Meet-in-the-Middle Attacks on SHA-3 Candidates

Dmitry Khovratovich, Ivica Nikolić, and Ralf-Philipp Weinmann

University of Luxembourg
{dmitry.khovratovich,ivica.nikolic,ralf-philipp.weinmann}@uni.lu

Abstract. We present preimage attacks on the SHA-3 candidates Boole, EnRUPT, Edon-R, and Sarmal, which are found to be vulnerable against a meet-in-the-middle attack. The idea is to invert (or partially invert) the compression function and to exploit its non-randomness. To launch an attack on a large internal state we manipulate the message blocks to be injected in order to fix some part of the internal state and to reduce the complexity of the attack. To lower the memory complexity of the attack we use the memoryless meet-in-the-middle approach proposed by Morita-Ohta-Miyaguchi.

1 Introduction

Recent attacks on widely used hash functions standards [2,16] drew much attention to the hash function design not only from cryptographers, but also from the institutions responsible for the standardization. After several workshops and discussions had been held, NIST started the so-called SHA-3 competition [7], which called for new designs by the end of October 2008.

Since most attacks on hash functions have been differential-based collision attacks, the majority of the designs we investigated so far claimed to be resistant to differential cryptanalysis while to the resistance against other attacks were given less attention. The subject of this paper is meet-in-the-middle attacks and their application to preimage search.

A meet-in-the-middle attack on a cryptographic primitive is applicable if the execution can be expressed as a sequence of transformations all of which have at least one input that is independent of the other transformations. Providing the invertibility of the last transformation, the full execution can be divided into independent parts, which are connected using the birthday paradox.

One of the first such attack was the attack on Double-DES [3]. Double-DES, being composed of two consecutive iterations of single DES with different keys, was found to be vulnerable to the following meet-in-the-middle attack: given a pair (*plaintext*, *ciphertext*) one can find a Double-DES key (a pair of single DES keys), which is valid for this pair, with complexity of about 2^{32} encryptions. A full attack on Double-DES, which gives the real key, is based on this approach as well and it is faster than the brute-force.

Meet-in-the-middle attacks on hash functions based on the Merkle-Damgård construction are hard to apply since the compression function is usually assumed to be non-invertible. The alternative sponge construction [1] allows invertible transformations, but requires the internal state to be large so that the meet-in-the-middle approach can not be applied.

Surprisingly, several SHA-3 proposals are vulnerable to this type of attack. In this paper we describe meet-in-the-middle based preimage attacks on Boole, EnRUPT, Edon-R, and Sarmal. Two ideas are common for all the attacks. First, all the functions have invertible (or partially invertible) transformations, which allows us to execute the meet-in-the-middle. Secondly, we reduce the intermediate state space exploiting the non-random behavior of the round transformations.

This paper is composed as follows. First, we describe the meet-in-the-middle preimage attack in general and remind how it can be maintained with little memory. Then we show how preimages for Boole, EnRUPT, Edon-R, and Sarmal can be found. We also discuss possible computation-memory tradeoffs.

2 Meet-in-the-Middle Attacks on Hash Functions

Hash functions with invertible compression functions become susceptible to preimage attacks if the size of the internal state is too small. Preimages can be obtained by performing a meet-in-the-middle attack on the compression function. In this section we will describe this generic scenario in more details.

Let $F : D \to D$ and $G : D \to D$ be two random permutations and $H = G \circ F$ the composition of these permutations. In our setting, the function H is the hash function, F is defined as the compression function with a fixed IV and G is the inverse of the compression function for a fixed target value. Furthermore, we define auxilliary functions $\pi_{1,2} : D \times D \to D$ that map tuples to their first, respectively second component.

Assume we want to perform a meet-in-the-middle attack on h. The standard technique is to compute two sets

$$S_1 = \{(F(x), x) : x \in_R D\} \quad \text{and} \quad S_2 = \left\{\left(G^{-1}(y), y\right) : y \in_R D\right\}$$

such that $|S_1|\cdot|S_2| = |D|$. Either sorting these two sets in their first component or computing them in such a way that they are already ordered in this component allows us to easily find colliding values

$$\pi_1\left((F(x), x)\right) = \pi_1\left(\left(G^{-1}(y), y\right)\right)$$

by comparing the elements of the two sets in linear time. Each collision gives us a pair (x, y) such that $H(x) = y$. How to balance the size of the sets S_1 and S_2 depends on the relative cost of the function G^{-1} compared to an evaluation of the function F. It may for instance be that G is easily invertible, meaning an evaluation of G^{-1} costs about the same number of operations as an evaluation of the function F. In this case we choose the sets S_1 and S_2 to be of equal size $\left\lceil\sqrt{|D|}\right\rceil$. However, if the evaluation of G^{-1} is k times more expensive than the evaluation of F, we should choose the set $|S_1|$ to be of size $\sqrt{k \cdot |D|}$ and S_2 of size $\sqrt{k^{-1} \cdot |D|}$ to obtain a minimum number of overall operations. The memory complexity of this naive approach is non-neglible however: We need to store a total of $2 \cdot \left(\sqrt{|D|}(\sqrt{k} + \sqrt{k^{-1}}\right)$ elements of the domain D to carry it out. Storing both sets is not really necessary: Only the smaller should be stored, the values of the larger can be computed on the fly and compared against the elements of the smaller set.

In some cases the memory requirement can be completely eliminated by a technique based on Floyd cycle finding first described in an article by Morita, Ohta and Miyaguchi [6]. Although several works on hash functions refer to memoryless variants of meet-in-the-middle attacks [10,5], all of them cite either one or both papers by Quisquater and Delescaille on collision search for DES [12,11]. These two papers however do not directly deal with meet-in-the-middle attacks, but describe the technique of using distinguished points for collision search. Oorschot and Wiener describe the same technique for memoryless meet-in-the-middle later in [14].

2.1 Eliminating the Memory Requirement

Assume we are given another function $r : D \to \{0, 1\}$ which maps elements of the domain D to a single bit in a random fashion. Using this *switching function* we can define a step function s that evaluates x either to $F(x)$ or to $G(x)$, depending on the value of x:

$$s : D \to D, \quad x \mapsto \begin{cases} F(x) & \text{if } r(x) = 0 \\ G(x) & \text{if } r(x) = 1 \end{cases}$$

This function s can then be used in a Floyd cycle finding algorithm: We start from a random value $x \in D$ and use just two elements $a = s(x)$ and $b = s^2(x)$. In each step we then update a by applying s to it and b by applying s^2 to it. Upon finding a cycle, we must check whether we really have found a pair $F(x) = G^{-1}(y)$ or whether we have found a cycle in F or in G. If the output of r is equidistributed, for each cycle we find $\Pr(F(x) = G^{-1}(y)) = 0.5$. In case of encountering a cycle in F or G we restart the algorithm with another random element $x \in D$.

Significant problems can arise if the output of r is not equidistributed, for instance if G is very costly to compute relative to F and we want to simulate the case of $|S_1| = k \cdot |S_2|$ with k large.

For the hash functions that we attack we define two functions F and G that are used in the memoryless approach. The F function is used for the forward direction and the G function is used for the backward one. The switching function r is defined as the parity of x.

2.2 Reduced State Principle

The meet-in-the-middle (MITM) attack needs a collision in the intermediate state. However, the state may be so large that a straightforward application of the MITM approach would require more than 2^n computations for a n-bit hash digest. Thus the generic principle we use further is to generate intermediate states only from a smaller subspace (where some bits are fixed to zero) thus reducing the birthday dimension and the complexity of the attack.

The generic framework is defined as follows. A hash function with an n-bit digest has an internal state of size k bits. We manage to get intermediate states with t bits fixed to 0. Then to get a MITM connection we need to get two states that collide in $(k-t)$ bits so that the *birthday space* D has size 2^{k-t}. This implies that we must get two sets S_1 and S_2 such that $|S_1| \cdot |S_2| = 2^{k-t}$. The exact ratio between S_1 and S_2 is defined by the complexity of inverting the compression (round) function.

For the memoryless version of the MITM attack, we need to tweak the attack slightly such that we can define the functions F and G. Each of the functions is a composition of two functions, first projecting the birthday space into the state space, the second mapping the state space into the birthday space again (fixing some bits to zero). In other words, let $F = f \circ \mu$ and $G = g \circ \nu$. When memoryless meet-in-the-middle is possible in our attacks we will define these functions accordingly.

3 Boole

Boole is a family of hash functions [13] based on a stream design. Internally, Boole has a large state $\sigma_t = (R_t[0], R_t[1], \ldots, R_t[15])$ of 16 words plus 3 additional word accumulators denoted by l_t, x_t, and r_t (t is the time). The words are 64 bits each. Hashing a message in Boole is done in three phases: 1)Input phase, where the whole message is processed word by word, and for each input word the state and the accumulators are updated, 2)mixing phase, where only the state is updated depending on the values of the accumulators, 3)output phase, where the output is produced.

The *update of the state*, referred to as a *cycle*, is defined as:

$$\begin{aligned}
R_{t+1}[i] &\leftarrow R_t[i+1], \text{for } i = 1 \ldots 14\\
R_{t+1}[15] &\leftarrow f_1(R_t[12] \oplus R_t[13]) \oplus (R_t[0] \lll 1)\\
R_{t+1}[0] &\leftarrow R_{t+1}[0] \oplus f_2(R_{t+1}[2] \oplus R_{t+1}[15]),
\end{aligned}$$

where f_1 and f_2 are some non-linear functions, intended to simulate random functions.

Let w_t be a message word. The *update of the accumulators* is defined as:

$$\begin{aligned}
temp &\leftarrow f_1(l_t) \oplus w_t\\
l_{t+1} &\leftarrow temp \lll 1\\
x_{t+1} &\leftarrow x_t \oplus w_t\\
r_{t+1} &\leftarrow (r_t \oplus temp) \ggg 1
\end{aligned}$$

The whole message is absorbed in the input phase. Sequentially, for each message word w_t the following is done:

1. *update the accumulators*
2. $R_t[3] \leftarrow R_t[3] \oplus l_{t+1}$
3. $R_t[13] \leftarrow R_t[13] \oplus r_{t+1}$
4. *update the state (cycle)*

The mixing phase is invertible and its description is irrelevant in our attack.

Each iteration of the output phase produces one output word. One iteration is defined as:

1. *cycle*
2. Output the word $v = R[0] \oplus R[8] \oplus R[12]$

For example, the output for Boole-256 is produced in 8 iterations.

Let us present two observations about the invertibility of the update functions of the state and the accumulators.

Observation 1. The state update (*cycle*) is an invertible function. If a new state σ_{t+1} is given, then the state σ_t that produced σ_{t+1} in a single *cycle* can be found from the following equations:

$$\begin{aligned} R_t[0] &= (R_{t+1}[15] \oplus f_1(R_{t+1}[11] \oplus R_{t+1}[12])) \ggg 1 \\ R_t[1] &= R_{t+1}[0] \oplus f_2(R_{t+1}[2] \oplus R_{t+1}[15]) \\ R_t[i] &= R_{t+1}[i-1], i = 2, \dots 15 \end{aligned}$$

Observation 2. The update of the accumulators can be inverted with probability $1 - 1/e$. If the values of the new accumulators $l_{t+1}, x_{t+1}, r_{t+1}$ and the input message word w_t are fixed, the values of the previous accumulators l_t, x_t, r_t are determined as:

$$\begin{aligned} l_t &= f_1^{-1}((l_{t+1} \ggg 1) \oplus w_t) \\ x_t &= x_{t+1} \oplus w_t \\ r_t &= r_{t+1} \lll 1 \oplus f_1(l_t \oplus w_t) \end{aligned}$$

Moreover, if the values of l_t, l_{t+1} (or r_t, r_{t+1}) are fixed, the value of the message word w_t can be found uniquelly:

$$\begin{gathered} w_t = (l_t + 1 \ggg 1) \oplus f_1(l_t) \\ (\ w_t = (r_{t+1} \lll 1) \oplus r_t \oplus f_1(l_t)) \end{gathered}$$

In order to invert the function f_1 we will use a look-up table $(x, f_1(x))$ with all 2^{64} values for x, sorted by the second entry. Then, a inversion of $f_1(x)$ is equivalent to a look-up in this table.

3.1 Preimage Attack on Boole-384 and Boole-512

The intermediate state of Boole has 16 state words and 3 accumulators, hence 19 words in total. Further, we will show how to fix the values of the state words $R[3], \dots, R[12]$ (10 words in total) to zero in forward and backward directions. This will mean that $k = 19 \cdot 64 = 1216$ and $t = 10 \cdot 64 = 640$, and the birthday space D has only 9 words (576 bits). We will also define $f(x)$ and $g(x)$ for the memoryless MITM attack.

Defining μ - fixing $R[3], R[4], \ldots, R[12]$ forwards. From the description of the input phase it follows that:

$$R_{10}[3] = R_9[4] = \ldots = R_1[12] = R_0[13] \oplus r_1$$

Note that the value of r_1 can be controlled with w_0 (Observation 2). Hence, if we take $r_1 = R_0[13]$, we will get $R_{10}[3] = 0$. Similarly, for $R_{10}[4]$ we have:

$$R_{10}[4] = R_9[5] = \ldots = R_2[12] = R_1[13] \oplus r_2$$

We can change the value of r_2 with w_1 such that $R_1[13] \oplus r_2 = 0$ holds. Then $R_{10}[4] = 0$. The same technique can be applied for fixing the values of $R_{10}[5], \ldots, R_{10}[12]$.

Note that we can not fix the values of more than these 10 words. When we control the value of r_t with the input word w_{t-1}, it means that we also change the value of l_t (which is added to $R_t[3]$). Since we can not control the value of both accumulators with a single message word, and both of them are xor-ed into the registers $R[3]$ and $R[13]$, it means that we can not control the values of more than 10 words.

Defining $f(y)$ for the memoryless MITM attack. The birthday space D has 9 words. Let $y = y_1||y_2||\ldots||y_9$, then $f(y)$ can be defined as compression of the input words y_i with $1 \leq i \leq 9$ in the first 9 *cycles*. Thus when fixing $R[3], \ldots, R[12]$ in forward direction, we first compress y, and then we start with our technique for fixing these words to zero (function μ).

Defining ν - fixing $R[3], R[4], \ldots, R[12]$ backwards. Our backwards strategy is the following: first we invert the output and the mixing phase and obtain one valid intermediate state. Then, by changing the input words, we fix $R[3], R[4], \ldots, R[12]$.

First, let us deal with the inversion of the output phase. In each *cycle* of this phase one output word is produced. Hence, the digest is produced in 8 cycles[1]. The output word v_t is defined as $v_t = R_t[0] \oplus R_t[8] \oplus R_t[12]$. Let $H^* = (h_0, \ldots, h_7)$ be the target hash value. We have to construct a state $\sigma_t = (R_t[0], \ldots, R_t[15])$ such that $h_0 = v_t, h_1 = v_{t+1}, \ldots, h_7 = v_{t+7}$. First, we put any values in $R_t[0], R_t[9], R_t[10], \ldots, R_t[15]$. The rest of the words are undefined. Then, we find $R_t[8]$ from the equation $R_t[8] = R_t[0] \oplus R_t[12] \oplus h_0$. Obviously we get that $v_t = h_0$. After the *cycle* update we obtain a new state σ_{t+1}. Then, we determine the value of $R_t[1]$ from the equation $R_t[1] = R_{t+1}[0] = R_{t+1}[8] \oplus R_{t+1}[12] \oplus h_1$, and therefore $h_1 = v_{t+1}$. The values for $R_t[2], \ldots, R_t[7]$ are determined similarly. This way we can define the rest of the words in the state σ_t, which in the 7 sequential *cycle* updates produces the target hash value.

Let us fix the accumulators to any values. Then, inverting the mixing phase is trivial because the length of the preimage, as shown further, is known and the values of the accumulators are also known.

[1] In Boole-384, the output is produced in 6 cycles.

Now that we have inverted the output and mixing phase, we have the freedom of choosing the input message words. The technique for fixing is rather similar to the one used for fixing this set in forward direction. But in the backward direction, we control the values of the l_t accumulators (rather then the values of r_t as in the forward direction) with the input words w_t (Observation 2). From the description of the input phase we get:

$$R_{10}[12] = R_{11}[11] = \ldots = R_{18}[4] = R_{19}[3] \oplus l_{20}$$

Therefore if we take $l_{20} = R_{19}[3]$ we will get $R_{10}[12] = 0$. Similarly, for $R_{10}[11]$ we have:

$$R_{10}[11] = R_{11}[10] = \ldots = R_{17}[4] = R_{18}[3] \oplus l_{19}$$

If we take $l_{19} = R_{18}[3]$ we obtain $R_{10}[11] = 0$. The same technique can be used to fix the variables $R_{10}[10], \ldots, R_{10}[3]$.

One may argue that for controlling the values of the l_t registers when going backwards we have to pay an additional cost because f_1 is not always invertible. But we have to keep in mind that there are values for which f_1 has many inversions. Hence, if we start with a set of N different values, we can expect to find N different inversions for these values and thus we do not have to repeat the inversion.

Defining $g(y)$ for the memoryless MITM attack. The function $g(y)$, where $y = y_1||y_2||\ldots||y_9$, is defined as 9 consecutive backward rounds of the input phase with inputs y_i. The starting state of these 9 rounds is the state obtained after the inversion of the output and mixing phases (as described above). Note that after the application of the function $g(y)$ a new state is obtained. Then, to this state, we apply our technique for fixing $R[3], \ldots, R[12]$ in 10 backwards rounds (function ν).

3.2 Complexity of the Attack

The preimage that we obtained has a length of at least $9 + 9 + 10 + 10 = 38$ words. The memoryless MITM attack requires about $2^{\frac{9 \cdot 64}{2}} = 2^{288}$ computations[2] and 2^{64} memory (for inverting f_1).

4 Edon-R

The hash family Edon-R [4] uses the well known Merkle-Damgård design principle. The intermediate hash value is rather large, two times the digest length[3]. For an n-bit digest the chaining value H_i of Edon-R is composed of two block of n bits each, i.e. $H_i = (H_i^1, H_i^2)$. The message input M_i for the compression

[2] One computation is equivalent to one round of the input phase or one round of the mixing phase.

[3] Edon-224 and Edon-384 have 512 and 1024 bits chaining values, respectively.

function is also composed of two blocks, i.e. $M_i = (M_i^1, M_i^2)$. Let Edon be the compression function. Then the new chaining value is produced as follows:

$$H_{i+1} = (H_{i+1}^1, H_{i+1}^2) = \text{Edon}(M_i^1, M_i^2, H_i^1, H_i^2)$$

The hash value of a message is the value of second block of the last chaining value.

Internally, the state of Edon-R has two n-bit blocks, A and B. The compression function of Edon-R consists of eight updates, each being an application of the quasigroup operation $Q(x, y)$[4], to one of these blocks. With A_i and B_i we will denote the values of these blocks after the i-th update in the compression function (please refer to Fig. 1). Hence, each input pair (H_i, M_i) generates internal state blocks $(A_1, B_1), (A_2, B_2), \dots, (A_8, B_8)$. The new chaining value (the output of the compression function) H_{i+1} is the value of the blocks (A_8, B_8).

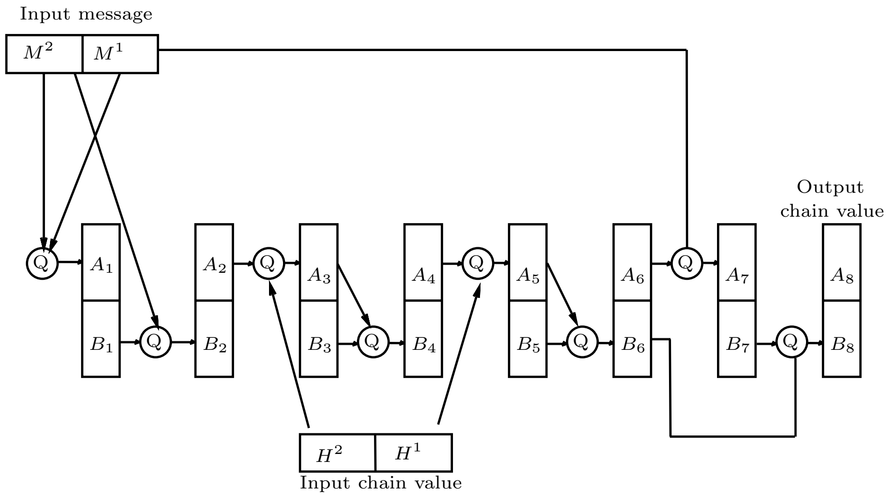

Fig. 1. Outline of the Edon-R compression function

Let us present a simple observation that is used in the attack.

Observation. The quasigroup operation $Q(x, y)$ of Edon-R is easily invertible, i.e. if A and C (B and C) are fixed then one can easily find B (C) such that $Q(A, B) = C$.

4.1 Preimage Attack on Edon-R-n

The internal state of Edon-R-n (the chaining value $H = H_1 || H_2$) has $2n$ bits. We will show how to fix $H_1 = 0$. Then the preimage attack can be mounted using the MITM approach (Section 2), where $k = 2n$ and $t = n$. The backward step is time-consuming so we will use the memory MITM attack.

[4] The exact definition of the quasigroup operation can be found in [4].

Fixing H_1 in forward direction. We need only one message block to get the desired $H_1^{\mathrm{new}} = 0$. Both initial value blocks are fixed as well. We claim that for each M_1 we can find M_2 such that this message input and the initial value blocks will produced a zero value in H_1^{new}.

Indeed, let M_1 be set to some random value. Then we obtain the value of A_6 since $A_7 = H_1^{\mathrm{new}} = 0$ and the function Q is invertible. We consecutively obtain the values of A_5, A_4, A_3, A_2, and A_1 (keep in mind that the initial chaining value is fixed). Given A_1 and M_1, we derive M_2 by inverting the first application of Q. Finally we obtain all B's and thus a pair $(H_1^{\mathrm{new}} = 0, H_2^{\mathrm{new}})$.

Fixing H_1 in backward direction. We need only one step (one message block) to get a pair of the form $(0, H_2)$ from a given hash value $H = H_2^{\mathrm{new}}$.

First, we set M_1 to some predefined value m. Then we assign to A_8 some random value and consecutively obtain the values of the following internal variables (in this order): A_7, B_7, B_6, A_6 (using M_1), A_5, B_5, B_4, A_4, A_3, B_3. We repeat this step 2^k times for different values of A_8 and store 2^k different pairs (A_3, B_3).

Now we set M_2 to some random value[5] and obtain the values of A_1, A_2, and B_2 using the value of M_1. If we repeat this step 2^{n-k+s} times then we will find 2^s different values of B_2 that coincide with some values of B_3 from the stored set. For each of these values we define H_2 such that $Q(A_2, H_2) = A_3$. The complexity requirements for this part are: 2^{n-k+s} computations[6], where $s - k < 65$, and $2^s + 2^k$ memory.

These 2^s pairs can be obtained using the memoryless MITM as well, where the MITM space is the value of B_2. Because of the message padding we should take any $n-65$ bits of B_2 so that the input and the output of the MITM function F and G would have the same size. The $(n-65)$-bit input to the function F is padded with the message padding, and the input to the function G is padded, for example, with zeros. Then, if a $(n-65)$-bit collision between F and G is obtained, the probability that they coincide in the rest of the 65 bits is 2^{-65}. Hence, for constructing 2^s pseudo preimages with the memoryless MITM, one needs $2^s \cdot 2^{\frac{n-65}{2}+65} = 2^{\frac{n}{2}+s+32.5}$.

4.2 Complexity of the Attack

Starting from the initial value, we generate 2^{n-s} different chaining values with $H_1 = 0$. Note that we do not store these values. Then, with high probability, we can expect that one of these values will be in the set of the 2^s pseudo preimages generated in the backward direction. Under the condition $s - k < 65$ the total complexity of the attack when memory is used in the backward step is $2^{n-s} + 2^{n-k+s}$ computations and $2^s + 2^k$ memory. If only negligible memory in the backward step is used the computational complexity is $2^{n-s} + 2^{\frac{n}{2}+s+32.5}$ at the same time needing 2^s memory.

[5] The value is not truly random: 65 bits of the last message block are reserved for padding.

[6] Here and below, one computation is not more than one compression function call.

5 EnRUPT

The family of hash functions EnRUPT [9] is a member of a set of cryptographic primitives first presented at SASC 2008 [8].

The pseudocode of 512-bit version of EnRUPT, called ïrRUPT-512, is presented below. For details we refer the interested reader to [9].

Algorithm 1. ïrRUPT-512

```
Require: $p_0, \dots, p_n$ { message blocks}
  $(d_0, d_1, r) \leftarrow (0, 0, 0)$
  $(x_0, \dots, x_{47}) \leftarrow (0, \dots, 0)$
  for $i = 0$ to $n$ do
    $(d_0, d_1, r, (x_0, \dots, x_{47})) \leftarrow$ ïr8$(p_i, d_0, d_1, r, (x_0, \dots, x_{47}))$ {squeezing}
  end for
  $(d_0, d_1, r, (x_0, \dots, x_{47})) \leftarrow$ ïr8$(512, d_0, d_1, r, (x_0, \dots, x_{47}))$
  for $i = 0$ to 199 do
    $(d_0, d_1, r, (x_0, \dots, x_{47})) \leftarrow$ ïr8$(0, d_0, d_1, r, (x_0, \dots, x_{47}))$ {blank rounds}
  end for
  for $i = 0$ to 7 do
    $(d_0, d_1, r, (x_0, \dots, x_{47})) \leftarrow$ ïr8$(0, d_0, d_1, r, (x_0, \dots, x_{47}))$
    $z_i \leftarrow d_1$ {output}
  end for
  return $(z_0, \dots, z_7)$
```

Algorithm 2. ïr8

```
Require: $p, d_0, d_1, r, (x_0, \dots, x_{47})$
  for $k = 0$ to 7 do
    $t \leftarrow (9 \cdot ((2 \cdot x_{(r \oplus 1) \bmod 48} \oplus x_{(r+4 \bmod 48)} \oplus d_{r\&1} \oplus r) \ggg 16)$
    $x_{(r+2) \bmod 48} \leftarrow x_{(r+2) \bmod 48} \oplus t$
    $d_{r\&1} \leftarrow d_{r\&1} \oplus t \oplus x_{r \bmod 48}$
    $r \leftarrow r + 1$
    $d_1 \leftarrow d_1 \oplus p$
  end for
  return $(d_0, d_1, r, x_0, \dots, (x_0, \dots, x_{47}))$
```

In the pseudo-code all indices are taken modulo 16, all multiplications are performed modulo 2^{64}, $\ggg$ stands for cyclic rotation to the right, and $\oplus$ denotes XOR. Now let us define and explain some points that are further used in the attack on ïrRUPT-512.

Equation invertibility. The accumulators d_i are updated by a non-invertible function, which can be expressed as $x \oplus g(x \oplus y)$ (see pseudo-code). Given the output of the function and the value of x a solution does not always exist. However, if we assume that the output and y are independent then the probability that the function can be inverted can be estimated by $1 - 1/e$. We did statistical tests that support this estimate.

Furthermore, while there is no solution for some input there are two (or more) solutions for other inputs (one solution on average). Thus when we perform backtracking we actually do not lose in quantity of solutions.

Look-up tables. We use look-up tables in order to find a solution for the equations arising from the round functions. All the tables used below refer to functions that have space of arguments smaller than the complexity of the attack, e.g., when we try to solve an equation $f(x \oplus C) = x$ (where C is one of 2^{64} possible constants) we use 2^{64} precomputed tables that contain values of $f(x \oplus C) \oplus x$ for all C and x.

Solving a system of equations is more complicated. Below we solve systems of form

$$\begin{cases} x = f(x, y, z, C_1); \\ y = g(x, y, z, C_2); \\ z = h(x, y, z, C_3), \end{cases}$$

where C_i are constants. We precompute for all possible x, y, z, C_i (2^{384} tuples) the sums $x \oplus f(x, y, z, C_1), y \oplus g(x, y, z, C_2)$, and $z \oplus h(x, y, z, C_3)$ and then sort them so that it is easy to find a solution (or many) given C_i.

We also estimate that the time needed to find a solution is given by the complexity of the binary search which is negligible compared to the table size.

Inverting the updates in ïr8(p_i). The compression function of ïrRUPT-512 consists of the update of the state words $x_0, x_1, \ldots, x_{15}$, and the update of the accumulators d_0 and d_1. Inverting the update of the state words $x_0, x_1, \ldots, x_{15}$ is trivial:

$$x_{r+2}^{\text{old}} = x_{r+2}^{\text{new}} \oplus f.$$

The accumulator d_0 (similar formula holds for d_1) is updated by the following scheme:

$$d_0^{\text{new}} = f(x_{r \oplus 1}, x_{r+4}, d_0^{\text{old}}, r) \oplus d_0^{\text{old}} \oplus x_r$$

Instead of solving this equation for d_0^{old}, we simply use a table look-up (see above). Since the arguments of f are xored, we solve an equation of form $f(x \oplus C_1) \oplus x = C_2$. We spend $(2^{64})^2 = 2^{128}$ memory and effort to build this table for all x and C_1.

5.1 Preimage Attack on ïrRUPT-512

The preimage attack is mounted using the MITM approach (Section 2). The internal state of ïrRUPT-512 has 18 words, hence 1152 bits. We will show how to fix x_3 and x_{11} in forward and backward directions. Also, since EnRUPT does not have a message schedule and just adds the message block, we can reduce the birthday space D for an additional one word. Hence, the parameters for MITM are $k = 1152$ and $t = 192$. Getting states in both directions in not time-consuming. Therefore we will define the functions $f(x)$ and $g(x)$ and launch a memoryless MITM attack.

Defining μ - fixing x_3 and x_{11} in forward direction. We will fix the values of these two words in two consecutive application of the compression function. We will fix the value of x_3 to zero by changing the previous input message word p_0. In the following compression function iteration this value is not changed. In this iteration, we fix the value of x_{11} by setting the value of p_1.

From the definition of x_3 (notice that x_3 is updated second in the iteration but does not depend on x_2 and d_0, which has been updated before) we have:

$$x_3^{\text{new}} = 9[(2x_0 \oplus x_7 \oplus d_1) \ggg 16] \oplus x_3^{\text{old}}$$

We want to fix the value of x_3 to zero. Hence we require:

$$0 = 9[(2x_0 \oplus x_7 \oplus d_1) \ggg 16] \oplus x_3^{\text{old}}$$

In this equation the value of d_1 can be chosen freely. Simply, in the previous iteration of the compression function, the message word p, which is added to d_1 ($d_1^{\text{new}} = d_1^{\text{old}} \oplus p$) can be changed without affecting the values of the state words and d_0.

Therefore, by using a predefined table for this equation, we can find the necessary value of d_1 so that the equation holds. To build this table we spend $(2^{64})^4 = 2^{256}$ memory and computations. Notice that after the value of x_3 is fixed then, in iteration that follows, this value is not changed. In this iteration, we fix the value of x_{11} using exactly the same method. Hence, in two sequential rounds, we can fix the value of exactly two state words: x_3 and x_{11}.

Defining $f(y)$ for the memoryless MITM attack. The birthday space D has 15 words. We denote $y = y_1||y_2||\ldots||y_{15}$. Then $f(y)$ can be defined as compression of the input words $y_i, i = 1, \ldots, 15$ in the first 15 applications of the compression function. Thus when fixing x_3 and x_{11} in forward direction, we first compress y, and then we start with our technique for fixing these two words to zero.

Defining ν - fixing x_3 and x_{11} in backward direction. When going backwards we have to take into account two things: 1)the output hash value is produced in 8 iterations, and 2)the input message words in the last 17 iterations are fixed. Let us first address 1). When the hash value is given (as in a preimage attack), it is still hard to reconstruct the whole state of ÏRRUPT-512. This is made more difficult by outputting only a small chunk of the state (the value of d_1) in each of the 8 final iterations (and not at once). So, not only we have to guess the value of the rest of the state, but we have to guess it so that in the following iterations the required values of d_1 will be output. Yet, this is possible to overcome.

Let the hash value be $H = (d_1^t, d_1^{t+1}, \ldots, d_1^{t+7})$. Consider a state where $d_1 = d_1^t$ and all of the other words of the state are left undefined. Then, we take 2^{448} different values for the rest of the state and iterate forward for 7 rounds, while producing an output word at each round. With overwhelming probability, one of these outputs will coincide with $(d_1^{t+1}, \ldots, d_1^{t+7})$. After we find the state that produces the required output, we go backwards through the blank iterations and the message length iteration. In total there are 17 iterations which is 136 rounds. The accumulators are updated non-bijectively. Therefore one may argue that the cost of inverting the accumulators through these rounds should be $(1-1/e)^{136}$. Yet, if in some cases solution for the accumulator doesn't exist in other cases there is more then one solution. Hence, if we start with two internal states, we can pass these iterations with a cost of two times hashing in forward direction.

Now after we have passed the output, blank rounds and message length iterations, and obtained one state, we can fix x_3 and x_{11} in two backward applications of the compression function. The following lemma holds:

Lemma 1. *Given a state* $S = (x_0^{\text{new}}, \ldots, x_{15}^{\text{new}}, d_0^{\text{new}}, d_1^{\text{new}})$ *one can build a state* $S' = (x_0, \ldots, x_{15}, d_0, d_1)$ *and a message* p *such that* $x_3 = 0$ *and* $\text{ir8}(S', p_i) = S$.

The proof is given in Appendix. The same proposition can be applied to x_{11}. Since the compression function in one application changes either x_3 or x_{11}, then in two consecutive backward applications of the compression function we can fix the values of these two words.

Defining $g(y)$ for the memoryless MITM attack. The function $g(y)$, where $y = y_1||y_2||\ldots||y_{15}$, is defined as 15 consecutive backward rounds of the input phase with inputs y_i. The starting state of these 9 rounds is the state obtained after the inversion of the output, blank rounds and message length iterations (as described above).

5.2 Complexity of the Attack

We spend at most 2^{384} computations to build the pre-computation tables so it is not a bottleneck. To compose a valid state after the blank rounds that gives the desired hash we need about 2^{448} trials. We also pass the blank rounds for free since the absence of solutions for some states is compensated by many of them for other ones. Thus the most computations-consuming part is the memoryless MITM attack. It requires $2^{\frac{960}{2}} = 2^{480}$ computations. The memory requirement is determined by the precomputed tables, hence it is 2^{384}.

6 Sarmal

Sarmal-n [15] is a hash family based on the HAIFA design. After the standard padding procedure, the padded message is divided into blocks of 1024 bits each, i.e. $M = M_1||M_2||\ldots||M_k, |M_i| = 1024, i = 1, \ldots, k$. Each block is processed by the compression functions. HAIFA design implies that the compression function f has four input arguments: the previous chain value h_{i-1}, the message block M_i, the salt s, and the block index t_i. Hence, h_i is defined as $h_i = f(h_{i-1}, M_i, s, t_i)$. The final chaining value h_k is the hash value of the whole message M. For Sarmal-n the chaining value h_i has 512 bits. Let us denote the left and the right half of

h_i as L_i and R_i respectively, i.e. $h_i = L_i||R_i$. The salt s has 256 bits (similarly let $s = s_1||s_2$), and the block index t_i has 64 bits. Then, the compression function of Sarmal-n can be defined as:

$$f(h_{i-1}, M_i, s, t_i) = \mu(L_{i-1}||s_l||c_1||t_i, M_i) \oplus \nu(R_{i-1}||s_r||c_2||t_i, M_i) \oplus h_{i-1}, \quad (1)$$

where μ and ν are functions that output 512 bit values, and c_1, c_2 are some constants. The exact definition of these functions is irrelevant for our attack.

6.1 Preimage Attack on Sarmal-512

We will show how to invert the compression function of Sarmal-512. Note that the intermediate chaining value of Sarmal has 512 bits. Then the preimage attack can be launched using the MITM approach (Section 2), where $k = 512$ and $t = 0$. The inversion of the compression function is time-consuming so we will use the memory MITM attack.

Going forward from the IV. Since we do not fix anything ($t = 0$), going forward from the IV is trivial. We simply generate a number of intermediate chaining values, by taking different random messages as an input for the first compression function.

Going backward from the target hash value. Let us explain how the compression function can be inverted.

From (1) we get:

$$\begin{aligned} f\ (\ & h_{i-1}, M_i, s, t_i) = \\ = \mu\ & (L_{i-1}||s_l||c_1||t_i, M_i) \oplus \nu(R_{i-1}||s_r||c_2||t_i, M_i) \oplus h_{i-1} = \\ = \mu\ & (L_{i-1}||s_l||c_1||t_i, M_i) \oplus \nu(R_{i-1}||s_r||c_2||t_i, M_i) \oplus L_{i-1}||R_{i-1} = \\ = \mu\ & (L_{i-1}||s_l||c_1||t_i, M_i) \oplus \nu(R_{i-1}||s_r||c_2||t_i, M_i) \oplus L_{i-1}||0 \oplus 0||R_{i-1} = \\ = (\mu\ & (L_{i-1}||s_l||c_1||t_i, M_i) \oplus L_{i-1}||0) \oplus (\nu(R_{i-1}||s_r||c_2||t_i, M_i) \oplus 0||R_{i-1}) \end{aligned}$$

Let us fix the values of M_i, s, and t_i. Then, we can introduce the functions $F(L_{i-1}) = \mu(L_{i-1}||s_l||c_1||t_i) \oplus L_{i-1}||0$, and $G(R_{i-1}) = \nu(R_{i-1}||s_r||c_2||t_i) \oplus 0||R_{i-1}$. Let H^* be the target hash value. Then we get the equation:

$$F(L) \oplus G(R) = H^*$$

If we generate 2^{256} different values for $F(L)$ and the same amount for $G(R)$, then, by the birthday paradox, with high probability we can expect to get at least one pair $(F(L_l), G(R_m))$ that will satisfy the above equation and therefore obtain that $h = L_l||R_m$ is a preimage of H^*.

A memoryless version of this pseudo-preimage attack can be obtained by introducing the function $\tilde{F}(L) = F(L) \oplus H^*$, and launching the memoryless MITM attack on $\tilde{F}$ and G. This would require 2^{256} computations and negligible memory.

Table 1. Complexity of the preimage attacks described in this paper

	Computations	Memory
Boole-384/512	2^{288}	2^{64}
Edon-R-n	$2^{n-s}+2^{n-k+s}$	2^s+2^k
	$2^{n-s}+2^{\frac{n}{2}+s+32.5}$	2^s
EnRUPT-512	2^{480}	2^{384}
Sarmal-512	$2^{512-s}+2^{256+s}$	2^s

6.2 Complexity of the Attack

Since the backward direction, i.e. inverting the compression function, is time consuming we will use the memory version of MITM attack. Going backwards from the target hash value we create a set S_2 of 2^s different chaining values. To create this set we need $2^{256}\cdot 2^s = 2^{256+s}$ computations. Then, starting from the initial value, we generate 2^{512-s} different chaining values. Note, we do not store these values, we store only the smaller set S_2. Then, with a high probability, we can expect that these two sets coincide. The total complexity of the attack is $2^{512-s}+2^{256+s}$ computations and 2^s memory.

7 Conclusions

We have presented meet-in-the-middle attacks on four SHA-3 candidates. These attacks became possible because we managed to invert (or partially invert) the compression functions and to reduce the birthday space so that collisions in this space can be found faster than 2^n and give a preimage.

We have also applied, when it was possible, the memoryless version of the MITM attack and thus significantly reduced the memory requirements for the attacks. For these cases we provided estimates on the computation-memory tradeoffs.

The complexity of our attacks on the hash functions are summarized in the following table.

Acknowledgments

The authors would like to thank Greg Rose and the anonymous reviewers of FSE'09 for helpful comments. Dmitry Khovratovich is supported by PRP "Security & Trust" grant of the University of Luxembourg. Ivica Nikolić is supported by the Fonds National de la Recherche Luxembourg grant TR-PHD-BFR07-031. Ralf-Philipp Weinmann is supported by the ESS project of the University of Luxembourg.

References

1. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Sponge functions (2007), http://sponge.noekeon.org/
2. De Cannière, C., Rechberger, C.: Finding SHA-1 characteristics: General results and applications. In: Lai, X., Chen, K. (eds.) ASIACRYPT 2006. LNCS, vol. 4284, pp. 1–20. Springer, Heidelberg (2006)
3. Diffie, W., Hellman, M.E.: Exhaustive cryptanalysis of the NBS data encryption standard. Computer 10, 74–84 (1977)
4. Gligoroski, D., Ødegård, R.S., Mihova, M., Knapskog, S.J., Kocarev, L., Drápal, A.: Cryptographic hash function Edon-R. Submission to NIST (2008), http://people.item.ntnu.no/danilog/Hash/Edon-R/Supporting_Documentation/EdonRDocumentation.pdf
5. Mendel, F., Pramstaller, N., Rechberger, C., Kontak, M., Szmidt, J.: Cryptanalysis of the GOST hash function. In: Wagner, D. (ed.) CRYPTO 2008. LNCS, vol. 5157, pp. 162–178. Springer, Heidelberg (2008)
6. Morita, H., Ohta, K., Miyaguchi, S.: A switching closure test to analyze cryptosystems. In: Feigenbaum, J. (ed.) CRYPTO 1991. LNCS, vol. 576, pp. 183–193. Springer, Heidelberg (1992)
7. National Institute of Standards and Technology. Announcing Request for Candidate Algorithm Nominations for a New Cryptographic Hash Algorithm (SHA–3) Family 72(212) of Federal Register (November 2007)
8. O'Neil, S.: EnRUPT: First all-in-one symmetric cryptographic primitive. In: SASC 2008 (2008), http://www.ecrypt.eu.org/stvl/sasc2008/
9. O'Neil, S., Nohl, K., Henzen, L.: EnRUPT hash function specification (2008), http://enrupt.com/SHA3/
10. Preneel, B.: Analysis and Design of Cryptographic Hash Functions. PhD thesis, Katholieke Universiteit Leuven, Leuven, Belgium (January 1993)
11. Quisquater, J.-J., Delescaille, J.-P.: How easy is collision search? Application to DES (extended summary). In: Quisquater, J.-J., Vandewalle, J. (eds.) EUROCRYPT 1989. LNCS, vol. 434, pp. 429–434. Springer, Heidelberg (1989)
12. Quisquater, J.-J., Delescaille, J.-P.: How easy is collision search. new results and applications to DES. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 408–413. Springer, Heidelberg (1990)
13. Rose, G.G.: Design and primitive specification for Boole, http://seer-grog.net/BoolePaper.pdf
14. van Oorschot, P.C., Wiener, M.J.: Parallel collision search with application to hash functions and discrete logarithms. In: ACM Conference on Computer and Communications Security, pp. 210–218 (1994)
15. Varıcı, K., Özen, O., Kocair, Ç.: Sarmal: SHA-3 proposal. Submission to NIST (2008)
16. Wang, X., Yu, H.: How to break MD5 and other hash functions. In: Cramer, R. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 19–35. Springer, Heidelberg (2005)

A Proof of the EnRUPT Lemma 1

Let $(x_0^{\text{new}}, x_1^{\text{new}}, x_2^{\text{new}}, x_3^{\text{new}}, \ldots, x_{15}^{\text{new}}, d_0^{\text{new}}, d_1^{\text{new}})$ be our starting state. We want to invert backwards one iteration of the compression function. Hence, we want to obtain the previous state $(x_0^{\text{new}}, x_1^{\text{new}}, x_2^{\text{old}}, x_3^{\text{old}}, \dots, x_{15}^{\text{new}}, d_0^{\text{old}}, d_1^{\text{old}})$ where $x_3^{\text{old}} = 0$. From the description of ïRRUPT-512 we get:

$$x_2^{\text{new}} = f(x_1^{\text{new}}, x_6^{\text{old}}, d_0^0, r) \oplus x_2^{\text{old}} \tag{2}$$
$$x_3^{\text{new}} = \underbrace{f(x_0^{\text{new}}, x_7^{\text{old}}, d_1^1, r+1)}_{f_3} \oplus x_3^{\text{old}}, \quad d_1^3 = f_3 \oplus d_1^1 \oplus x_1^{\text{new}} \tag{3}$$
$$x_4^{\text{new}} = f(x_3^{\text{new}}, x_8^{\text{old}}, d_0^2, r+2) \oplus x_4^{\text{old}} \tag{4}$$
$$x_5^{\text{new}} = \underbrace{f(x_2^{\text{new}}, x_9^{\text{old}}, d_1^3, r+3)}_{f_5} \oplus x_5^{\text{old}}, \quad d_1^5 = f_5 \oplus d_1^3 \oplus x_3^{\text{new}} \tag{5}$$
$$x_6^{\text{new}} = f(x_5^{\text{new}}, x_{10}^{\text{new}}, d_0^4, r+4) \oplus x_6^{\text{old}} \tag{6}$$
$$x_7^{\text{new}} = \underbrace{f(x_4^{\text{new}}, x_{11}^{\text{new}}, d_1^5, r+5)}_{f_7} \oplus x_7^{\text{old}}, \quad d_1^7 = f_7 \oplus d_1^5 \oplus x_5^{\text{new}} \tag{7}$$
$$x_8^{\text{new}} = f(x_7^{\text{new}}, x_{12}^{\text{new}}, d_0^6, r+6) \oplus x_8^{\text{old}} \tag{8}$$
$$x_9^{\text{new}} = \underbrace{f(x_6^{\text{new}}, x_{13}^{\text{new}}, d_1^7, r+7)}_{f_9} \oplus x_9^{\text{old}}, \quad d_1^{\text{new}} = f_9 \oplus d_1^7 \oplus x_7^{\text{new}} \oplus p \tag{9}$$

With d_1^i we denote the value of the accumulator d_1 used in the update of the state word x_i. We need to fix x_3^{old} to zero. Hence, from (3), we get the equation:

$$\begin{aligned} x_3^{\text{new}} =& f_3 = f(x_0^{\text{new}}, x_7^{\text{old}}, d_1^1, r+1) = \\ =& 9 \cdot ((2x_0^{\text{new}} \oplus r \oplus (x_7^{\text{old}} \oplus d_1^1)) \ggg 16). \end{aligned}$$

In the upper equation we can denote by $X = x_7^{\text{old}} \oplus d_1^1$. Since, all the other variables are already known, a table can be built for this equation, and solution for X can be found. Let $C_1 = X = x_7^{\text{old}} \oplus d_1^1$. If we express the value of x_7^{old} from (7) then we get the following equation:

$$x_7^{\text{new}} \oplus f_7 \oplus d_1^1 = C_1. \tag{10}$$

Further, from (3), (5), (7), and (9), this equation can be rewritten as:

$$x_7^{\text{new}} \oplus f_7 \oplus f_3 \oplus f_5 \oplus f_7 \oplus f_9 \oplus x_1^{\text{new}} \oplus x_3^{\text{new}} \oplus x_5^{\text{new}} \oplus x_7^{\text{new}} \oplus p = C_1.$$

Since, $x_1^{\text{new}}, x_3^{\text{new}}, x_5^{\text{new}}, x_7^{\text{new}}$, and f_3 are all constant (the value of f_3 is equal to x_3^{new}), the upper equation can be rewritten as:

$$f_5 + f_9 + p = K, \tag{11}$$

where $K = x_3^{\text{new}} \oplus x_5^{\text{new}} \oplus f_3 \oplus C_1$. So given the values of f_5 and f_9 from (11) we can easily find the value for the message word p such that $x_3^{\text{old}} = 0$ holds. Let us try to find the values of f_5 and f_9.

The value of f_5 (from (5)) depends, in particular, on x_9^{old} and d_1^3. From (9) we get that $x_9^{\text{old}} = f_9 \oplus x_9^{\text{new}}$. From (3) and (10) we get:

$$d_1^3 = f_3 \oplus x_1^{\text{new}} \oplus d_1^1 = f_3 \oplus x_1^{\text{new}} \oplus x_7^{\text{new}} \oplus f_7 \oplus C_1. \tag{12}$$

Therefore, for the value of f_5 we get:

$$f_5 = 9 \cdot ((2x_2^{\text{new}} \oplus (r+3) \oplus x_9^{\text{old}} \oplus d_1^3) \ggg 16) = \\ = 9 \cdot ((K_1 \oplus f_7 \oplus f_9) \ggg 16), \quad (13)$$

where $K_1 = 2x_2^{\text{new}} \oplus (r+3) \oplus x_9^{\text{new}} \oplus f_3 \oplus x_1^{\text{new}} \oplus x_7^{\text{new}} \oplus C_1$.

Similarly, for f_7 from (7), we can see that depends on d_1^5. For this variable, from (12) and (5), we get:

$$d_1^5 = f_5 \oplus x_3^{\text{new}} \oplus d_1^3 = f_5 \oplus x_3^{\text{new}} \oplus f_3 \oplus x_1^{\text{new}} \oplus x_7^{\text{new}} \oplus f_7 \oplus C_1. \quad (14)$$

Hence, for f_7 we get:

$$f_7 = 9 \cdot ((2x_4^{\text{new}} \oplus (r+5) \oplus x_{11}^{\text{new}} \oplus d_1^5) \ggg 16) = \\ = 9 \cdot ((K_2 \oplus f_5 \oplus f_7) \ggg 16), \quad (15)$$

where $K_2 = 2x_4^{\text{new}} \oplus (r+5) \oplus x_{11}^{\text{new}} \oplus x_3^{\text{new}} \oplus f_3 \oplus x_1^{\text{new}} \oplus x_7^{\text{new}}$.

Finally, for f_9 from (7)), we get that it depends on d_1^7. From (14) and (7), for the value of d_1^7 we get the following:

$$d_1^7 = f_7 \oplus x_5^{\text{new}} \oplus d_1^5 = \\ = f_7 \oplus x_5^{\text{new}} \oplus f_5 \oplus x_3^{\text{new}} \oplus f_3 \oplus x_1^{\text{new}} \oplus x_7^{\text{new}} \oplus f_7 \oplus C_1 = \\ = x_5^{\text{new}} \oplus f_5 \oplus x_3^{\text{new}} \oplus f_3 \oplus x_1^{\text{new}} \oplus x_7^{\text{new}} \oplus C_1.$$

For the value of f_9 we get:

$$f_9 = 9 \cdot ((2x_6^{\text{new}} \oplus (r+7) \oplus x_{13}^{\text{new}} \oplus d_1^7) \ggg 16) = 9 \cdot ((K_3 \oplus f_5) \ggg 16), \quad (16)$$

where $K_3 = 2x_6^{\text{new}} \oplus (r+7) \oplus x_{13}^{\text{new}} \oplus x_5^{\text{new}} \oplus x_3^{\text{new}} \oplus f_3 \oplus x_1^{\text{new}} \oplus x_7^{\text{new}} \oplus C_1$.

As a result, we get a system of three equations ((13),(15), and (16)) with three unknowns f_5, f_7, and f_9:

$$\begin{cases} f_5 = 9 \cdot ((K_1 \oplus f_7 \oplus f_9) \ggg 16); \\ f_7 = 9 \cdot ((K_2 \oplus f_5 \oplus f_7) \ggg 16); \\ f_9 = 9 \cdot ((K_3 \oplus f_5) \ggg 16). \end{cases}$$

We can build a table that solves this system. There are six columns in the table: three unknowns and three constants: K_1, K_2, and K_3.

After we find the exact values of f_5 and f_9 we can easily compute the value of p from (11).