (1, 0, 1, -1)$ 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.

After this introductory part we are equipped with the necessary tools and can go back to our initial problem. Rewriting (1) as

$$(y + B) \oplus (y' + B) = z \oplus z' \ . \tag{2}$$

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 all possible signed binary representations that can "fit" into 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 $I = \{k_0, k_1, \ldots, k_m\} \subset \{0, \ldots, 31\}$ (that is determined by non-zero bits of $\Delta^{\oplus}(z, z')$), decide whether it is possible to find a binary signed representation $r = (r_0, \ldots, r_{31})$ corresponding to $\partial y$ such that

$$\partial y = \sum_{i=0}^{m} 2^{k_i} \cdot r_{k_i} \quad \text{where } r_{k_i} \in \{-1, 1\} \ . \tag{3}$$

Substituting $t_i = (r_{k_i} + 1)/2$ we can rewrite the above equation in the equivalent form

$$\partial y + \sum_{i=0}^{m} 2^{k_i} = 2^{k_0+1} t_0 + 2^{k_1+1} t_1 + \cdots + 2^{k_m+1} t_m \ , \tag{4}$$

where $t_i \in \{0, 1\}$. Deciding if there are numbers $t_i$ that satisfy (4) is an instance of the knapsack problem and since it is superincreasing (because weights are powers of two), we can do this very efficiently.

This gives us a computationally efficient necessary condition for microcollision in a line: if $\partial y = y - y'$ cannot be represented as (3), no constant $B$ exist and there is no solution of (1).

Moreover, we can show that this is as well a sufficient condition: if we can find a solution to the problem (3), then there exist a constant $B$ that modifies the signed difference in such a way that it "fits" the prescribed XOR pattern.

Observe that since the solution of the superincreasing knapsack problem (4) is unique, so is the solution of the equivalent problem (3). This means that we know the unique signed representation $\Delta^{\pm}(u, u + \partial y) = (r_0, \ldots, r_{31})$ 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 be arbitrary. 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 shows also that if we can have a microcollision in a line, there are $|\mathcal{B}| = 2^{32 - h_w(z \oplus z')}$ constants that yield the microcollision if the most significant bit of $z \oplus z'$ is zero and $2^{32 - h_w(z \oplus z') + 1}$ if the MSB of $z \oplus z'$ is one. The difference is caused by the fact that if $31 \in I$, we don't need to change $u_{31}$ in a particular way (i.e. either $1 \rightarrow 0$ or $0 \rightarrow 1$), any change is fine since we don't introduce carries anyway.

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

## 4.2 Estimation of probabilities of microcollisions

From a practical point of view, we are interested in the probability that a random pair of values $(A, A')$ may lead to simultaneous microcollisions and what is the overall probability of characteristics of the form $(\Delta A, 0, 0, 0) \rightarrow (\Delta A, 0, 0, 0)$ when we cannot manipulate the values of registers $A$, $B$, $C$, $D$.

We conducted some experiments for $Q_L$ and $Q_R$ with different constants $\delta$. Our results indicate that the probability that a random pair of inputs $(A, A')$ may lead to simultaneous microcollisions in all three lines is around $2^{-23}$ with probability for a single line close to $2^{-13}$.

The probability that random constants $B$, $C$, $D$ adjust the difference in $f(x)$ properly depends on Hamming weights of $\Delta^{\oplus}(z, z')$. One example of such distribution of weights obtained by testing $2^{32}$ random pairs[1] is presented in Table 3.

We can see a clear peak around weights 24–26, so, according to the formula describing the size of the set of constants from the previous subsection, we can expect $2^6 \sim 2^8$ "good" constants in each of the sets $\mathcal{B}$, $\mathcal{C}$, $\mathcal{D}$ and thus the probability that a random constant falls into that set is around $2^{-24} \sim 2^{-26}$. Of course to get a result for all three branches we need to cube that number.

[1] In all experiments we were using Mersenne Twister [5] as the source of pseudorandom numbers

**Table 3.** Distribution of Hamming weights of $\Delta^{\oplus}(g(x+\delta_0), g(x'+\delta_0))$ corresponding to potential simultaneous microcollisions after testing $2^{32}$ random pairs $x, x'$

| $h_w$ | 0 | 1 | ... | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| count | 1 | 0 | ... | 0 | 1 | 6 | 18 | 29 | 59 | 78 | 74 | 90 | 56 | 39 | 14 | 1 | 0 | 0 |

Using the above results, we can try to estimate the probability that a set of three simultaneous microcollisions occurs if we have no control of any values $A$, $A'$, $B$, $C$, $D$. Multiplying $2^{-23}$ by $2^{-72} \sim 2^{-78}$ we get an estimation of $2^{-95} \sim 2^{-101}$. It shows that such differentials are not immediately useful, but if we can force specific values of registers to desired values, they may be used to construct collisions for at least simplified variants of FORK, as presented in next sections.

## 5 Finding high-level differential paths in FORK-256

If we can avoid mixing introduced by the structures $Q_L$ and $Q_R$ (i.e. we know how to get differentials $(\Delta A, 0, 0, 0) \rightarrow (\Delta A, 0, 0, 0)$ and $(\Delta E, 0, 0, 0) \rightarrow (\Delta E, 0, 0, 0)$) and we can assume that differences in the registers $B, C, D$ and $F, G, H$ remain unchanged, 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 are simple linear functions of registers of the initial vector and message words. If we denote $\Delta X_0 + \Delta M_{\sigma_j(a)}$ by [X,a] and $\Delta X_0 + \Delta M_{\sigma_j(a)} + \Delta M_{\sigma_j(b)}$ by [X,a,b], where $\sigma_j$ is the permutation of message words used in branch $j = 1, 2, 3, 4$, we can write this down concisely in a tabular form presented in Table 4.

**Table 4.** If no mixing through $Q_L$ and $Q_R$ occurs, differences in registers are combinations of differences in initial vectors and message words. [X,i] stands for $\Delta X_0 + \Delta M_{\sigma_j(i)}$ and [X,a,b] stands for $\Delta X_0 + \Delta M_{\sigma_j(a)} + \Delta M_{\sigma_j(b)}$

| | registers | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| step | $\Delta A$ | $\Delta B$ | $\Delta C$ | $\Delta D$ | $\Delta E$ | $\Delta F$ | $\Delta G$ | $\Delta H$ |
| 1 | [A,0] | [B] | [C] | [D] | [E,1] | [F] | [G] | [H] |
| 2 | [H,2] | [A,0] | [B] | [C] | [D,3] | [E,1] | [F] | [G] |
| 3 | [G,4] | [H,2] | [A,0] | [B] | [C,5] | [D,3] | [E,1] | [F] |
| 4 | [F,6] | [G,4] | [H,2] | [A,0] | [B,7] | [C,5] | [D,3] | [E,1] |
| 5 | [E,1,8] | [F,6] | [G,4] | [H,2] | [A,0,9] | [B,7] | [C,5] | [D,3] |
| 6 | [D,3,10] | [E,1,8] | [F,6] | [G,4] | [H,2,11] | [A,0,9] | [B,7] | [C,5] |
| 7 | [C,5,12] | [D,3,10] | [E,1,8] | [F,6] | [G,4,13] | [H,2,11] | [A,0,9] | [B,7] |
| 8 | [B,7,14] | [C,5,12] | [D,3,10] | [E,1,8] | [F,6,15] | [G,4,13] | [H,2,11] | [A,0,9] |
| output | [A,0,9] | [B,7,14] | [C,5,12] | [D,3,10] | [E,1,8] | [F,6,15] | [G,4,13] | [H,2,11] |

It is clear that differences in registers at any particular step are combinations of differences introduced in the initial vector $(A_0, \ldots, H_0)$ and differences in message words $M_0, \ldots, M_{15}$.

If we consider the simplest case and assume (very optimistically) that any two differences can cancel each other (this is the case with XOR differences), we are in fact working over $\mathbb{F}_2$ and differences in all registers are $\mathbb{F}_2$-linear combinations of differences $\Delta A_0, \ldots, \Delta H_0$ and $\Delta M_0, \ldots, \Delta M_{15}$ (which are now seen as elements of $\mathbb{F}_2$). Now output differences of the whole compression function (including feed-forward) are also linear combinations of differences from $S = (\Delta A_0 \ldots, \Delta H_0, \Delta M_0, \ldots, \Delta M_{15})$ and we can represent this map as an $\mathbb{F}_2$-linear function, $(\Delta A, \ldots, \Delta H) = L_{out}(S)$. This means we can easily find the set $\mathcal{S}_c$ of all vectors $S = (\Delta A_0 \ldots, \Delta H_0, \Delta M_0, \ldots, \Delta M_{15})$ 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 minimize 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 $\Delta A_0 \ldots, \Delta H_0, \Delta M_0, \ldots, \Delta M_{15}$ 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 differences $S$ that give the smallest one. This straightforward process can be improved. If we denote by $V$ the vector of register states we are interested in, there is a matrix $\Psi$ such that $V = S \cdot \Psi$. The matrix $\Psi$ can be seen as a generator matrix of a linear code over $\mathbb{F}_2$. Minimum words of that code correspond to register states with minimal weight. To find collisions (or other restricted paths), the appropriate generating matrix is $Basis(\ker L_{out}) \cdot \Psi$ (or $Basis(\ker(L)) \cdot \Psi$ where $L$ is the linear map describing those registers we want to be zero). Here $Basis(A)$ denotes the basis matrix of a linear space $A$. Using systems like MAGMA [2], finding minimum words in such codes takes only a fraction of a second.

Our computations show that

- Minimal *collision* path in branches 1-2 uses differences in $M_0$ and $M_9$,
- Minimal *collision* path in branches 3-4 uses differences in $M_{14}$ and $M_{15}$,
- Minimal *collision* path for all four branches requires differences in message words $M_6$ and $M_{12}$,
- Minimal *unrestriced* path for all branches has differences in the message $M_{12}$ only

However, differences in registers other than $A$ and $E$ don't contribute to the complexity of the attack that much. The measure based on the number of differences in registers $A$ and $E$ only corresponds more closely to the number of "difficult" differentials we need to handle that require finding microcollisions. Considering this, we also conducted experiments for different variants of FORK-256 counting only differences in registers $A$ and $E$.

The results are presented in Table 5. The first column specifies whether we are interested in collision, pseudo-collisions (differences also appear in the initial vector) of just a free path – no specific conditions on differences are imposed.

**Table 5.** Minimal numbers $m$ of sets of simultaneous microcollisions in $Q_L$ and $Q_R$ necessary in different attack scenarios on variants of FORK-256

| Scenario | Branches | $m$ | Differences in |
|---|---|---|---|
| Collisions | 1,2 | 2 | $M_0$, $M_9$ |
| Collisions | 3,4 | 2 | $M_{14}$, $M_{15}$ |
| Collisions | 1,3 | 3 | $M_5$ |
| Collisions | 1,4 | 3 | $M_2$ |
| Collisions | 2,3 | 3 | $M_3$ |
| Collisions | 2,4 | 3 | $M_9$ |
| Pseudo-collisions | 1,2,3 | 6 | $B_0$ |
| Pseudo-collisions | 1,2,4 | 6 | $B_0$ |
| Pseudo-collisions | 1,3,4 | 6 | $B_0$ |
| Pseudo-collisions | 2,3,4 | 6 | $B_0$ |
| Collisions | 1,2,3,4 | 12 | $M_6$, $M_{12}$ |
| Free path | 1,2,3,4 | 6 | $M_{12}$ |

The third column gives the minimal number of $Q$–structures that require special differentials and thus also microcollisions in registers $B$,$C$,$D$ or $F$,$G$,$H$. The last column gives an example of message and/or chaining variables differences that induce the high-level path with the given number of sets of microcollisions.

## 5.1 More general variant of path finding

We can generalize this approach even further. Depending on whether we force a microcollision 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

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

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 microcollisions to happen. In fact, results presented in Table 5 correspond to a special case when all coefficients $q$ are equal to zero. Relaxing this condition and allowing for microcollisions 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 strucutre.

Results of our search for such paths are summarized in Table 6. They show that by introducing such an extended model of $Q$-structures we can significantly decrease the number of necessary microcollisions. Particuraly interesting is the result showing that, under favourable conditions, collisions can be achieved by using a single difference in $M_{12}$ with 6 microcollision places in the path. In section 7 we show how to use this situation to generate near-collisions and theoretically also collisions.

**Table 6.** Minimal numbers $m$ of $Q$-structures with microcollisions 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 od branch 4.

| Scenario | Branches | $m$ | Differences in | active $Q$-structures |
|---|---|---|---|---|
| Pseudocollisions | 1,2,3,4 | 5 | $IV[7], M_2, M_{11}$ | 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 |
| Pseudocollisions | 1,2,3 | 2 | $IV[1], M_{12}$ | 8:100, 24:0 |
|  | 1,2,4 | 3 | $IV[7], M_{11}$ | 3:000, 51:010, 60:000 |
|  | 1,3,4 | 3 | $IV[7], M_2$ | 35:001, 44:000, 51:000 |
|  | 2,3,4 | 3 | $IV[3], M_9$ | 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 |

## 6 Collisions for two branches of FORK

We can use the minimal path for branches 1&2 to get collisions for these two branches of FORK-256. The idea is to find two related simultaneous microcollisions, the first one of type $f$ - $\delta_0$ - $g$ ($f$ is followed by $\delta_0$ and then by $g$) to be used in the left part of the first step of branch 1 and the other one of type $g$ - $\delta_{12}$ - $f$ to be used in step 2 of branch 2.

If we can find a pair of values $(x, x')$ that yields $f$ - $\delta_0$ - $g$ microcollisions and a pair $(y, y')$ that yields $g$ - $\delta_{12}$ - $f$ microcollisions such that the values satisfy the condition $x - x' = y' - y$, we can construct a collision for branches 1&2 by preserving differences $\partial x = x - x'$ in steps 2, 3, 4 of branch 1 and $\partial y = y - y'$ in steps 3, 4, 5 of branch 2.

The algorithm works as follows:

1. find a pair of values $x, x'$ that produce $f$ - $\delta_0$ - $g$ simultaneous microcollisions and determine the three compatible constants $\rho_1, \rho_2, \rho_3$, (this step requires around $2^{23}$ tests of random pairs $x, x'$)

2. for the fixed difference $\partial x = x - x'$ test pairs of the form $y, y' = y + \partial x$ until a simultaneous microcollision of type $g$ - $\delta_{12}$ - $f$ is found. Determine compatible constants $\tau_1, \tau_2, \tau_3$. (Again, experiments suggest that the complexity of this step is $2^{23}$ tests)
3. set $IV[1] := \rho_1$, $IV[2] := \rho_2$, $IV[3] := \rho_3$,
4. compute $M_0 := x - IV[0]$, $M'_0 := x' - IV[0]$,
5. set both $M_{15}$ and $M'_{15}$ to $\tau_1 - IV[4] - \delta_{14}$,
6. compute initial values $IV[5]$ and $IV[6]$ as follows

$$IV[5] := (\tau_2 \oplus f(IV[4] + M_{15} + \delta_{14})) - g(IV[4] + M_{15}),$$
$$IV[6] := (\tau_3 \oplus ROL^5(f(IV[4] + M_{15} + \delta_{14}))) - ROL^9(g(IV[4] + M_{15}))$$

7. compute the values $M_9 := y - E_1^{(2)}$ and $M'_9 := y' - E_1^{(2)}$, where

$$E_1^{(2)} = ((IV[3] + ROL^{17}(f(IV[0] + M_{14}))) \oplus ROL^{21}(g(IV[0] + M_{14} + \delta_{15}))),$$

   is the value of register $E$ after step 1 in branch 2.
8. preserve the difference $\partial x$ by forcing the value of $g$ to zero in steps 2, 3, 4 (XOR-ing with zero doesn't change the modular difference)
   – set $M'_2 := M_2 := -A_1^{(1)} - \delta 2$,
   – set $M'_4 := M_4 := -A_2^{(1)} - \delta 4$,
   – set $M'_6 := M_6 := -A_3^{(1)} - \delta 6$,
9. similarly, preserve the difference $\partial y$ by forcing the value of $f$ to zero in steps 3, 4, 5 of branch 2
   – set $M'_{10} := M_{10} := -E_2^{(2)} - \delta_{10}$,
   ⋄ in step 3 we cannot modify the value of $M_4$ as it is already fixed by correction done in branch 1. However, we can modify freely the value of $M_8$ (and $M'_8$) which indirectly influences the value of $E_3^{(2)}$ we need to adjust. We do this until the difference in $H_4^{(2)}$ is equal to the difference at the beginning of the step, i.e. in $G_3^{(2)}$. If we exhaust all possible values of $M_8$, we can modify the value of $M_{11}$ and go to step 9 or pick another constant $\rho_1$ and start over from step 3.
   – set $M'_{13} := M_{13} := -E_4^{(2)} - \delta_6$,

The complexity of the attack on branches 1 and 2 depends on the effort to find suitable pair of microcollisions and the amount of work necessary to find the appropriate value of $M_8$ in step 9.⋄. Microcollisions can be precomputed using around $2^{23}$ evaluations of functions $f$, $g$. The only part we need to deal with during the attack is the step 9.⋄. In our experiment we had to test $\approx 10000$ values of $M_8$ to find the right one. Since one test is roughly equivalent to computing single step in one branch of FORK (1/32 of the whole function), we can estimate the complexity of 9.⋄ to be less than $2^9$ evaluations of the compression function.

This algorithm (partially) uses the following variables: $IV[1]$, $IV[2]$, $IV[3]$, $IV[5]$, $IV[6]$, $M_0$, $M_2$, $M_4$, $M_6$, $M_8$, $M_9$, $M_{10}$, $M_{13}$, $M_{15}$. The following variables can have arbitrary values: $IV[0]$, $IV[4]$, $IV[7]$, $M_1$, $M_3$, $M_5$, $M_7$, $M_{11}$, $M_{12}$, $M_{14}$.

Finally, we present an example of a collision:

```
IV={6a09e667, ff03f03a, f7da19f9, a19f937d,
    510e527f, d1075199, c4bba02c, 00000000}
 M={97770819, 00000000, 90e31bf1, 00000000,
    e9b1a3b9, 00000000, 36ca5a85, 00000000,
    000024a1, 6ff47b82, 3f7bfaf6, 00000000,
    00000000, 014b4e3b, 00000000, 980100ed}
MM={b479fad2, 00000000, 90e31bf1, 00000000,
    e9b1a3b9, 00000000, 36ca5a85, 00000000,
    000024a1, 52f188c9, 3f7bfaf6, 00000000,
    00000000, 014b4e3b, 00000000, 980100ed}
```

Collisions for branches 3 and 4 can be obtained using exactly the same method by introducing appropriate differences in message words $M_{14}$ and $M_{15}$.

## 7 Near-collisions and possible collisions for the full compression function

In this section we show how a high-level path using differences in $M_{12}$ presented in section 5 can be used to find very low weight output differences of the compression function of FORK-256 and we argue that this approach might be applicable to finding full collisions faster than using the birthday paradox.

### 7.1 Overview

The foundation of our attack is the observation that if we introduce a difference in $M_{12}$ and we are able to prevent if from propagating to other registers in step 1 and step 5 of branch 4 and in step 4 of branch 3 and it does not introduce a difference in register $E_7$ of branch 1 then the output difference is confined to registers $B$,$C$,$D$ and $E$ only, ie. to at most 128 bits in total. This situation is illustrated in Figure 3.

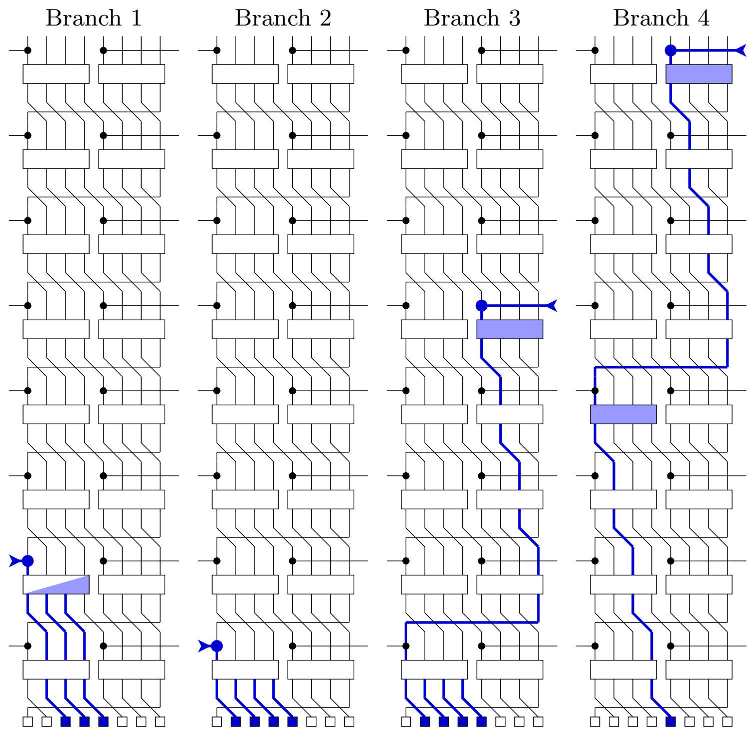
We can decrease the number of affected bits further by selecting a modular difference having differences in only a few most significant bits as the difference in register $B$ will be restricted to only those most significant bits as well.

Now, if we can find pairs of messages satisfying aforementioned conditions efficiently enough and we can assume that output differences have distribution close to uniform, we can expect to find very low weight differences and ultimately also a collision.

The attack consist of two phases. During the first one, we find simultaneous microcollisions in steps 1 and 5 of branch 4 and step 4 of branch 3 for a specially selected modular difference introduced in $M_{12}$.

In the second phase we use free message words $M_4$ and $M_9$ that do not interfere with already fixed microcollisions in branches 3 and 4 to find messages that yield no difference in register $E_7^{(1)}$ due to a single microcollision in line $D$ in step 7 of branch 1.

**Fig. 3.** High-level path used to find near-collisions in FORK-256. Thick lines show the propagation of differences. $Q$-structures for which microcollisions have to be found are grayed out.



## 7.2 Achieving microcollisions in branches 3 and 4

We assume that we have already chosen a suitable modular difference $d$. We proceed as follows.

- We start with branch 4. We find $x_1$ such that $x_1, x_1 + d$ give simultaneous $g$-$\delta_{15}$-$f$ microcollisions for step 1 of branch 4, compute corresponding constants $\tau_1, \tau_2, \tau_3$ and assign $IV[5] := \tau_1$, $IV[6] := \tau_2$, $IV[7] := \tau_3$. Set $M_{12}$ to $x_1 - IV[4]$ and $M'_{12}$ to $x_1 - IV[4] + d$.
- Fix arbitrary values of $M_5$, $M_1$, $M_8$, $M_{15}$, $M_0$, $M_{13}$ and $M_{11}$ and compute the first half of the branch, up to step 5. Then, in step 5 find a pair of values $x_2, x_2 + d^*$ (where $d^* = A_4^{(4)} - A_4'^{(4)}$ is the modular difference in register A after step 4) yielding simultaneous $f$-$\delta_6$-$g$ microcollisions and compute corresponding constants $\rho_1, \rho_2, \rho_3$. If no such solution exists, repeat this point, otherwise, set $M_3 := x_2 - A_4^{(4)}$.
- By manipulating message words $M_1$, $M_{15}$, $M_{13}$ (and also $M_0$ and $M_{11}$) we need to adjust the values of registers $B_4^{(4)}$, $C_4^{(4)}$, $D_4^{(4)}$ to $\rho_1, \rho_2, \rho_3$.

  - Since $B_4^{(4)} = A_3^{(4)} + M_{13} + \delta_8$ and we want $B_4^{(4)} = \rho_1$, we adjust the value of $B_4^{(4)}$ by setting $M_{13} := \rho_1 - A_3^{(4)} - \delta_8$.
  - Now, starting from $\rho_2$ we can go back one step and compute the necessary value of $\bar{B}_3^{(4)} = [\rho_2 \oplus g(A_3^{(4)} + M_{13} + \delta_8)] - f(A_3^{(4)} + M_{13})$. We can do this by setting $M_{15} := \bar{B}_3^{(4)} - A_2^{(4)} - \delta_{10}$. This change has also influenced the value of $E_3^{(4)}$ so we have to compensate it by adjusting the value of $M_{11}$.
  - Similarly, going back from $\rho_3$ two steps we can determine the necessary value of $B_2^{(4)}$ and adjust $M_1$ accordingly. Again, we need to compensate the induced change in $E_2^{(4)}$ by adjusting the value of $M_0$ and the change in $E_3^{(4)}$ by correcting $M_{11}$ again.
- Now we switch to branch 3. We choose values $x_3, x_3 + d$ that cause simultaneous $g$ - $\delta_6$ - $f$ microcollisions in step 4 and find corresponding constants $\lambda_1, \lambda_2, \lambda_3$. Using them we compute the necessary values of registers $E$–$H$, ie. $\bar{E}_3^{(3)} := x_3 - M_{12}$, $\bar{F}_3^{(3)} := \lambda_1$, $\bar{G}_3^{(3)} := \lambda_2$, $\bar{H}_3^{(3)} := \lambda_3$.
- Again, by going backwards and adjusting message words $M_2$, then $M_{14}$ and $M_{13}$ and then $M_6$ and $M_{10}$ and finally $IV[1]$ in a similar manner to what we did in branch 4 we obtain desired values of register at the beginning of step 4.
- Since we have just modified $IV[1]$ we need to go back to branch 4 and compensate for this change by adjusting the value $M_{11}$ once again.[2]

After this procedure we have obtained a differential path in branches 3 and 4 presented in Figure 3. The important fact is that changing the values of message words $M_4$ and $M_9$ do not change this path, so after fixing branches 3 and 4 we still have 64 bits of freedom left. Also note that there are many possible states of branches 3 and 4 following this path as at the beginning of the process we can select many arbitrary values, e.g. $M_5$, $M_8$, $M_7$ and $IV[0], IV[2], IV[4], IV[5]$. Additionally, there are many constants to choose from when fixing a microcollision.

### 7.3 Single microcollision in branch 1

What is left are branches 1 and 2. Fortunately, we do not need to pay attention to branch 2 at all as $M_{12}$ appears in the very last step and so in any case it induces differences in registers $B$–$E$ only.

In branch 1 we need a single microcollision in the third line of step 7. It seems to us that there is no better way of finding messages that cause that microcollision to happen than by randomly testing message words $M_4$ and $M_9$.

The probability of the success heavily depends on the modular difference in use. A few best modular differences we could find are presented in Table 7.

[2] Note that this description mentions modification of $M_{11}$ three times. Only the last one is necessary, but we include them all to make the process more intelligible by clear invariants.

**Table 7.** Best modular differences $d$ we could find and their probabilities of inducing a single microcollision in strand D of step 7 in branch 1. $\eta$ is the number of input values to strand $A$ that may result in the microcollision.

| 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 microcollision in terms of numbers of full FORK-256 evaluations. Denote by $\eta$ the number of allowable values for the modular difference in use. By an allowable value we mean an input $x$ for which there exist constants that cause a microcollision to happen for the pair $(x, x+d)$. For $d =$`22f80000` we have $\eta = 2^{21.7}$ (cf. Table 7). We proceed as follows.

- First, we fix the value of $M_4$. We will exhaust all values of $M_9$ for this value of $M_4$.
- Next, step through the computation up until step 7. We need to know all inputs into step 7.
- Then, for each allowable value into strand $A$ of step 7 (note that $M_{12}$ and $M'_{12}$ are already fixed) we step backwards one step to determine the corresponding "allowable outputs" to strand $G$ in step 5 and we store them in a hash table. For each allowable value we need to compute one XOR, one subtraction and store the element in a hash table, so the work effort for this step is about $1/64 \cdot \eta$ of full FORK-256 evaluations. For $\eta = 2^{21.7}$ the complexity of this step is about $2^{15.7}$.
- We loop through all $2^{32}$ values of $M_9$. For each one, we compute the output of $G$ only in step 5 and check if it matches something in the lookup table. If so, we proceed forward to see if it causes the difference to disappear in step 7. If not, we go to the next value of $M_9$.
  The cost of testing one value of $M_9$ is less than 1/64 of a full FORK evaluation. Essentially, we are computing less than a single $Q$-structure (FORK-256 consists of 64 of them). We assumed here that the cost of the table lookup is not exceeding the cost of computing the other parts of the $Q$-structure that we are omitting (strands $F$ and $H$), which seems to be a fairly safe assumption. For $\eta < 2^{22}$ values that match the allowable outputs we do a little bit more (about one more $Q$-structure in the left part of step 6), but the dominant term is $2^{32} \cdot 1/64 = 2^{26}$.
- After exhausting the list, we proceed with the next value of $M_4$.

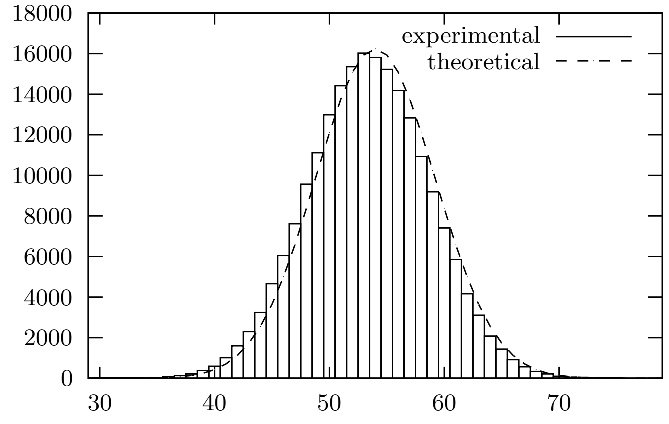
From the above analysis it follows that we process $2^{32}$ values of $M_9$ for the work effort of about $2^{26}$ full FORK evaluations. Since the observed probability of finding a solution is about $2^{-24.6}$ (cf. Table 7), we are getting about $2^{7.4}$ solutions for $2^{26}$ effort. This is equivalent to about $2^{18.6}$ FORK evalutations per solution.

## 7.4 Finding near-collisions: experimental results

If output differences are distributed uniformly on the positions where they can appear (ie. part of the register $B$ and register $C$, $D$, $E$) than we can expect a binomial distribution of their Hamming weights. After generating enough pairs we should be able to find some with exceptionally small weights, which can be called near-collisions.

We implemented this algorithm and performed some searches for such output differences with low weights. Comparison of the distribution of Hamming weights of differences obtained by the means of an experiment with theoretical binomial distribution is presented in Figure 4. It seems that the experimental distribution is indeed close to the theoretical one. However, one can see a slight bias towards lower weights.

**Fig. 4.** Distribution of Hamming weights of 211867 output differences generated by the algorithm for $d =$`0x22f80000`.



This phenomena can be explained by looking at Table 8 which contains counts of non-zero differences appearing at all bit positions of registers $B$, $C$, $D$ and $E$. The expected number of non-zero differences for a truly random bit stream would be $211867/2 \approx 105933$. All bits of registers $C$, $D$ and $E$ are very close to that value, but in some bits of register $B$ non-zero differences appear with a little lower probability. This most likely accounts for the bias towards smaller weights.

The best result we obtained so far has the output difference of weight 28 and is presented in Table 9. It took about a day of work of an average workstation PC to find it.

**Table 8.** Counts of non-zero bit differences in registers $B$, $C$, $D$ and $E$ after testing 211867 pairs of "close" hashes.

| Register | bits | counts | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| $B$ | 0 – 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 8 – 15 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 16 – 23 | 0 | 0 | 0 | 0 | 105812 | 102276 | 94549 | 82277 |
| | 24 – 31 | 68364 | 105987 | 102715 | 105458 | 97349 | 89530 | 105799 | 103214 |
| $C$ | 0 – 7 | 105460 | 105722 | 105723 | 105751 | 104016 | 106018 | 105458 | 105833 |
| | 8 – 15 | 105840 | 104074 | 106047 | 105842 | 105798 | 104450 | 99927 | 106058 |
| | 16 – 23 | 106255 | 106211 | 105918 | 108125 | 105860 | 105751 | 105785 | 105357 |
| | 24 – 31 | 110069 | 106080 | 105834 | 105726 | 106008 | 106559 | 106134 | 105892 |
| $D$ | 0 – 7 | 106072 | 105667 | 105443 | 105786 | 106165 | 106053 | 106019 | 105874 |
| | 8 – 15 | 105949 | 106556 | 105629 | 105597 | 105709 | 105308 | 102826 | 105302 |
| | 16 – 23 | 105637 | 105938 | 105993 | 104343 | 105727 | 106117 | 105800 | 105642 |
| | 24 – 31 | 106491 | 105858 | 105933 | 105595 | 104871 | 105884 | 106314 | 105622 |
| $E$ | 0 – 7 | 105496 | 105903 | 105954 | 105681 | 106193 | 105745 | 105652 | 105878 |
| | 8 – 15 | 103071 | 105674 | 106294 | 105795 | 105778 | 105893 | 105728 | 105701 |
| | 16 – 23 | 105913 | 105857 | 105977 | 105725 | 105963 | 106232 | 106061 | 105743 |
| | 24 – 31 | 106115 | 105974 | 107089 | 105738 | 105904 | 106000 | 105941 | 105671 |

**Table 9.** An example of an IV value and a message pair giving a pair of hashes differing on 28 bits.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| IV | `6a09e667` | `db1bb914` | `3c6ef372` | `a54ff53a` | `510e527f` | `767b0824` | `66410f7d` | `90f7ce64` |
| $M$ | `85a83e55` | `91d3ca9d` | `a6c2facb` | `027afd32` | `000000cb` | `00000000` | `9d4a6aba` | `00000000` |
| | `e649c148` | `4606ae35` | `6efb18d8` | `2d6ade8f` | `1dcb6936` | `ec995db1` | `d2ad257b` | `730f5bb4` |
| $M'$ | `85a83e55` | `91d3ca9d` | `a6c2facb` | `027afd32` | `000000cb` | `00000000` | `9d4a6aba` | `00000000` |
| | `e649c148` | `4606ae35` | `6efb18d8` | `2d6ade8f` | `40c36936` | `ec995db1` | `d2ad257b` | `730f5bb4` |
| diff | `00000000` | `8c300000` | `1d010204` | `52520104` | `c0908122` | `00000000` | `00000000` | `00000000` |

## 7.5 Feasibility of finding full collisions

Results from the previous subsection suggest that we can consider output differences to be very close to the uniform distribution. Using $d =$`0xdd080000`, there are 109 bits that may contain differences, but we know that differences in bit 19 of register $B$ will always cancel out each other. This means that since at most 108 bits are affected, after generating $2^{108}$ such pairs we expect to find a collision. We have already computed that the complexity of finding a single pair like this is about $2^{18.6}$ (or less, if better modular differences exist). So the total complexity of generating enough pairs to find a collision with high probability is $2^{108} \cdot 2^{18.6} = 2^{126.6}$, more than a factor of two better than the generic birthday attack.

It is worth mentioning that the additional advantage of our attack is that it does not need a huge storage, it requires only about $2 \cdot 2^{22}$ 32-bit words of memory for storing precomputed inputs for microcollisions and a hash table of similar size.

The above estimate is rather conservative, because if we multiply probabilities of single bit differences being zero (which can be easily derived from Table 8) 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 the computation of this product valid.

## 8 Conclusions

In this paper we exposed a number of weaknesses of the compression function of FORK-256. We showed how the unexpected property of $Q$-structures (allowing for finding microcollisions) can be exploited to easily find pairs of messages that after compressing result in a difference on only a small number of bits. We presented how this ease of finding output differences restricted to at most 108 bits may be exploited further to launch a collision-finding attack on the compression function faster and with smaller memory requirements than by birthday attack.

Our results are by no means final and complete. We expect that having more computational power to search for more favourable cases or investigating slighly different variations of the attacks we presented, it may be possible to improve them significantly.

Although we are intrigued by the design of FORK-256, at this point in time we are convinced that it should not be used in applications that require the highest level of security against collisions.

Finally, we would like to mention that an independent analysis of FORK-256 conducted by Mendel et al. [6] also discovered the existence of those pathological differentials and made use of them to produce collisions for two branches of FORK-256.

## References

1. R. Anderson and E. Biham. Tiger: A fast new hash function. In D. Gollmann, editor, *Fast Software Encryption – FSE'96*, volume 1039 of *LNCS*, pages 121–144. Springer-Verlag, 1996.
2. W. Bosma, J. Cannon, and C. Playoust. The Magma algebra system I: The user language. *Journal of Symbolic Computation*, 24(3–4):235–265, 1997. http://magma.maths.usyd.edu.au/.
3. D. Hong, J. Sung, S. Hong, S. Lee, and D. Moon. A new dedicated 256-bit hash function: FORK-256. First NIST Workshop on Hash Functions, 2005.
4. D. Hong, J. Sung, S. Lee, D. Moon, and S. Chee. A new dedicated 256-bit hash function. In *Fast Software Encryption – FSE'06*, LNCS. Springer-Verlag, 2006.
5. M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, 1998.
6. F. Mendel, J. Lano, and B. Preneel. Cryptanalysis of reduced variants of the FORK-256 hash function. Accepted to CT-RSA'07.

7. National Institute of Standards and Technology. Secure hash standard (SHS). FIPS 180-1, April 1995. Replaced by [8].
8. National Institute of Standards and Technology. Secure hash standard (SHS). FIPS 180-2, August 2002.
9. B. Preneel, A. Bosselaers, and H. Dobbertin. RIPEMD-160: A strenghtened version of RIPEMD. In D. Gollmann, editor, *Fast Software Encryption – FSE'96*, volume 1039 of *LNCS*, pages 71–82. Springer-Verlag, 1997.
10. R. L. Rivest. The MD4 message digest algorithm. In A. J. Menezes and S. A. Vanstone, editors, *Advances in Cryptology - CRYPTO'90*, volume 537 of *LNCS*, pages 303–311. Springer-Verlag, 1991.
11. R. L. Rivest. The MD4 message digest algorithm. Request for Comments (RFC) 1320, Internet Engineering Task Force, April 1992.
12. R. L. Rivest. The MD5 message digest algorithm. Request for Comments (RFC) 1321, Internet Engineering Task Force, April 1992.
13. B. Schneier and J. Kesley. Unbalanced Feistel networks and block cipher design. In D. Gollmann, editor, *Fast Software Encryption – FSE'96*, volume 1039 of *LNCS*, pages 121–144. Springer-Verlag, 1996.
14. X. Wang, X. Lai, D. Feng, H. Chen, and X. Yu. Cryptanalysis of the hash functions MD4 and RIPEMD. In R. Cramer, editor, *Advances in Cryptology – EUROCRYPT'05*, volume 3494 of *LNCS*, pages 1–18. Springer-Verlag, 2005.
15. X. Wang, Y. L. Yin, and H. Yu. Finding collisions in the full SHA-1. In *Advances in Cryptology - CRYPTO'05*, volume 3621 of *LNCS*, pages 17–36. Springer, 2005.
16. X. Wang and H. Yu. How to break MD5 and other hash functions. In R. Cramer, editor, *Advances in Cryptology – EUROCRYPT'05*, volume 3494 of *LNCS*, pages 19–35. Springer-Verlag, 2005.
17. X. Wang, H. Yu, and Y. L. Yin. Efficient collision search attacks on SHA-0. In *Advances in Cryptology - CRYPTO'05*, volume 3621, pages 1–16. Springer, 2005.
18. Y. Zheng, J. Pieprzyk, and J. Seberry. HAVAL – a one-way hashing algorithm with variable length of output. In J. Seberry and Y. Zheng, editors, *Advances in Cryptology - AUSCRYPT'92*, volume 718 of *LNCS*, pages 83–104. Springer-Verlag, 1993.