Cryptanalysis of EnRUPT

Dmitry Khovratovich and Ivica Nikolić

University of Luxembourg
5 November 2008

Abstract. In this paper we present a preimage attack on EnRUPT-512. We exploit the fact that the internal state is only a little bit larger than the critical security level: 1152 bits against 1024 bits. The absence of a message expansion and a fairly simple compression function allow us to fix the values for some state words and thus reduce the size of birthday state space in the meet-in-the-middle attack under 1024 bits. Equations that arise through the analysis are solved using look-up tables. The complexity of the attack is around 2^{480} compression function calls and the memory requirement is roughly the same.

1 Introduction

The family of hash functions EnRUPT [10] was designed by Sean O'Neil, Karsten Nohl, and Luca Henzen and was submitted to the SHA-3 hash function competition [8]. EnRUPT is itself a member of a set of cryptographic primitives first presented by O'Neil in SASC 2008 [9].

The EnRUPT hash functions work in so-called stream hashing mode following other stream-based hash functions: RadioGatun [1], Panama [3], Grindahl [6], LUX [7]. These hash functions have rather simple compression function with no message expansion, which allows to exploit freedom given by the message injection and thus to maintain various attacks [11, 2, 4, 5].

The EnRUPT specification defines 7 basic hash functions with respect to the size of produced hash value: 128–512 bit. In this paper we present a preimage attack on ïrRUPT-512, EnRUPT with a 512-bit digest. We also expect that our attack can be carried out to the other versions of EnRUPT.

The paper is organized as follows. First we recall the definition of ïrRUPT-512. Then we notice that independency of message blocks inserted may be used to reduce the state space thus allowing a shortcut meet-in-the-middle attack. We show that the attack can be maintained due to simplicity and linearity (in some sense) of the compression function. We conclude with a short list of weaknesses that let the attack be performed.

2 Definition of ïrRUPT-512

The ïrRUPT-512 hash function deals with a message (appropriately padded) divided into 64-bit blocks. A message block p is an input to the compression function ïr8, which is a transformation of a 1152-bit internal state. The internal state H consists of sixteen 64-bit words $x_0, x_1, \ldots, x_{15}$ and two 64-bit words d_0, d_1.

Before the compression function starts all the internal state words are initialized with zero. A round counter r is set to zero as well. The compression function consists of 8 iterations of the subfunction ïr1, each updating one of x_i and one of d_i.

```
irRUPT-512:

INPUT: message blocks p_0, p_1, ... p_n
    x_j  = 0, d_j =0 for all j;
    r=0;
    for i=0 to n do //Squeezing
        ir8(p_i);

    for i=0 to 199 //Blank rounds
        ir8(0);

    for i=0 to 7 //Output
        ir8(0)
        OUTPUT(d_1)

ir8:
INPUT: message block p
    for k=0 to 7 do
        ir1();
        r++;
    d_1 ^= p;

ir1:
    x_{r+2} ^= 9*((2*x_{r^1} ^ x_{r+4} ^ d_{r&1} ^ r)>>>16);
    d_{r&1} ^= 9*((2*x_{r^1} ^ x_{r+4} ^ d_{r&1} ^ r)>>>16) ^ x_r;
```

In the pseudo-code all indices are taken modulo 16, all multiplications are performed modulo 2^{64}. Here also $\ggg$ stands for cyclic rotation, ^ stands for XOR.

3 Preimage Attack on ïrRUPT-512

3.1 Basic Definitions and Observations

Equation invertibility. The accumulators d_i are updated by the non-invertible function, which can be expressed in form $x \oplus g(x \oplus y)$ (see pseudocode). 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, and they 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 actively use look-up tables in order to find a solution for the equations arising from 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 the 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

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

where C_i are constants. We precompute for all possible x, y, z, C_i (2^{384} lines) 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 it so that it is easy to find a solution (or many) given C_i.

We can 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.

State space. A state space S of a hash function is a set of all possible values of the internal state of the function. It is assumed that if the internal state has s bits then the state space has 2^s elements. For ïrRUPT-512 the state space can be described as a set of all possible vectors with $16 * 64 + 2 * 64 = 1152$ coordinates in F_2.

Meet-in-the-middle attack. The meet-in-the-middle approach is often used to find preimages for hash functions of a specific structure. The attack works as follows. Let the state space S of the hash function has s bits. The attacker, starting from the initial value builds a set $S_1 \in S$ of $2^{s/2}$ different intermediate hash values. Then, the attacker, starting from the final hash value, and going backwards, builds another set $S_2 \in S$ of $2^{s/2}$ different intermediate hash values. Then with overwhelming probability these two sets have at least one equal element. The two obvious basic requirements, for this attack to work, are:

1. The hash function is invertible
2. The state space has s bits, where $s < 2n$.

At first sight, none of these requirements are satisfied for ïrRUPT-512. In the compression function, the accumulators d_0 and d_1 are updated non-bijectively. Hence, the hash function is not invertible. The state space S of ïrRUPT-512 has 1152 which is greater than $2 * n = 1024$ and therefore the second requirement also doesn't hold. Further in the attack, we will describe how to overcome these two bottlenecks.

3.2 Inverting 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 table look-up (see above). Since 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.

3.3 Reducing the State Space

The state space of ÏRRUPTconsists of the state words $x_0, x_1, \ldots, x_{15}$, and the accumulators d_0 and d_1 (the round index r doesn't play important role in the attack when the message length is predefined, as in our case). Hence this space has 1152 bits. But, if all the elements of S_1 and S_2 at certain bits have the same values than the complexity of the meet-in-the-middle attack can be decreased. If m bits in S_1 and S_2 are fixed then the attack has a complexity $2^{\frac{1152-m}{2}}$. Further, we will show how to fix x_3, x_{11} and d_1.

ÏRRUPT-512 in one iteration of the compression function updates only one half of the state words: either $(x_2, x_3, \ldots, x_9)$ or $(x_{10}, x_{11}, \ldots, x_1)$. Therefore, it is possible to control the value of exactly one word of these halves, in each compression function iteration, through the input message block p_i. If $(x_2, x_3, \ldots, x_9)$ half is updated, then we call this iteration even, otherwise if $(x_{10}, x_{11}, \ldots, x_1)$ is updated then it is odd iteration. In meet-in-the-middle scenario, let us analyze how to fix x_3 and x_{11} to zero in both directions: starting from the IV and going forward and starting from the target hash value and going backwards.

Fixing x_3 and x_{11}: forward. Assume that after even number of iterations we have an internal state with any values of the state words and accumulators. By 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. For building this table, we spend $(2^{64})^4 = 2^{256}$ memory and effort[1]. Notice that after the value of x_3 is fixed then, in the odd iteration that follows, this value is not changed. In this odd iteration, we fix the value of x_{11} using exactly the same method. Hence, in two sequential rounds, even and odd, we can fix the value of exactly two state words: x_3 and x_{11}.

Fixing x_3 and x_{11}: backwards. Going backwards in the compression function is more complex when we want to set the values of x_3 and x_{11} because they are updated first in the iteration, which means last in the inverted iteration. We will explain how the value of x_3 can be set to zero in an even round. The same can be applied to x_{11} in an odd round.

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

$$x_2^{\text{new}} = f(x_1^{\text{new}}, x_6^{\text{old}}, d_0^0, r) \oplus x_2^{\text{old}} \tag{1}$$

$$x_3^{\text{new}} = \underbrace{f(x_0^{\text{new}}, x_7^{\text{old}}, d_1^1, r+1)}_{f_3} \oplus x_3^{\text{old}}, \quad d_1^3 = f_3 \oplus d_1^1 \oplus x_1^{\text{new}} \tag{2}$$

$$x_4^{\text{new}} = f(x_3^{\text{new}}, x_8^{\text{old}}, d_0^2, r+2) \oplus x_4^{\text{old}} \tag{3}$$

$$x_5^{\text{new}} = \underbrace{f(x_2^{\text{new}}, x_9^{\text{old}}, d_1^3, r+3)}_{f_5} \oplus x_5^{\text{old}}, \quad d_1^5 = f_5 \oplus d_1^3 \oplus x_3^{\text{new}} \tag{4}$$

$$x_6^{\text{new}} = f(x_5^{\text{new}}, x_{10}^{\text{new}}, d_0^4, r+4) \oplus x_6^{\text{old}} \tag{5}$$

$$x_7^{\text{new}} = \underbrace{f(x_4^{\text{new}}, x_{11}^{\text{new}}, d_1^5, r+5)}_{f_7} \oplus x_7^{\text{old}}, \quad d_1^7 = f_7 \oplus d_1^5 \oplus x_5^{\text{new}} \tag{6}$$

$$x_8^{\text{new}} = f(x_7^{\text{new}}, x_{12}^{\text{new}}, d_0^6, r+6) \oplus x_8^{\text{old}} \tag{7}$$

$$x_9^{\text{new}} = \underbrace{f(x_6^{\text{new}}, x_{13}^{\text{new}}, d_1^7, r+7)}_{f_9} \oplus x_9^{\text{old}}, \quad d_1^{\text{new}} = f_9 \oplus d_1^7 \oplus x_7^{\text{new}} \oplus p \tag{8}$$

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 (2), 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

[1] Another way is to solve the equation directly, which can be done almost for free but requires a bit more explanation.

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 (6) then we get the following equation:

$$x_7^{\text{new}} \oplus f_7 \oplus d_1^1 = C_1. \tag{9}$$

Further, from (2), (4), (6), and (8), 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:

$$f_5 + f_9 + p = K, \tag{10}$$

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 (10) 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 (4)) depends, in particular, on x_9^{old} and d_1^3. From (8) we get that $x_9^{\text{old}} = f_9 \oplus x_9^{\text{new}}$. From (2) and (9) we get:

$$d_1^3 = f_3 \oplus x_1^{\text{new}} \oplus d_1^1 = f_3 \oplus x_1^{\text{new}} \oplus x_7^{\text{new}} \oplus f_7 \oplus C_1. \tag{11}$$

Therefore, for the value of f_5 we get:

$$\begin{aligned} f_5 = 9 \cdot ((2x_2^{\text{new}} \oplus (r+3) \oplus x_9^{\text{old}} \oplus d_1^3) \ggg 16) = \\ = 9 \cdot ((K_1 \oplus f_7 \oplus f_9) \ggg 16), \end{aligned} \tag{12}$$

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 (6), we can see that depends on d_1^5. For this variable, from (11) and (4), we get:

$$d_1^5 = f_5 \oplus x_3^{\text{new}} \oplus d_1^3 = f_5 \oplus x_3^{\text{new}} \oplus f_3 \oplus x_1^{\text{new}} \oplus x_7^{\text{new}} \oplus f_7 \oplus C_1. \tag{13}$$

Hence, for f_7 we get:

$$\begin{aligned} f_7 = 9 \cdot ((2x_4^{\text{new}} \oplus (r+5) \oplus x_{11}^{\text{new}} \oplus d_1^5) \ggg 16) = \\ = 9 \cdot ((K_2 \oplus f_5 \oplus f_7) \ggg 16), \end{aligned} \tag{14}$$

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 (6)), we get that it depends on d_1^7. From (13) and (6), for the value of d_1^7 we get the following:

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

For the value of f_9 we get:

$$f_9 = 9 \cdot ((2x_6^{\text{new}} \oplus (r+7) \oplus x_{13}^{\text{new}} \oplus d_1^7) \ggg 16) = 9 \cdot ((K_3 \oplus f_5) \ggg 16), \tag{15}$$

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 ((12),(14), and (15)) 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 (10).

3.4 Meet-in-the-middle attack on ïrRUPT-512

Now that we explained how the compression function can be inverted and the state space can be reduced let us try to launch the attack itself. Notice that we have reduced the attack space only by two message words, x_3 and x_{11}, though we claimed that we can also fix d_1.

The final phase of the attack is outlined in Figure 1. Further we will explain how this is done.

Meet-in-the-middle: starting from the IV. Starting from the initial value, by changing the input message words, we create 2^{480} different internal states $S_i, i = 1, \ldots, 2^{480}$. This can be done in 8 iterations, hence the input message words for an internal state S_i can be denoted as $(p_{-7}^i, p_{-6}^i, \ldots, p_0^i)$. Then, for each state, in one iteration we fix the value of x_3 to zero, by changing the previous input message word p_0. In the second iteration, we fix the value of x_{11} by setting the value of p_1. As a result, we have obtained a set S_1 of 2^{480} different states that have zero values in x_3 and x_{11}. Notice we haven't fixed the value of p_2 yet.

Meet-in-the-middle: backwards from the hash value. 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 small chunk of 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 $H = (d_1^t, d_1^{t+1}, \ldots, d_1^{t+7})$. We take an empty state, and set a value of $d_1 = d_1^t$. Then, we take 2^{448} different values for the rest of the state and iterate forward for 7 rounds, while producing an output at each round. With overwhelming probability, one of these outputs will coincide with

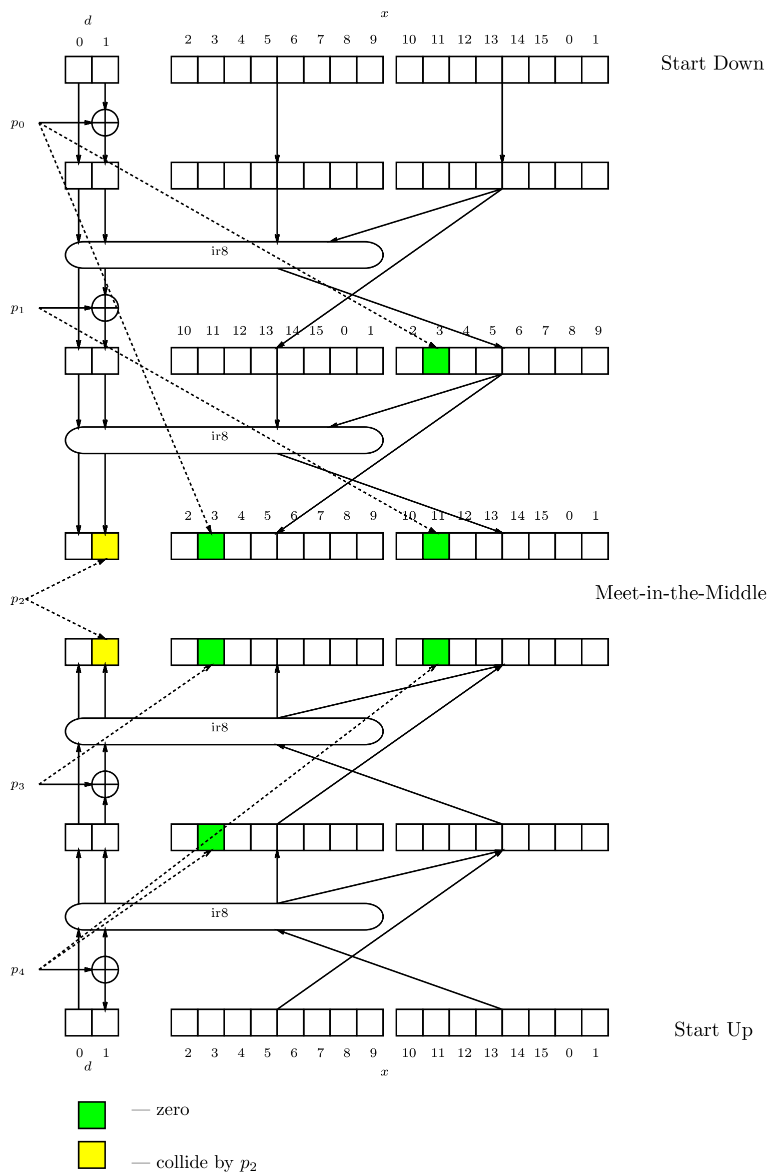

Fig. 1. Outline of the attack.

$(d_1^{t+1}, \ldots, d_1^{t+7})$. 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 start building the set of different internal states. We go backwards 8 rounds, and by injecting different message words we produce 2^{480} different internal states. Then, in two backward iteration we fix the values for x_3 and x_{11} as explained before. As a result we obtain 2^{480} a set S_2 of different states with x_3 and x_{11} set to zero.

We have two sets S_1 and S_2 each of 2^{480} elements. With high probability there are two element, $s_1 \in S_1$ and $s_2 \in S_2$, in each of these sets that have the same values for $x_0, x_1, x_2, x_4, \ldots, x_8, x_{10}, \ldots, x_{15}, d_0$ (in total 960 bits). But the values of x_3 and x_{11} also coincide because we set them to zero for all elements in the both sets. Hence, the only difference between s_1 and s_2 can be in their $d_1^{s_1}$ and $d_1^{s_2}$. Therefore we change the last message input block p_9 for s_1 and set it to $p_9 = d_1^{s_1} \oplus d_1^{s_2}$. The message addition is at the end of the compression function, so no other values, besides $d_1^{s_1}$, will change. As a result, we obtained the same internal state, by going forward from the initial values and backwards from the target hash value, and therefore a preimage of the target hash value.

3.5 Complexity of the Attack

We spend at most 2^{384} effort to build pre-computation tables so it is not a bottleneck. To compose two valid state after blank rounds that give the desired hash we need about 2^{448} trials. We also pass blank rounds for free since the absence of solutions for some states is compensated by many of them for other ones.

Thus the most time-consuming part is the preparation of two sets for the meet-in-the-middle, each of 2^{480} states. However, each state is produced with negligible complexity since we use pre-computation tables to find a suitable message block to be injected. Thus we estimate the time and memory complexity of our attack as about 2^{480} simple hash queries, which is smaller than a brute-force attack (2^{512}).

4 Conclusions

We presented a preimage attack on EnRUPT-512. Though being a shortcut attack only (it requires about 2^{480} operations), it points out a weak structure of EnRUPT. Let us detail the weaknesses that we discovered.

First, there is no message scheduling in EnRUPT. While this is not a weakness in itself (RadioGatun is a counterexample), this usually requires a bigger state in order to compensate actual freedom given by the message injection. The EnRUPT internal state is only 128 bits larger than the hash digest. Thus exploiting only three message injections ($3 * 64 = 192$) might provide a successful attack, which was demonstrated in our paper.

Secondly, message injection affects only a part of the internal state. Giving more freedom from message injections this fact was highly exploited not only in this paper but also in previous attacks on hash functions with similar design [5, 11]. Again, this might be counteracted by expanding the internal state.

Thirdly, the internal round function (ïr8) as well as its inversion allows easy manipulation with its output. This significantly simplified the cryptanalysis and allowed us to solve many arising equations with precomputations.

References

1. Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. Radiogatun, a belt-and-mill hash function, 2006, available at http://radiogatun.noekeon.org/.
2. Joan Daemen and Gilles Van Assche. Producing collisions for Panama, instantaneously. In *FSE'07*, volume 4593 of *LNCS*, pages 1–18. Springer, 2007.
3. Joan Daemen and Craig S. K. Clapp. Fast hashing and stream encryption with PANAMA. In *FSE'98*, volume 1372 of *LNCS*, pages 60–74. Springer, 1998.
4. Dmitry Khovratovich. Cryptanalysis of hash functions with structures. In *Hash Function Workshop, Leiden*, 2008, 2–6 June, slides also available at http://www.lorentzcenter.nl/lc/web/2008/309/presentations/Khovratovich.pdf .
5. Dmitry Khovratovich. Two attacks on Radiogatun. In *Indocrypt*, 2008, to appear.
6. Lars R. Knudsen, Christian Rechberger, and Søren S. Thomsen. The Grindahl hash functions. In *FSE'07*, volume 4593 of *LNCS*, pages 39–57. Springer, 2007.
7. Ivica Nikolić, Alex Biryukov, and Dmitry Khovratovich. Hash function family LUX, 2008, submitted to the SHA-3 competition.
8. National Institute of Standards and Technology. *Announcing Request for Candidate Algorithm Nominations for a New Cryptographic Hash Algorithm (SHA3) Family*, volume 72, No. 212 of *Federal Register*. November 2007.
9. Sean O'Neil. EnRUPT: First all-in-one symmetric cryptographic primitive. SASC 2008, avalaible at http://www.ecrypt.eu.org/stvl/sasc2008/.
10. Sean O'Neil, Karsten Nohl, and Luca Henzen. EnRUPT hash function specification, 2008, available at http://enrupt.com/SHA3/.
11. Thomas Peyrin. Cryptanalysis of Grindahl. In *ASIACRYPT'07*, volume 4833 of *LNCS*, pages 551–567. Springer, 2007.