A (Second) Preimage Attack on the GOST Hash Function

Florian Mendel, Norbert Pramstaller, and Christian Rechberger

Institute for Applied Information Processing and Communications (IAIK), Graz University of Technology, Inffeldgasse 16a, A-8010 Graz, Austria
Florian.Mendel@iaik.TUGraz.at

Abstract. In this article, we analyze the security of the GOST hash function with respect to (second) preimage resistance. The GOST hash function, defined in the Russian standard GOST-R 34.11-94, is an iterated hash function producing a 256-bit hash value. As opposed to most commonly used hash functions such as MD5 and SHA-1, the GOST hash function defines, in addition to the common iterated structure, a checksum computed over all input message blocks. This checksum is then part of the final hash value computation. For this hash function, we show how to construct second preimages and preimages with a complexity of about 2^{225} compression function evaluations and a memory requirement of about 2^{38} bytes.
First, we show how to construct a pseudo-preimage for the compression function of GOST based on its structural properties. Second, this pseudo-preimage attack on the compression function is extended to a (second) preimage attack on the GOST hash function. The extension is possible by combining a multicollision attack and a meet-in-the-middle attack on the checksum.

Keywords: cryptanalysis, hash functions, preimage attack

1 Introduction

A cryptographic hash function H maps a message M of arbitrary length to a fixed-length hash value h. A cryptographic hash function has to fulfill the following security requirements:

- *Collision resistance:* it is practically infeasible to find two messages M and M^*, with $M^* \neq M$, such that $H(M) = H(M^*)$.
- *Second preimage resistance:* for a given message M, it is practically infeasible to find a second message $M^* \neq M$ such that $H(M) = H(M^*)$.
- *Preimage resistance:* for a given hash value h, it is practically infeasible to find a message M such that $H(M) = h$.

The resistance of a hash function to collision and (second) preimage attacks depends in the first place on the length n of the hash value. Regardless of how a hash function is designed, an adversary will always be able to find preimages or second preimages after trying out about 2^n different messages. Finding collisions requires a much smaller number of trials: about $2^{n/2}$ due to the birthday paradox. If the internal structure of a particular hash function allows collisions or (second) preimages to be found more efficiently than what could be expected based on its hash length, then the function is considered to be broken.

Recent cryptanalytic results on hash functions mainly focus on collision attacks (see for instance [2,3,14,15,16,17]) but only few results with respect to (second) preimages have been published to date (see for instance [7,9]). In this article, we will present a security analysis with respect to (second) preimage resistance for the hash function specified in the Russian national standard GOST-R 34.11-94. This standard has been developed by *GUBS of Federal Agency Government Communication and Information* and *All-Russian Scientific and Research Institute of Standardization*. The standard also specifies amongst others the GOST block cipher and the GOST signature algorithm.

The GOST hash function is an iterated hash function producing a 256-bit hash value. Since the GOST block cipher is a building block of the hash function, it can be considered as a block-cipher-based hash function. While there have been published several cryptanalytic results regarding the block cipher (see for instance [1,6,8,12,13]), we are not aware of any published security analysis of the GOST hash function besides the work of Gauravaram and Kelsey in [4]. They show that the generic attacks on hash functions based on the Damgård-Merkle design principle can be extended to hash functions with linear/modular checksums independent of the underlying compression function.

In this article, we present an analysis of the GOST hash function. To denote the GOST hash function, we will simply write GOST for the remainder of this article. We exploit the internal structure of GOST to construct pseudo-preimages for the compression function of GOST with a complexity of about 2^{192}. Furthermore, we show how this attack on the compression function of GOST can be extended to a (second) preimage attack on the hash function. The attack has a complexity of about 2^{225} instead of 2^{256} what is expected from a 256-bit hash value. Both attacks are structural attacks in the sense that they are independent of the underlying block cipher.

The remainder of this article is structured as follows. In Section 2, we give a short description of GOST. Section 3 presents the pseudo-preimage attack on the compression function. The extension of the attack on the compression function resulting in the preimage attack is discussed in Section 4. Finally, we present conclusions in Section 5.

2 Description of GOST

GOST is an iterated hash function that processes message blocks of 256 bits and produces a 256-bit hash value. If the message length is not a multiple of 256, an unambiguous padding method is applied. For the description of the padding method we refer to [10]. Let $M = M_1 \| M_2 \| \cdots \| M_t$ be a t-block message (after

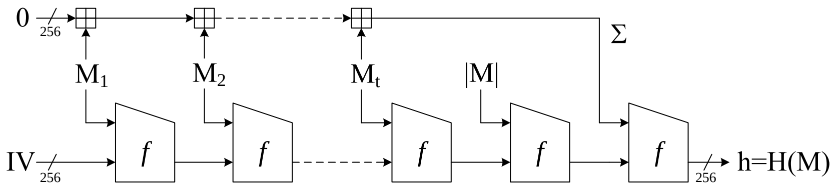

Fig. 1. Structure of the GOST hash function.

padding). The hash value $h = H(M)$ is computed as follows (see Fig. 1):

$$H_0 = IV \quad (1)$$
$$H_i = f(H_{i-1}, M_i) \quad \text{for } 0 < i \leq t \quad (2)$$
$$H_{t+1} = f(H_t, |M|) \quad (3)$$
$$H_{t+2} = f(h_{t+1}, \Sigma) = h \ , \quad (4)$$

where $\Sigma = M_1 \boxplus M_2 \boxplus \cdots \boxplus M_t$, and $\boxplus$ denotes addition modulo 2^{256}. IV is a predefined initial value and $|M|$ represents the bit-length of the entire message prior to padding. As can be seen in (4), GOST specifies a checksum (Σ) consisting of the modular addition of all message blocks, which is then input to the final application of the compression function. Computing this checksum is not part of most commonly used hash functions such as MD5 and SHA-1.

The compression function f of GOST basically consist of three parts (see also Fig. 2): the state update transformation, the key generation, and the output transformation. In the following, we will describe these parts in more detail.

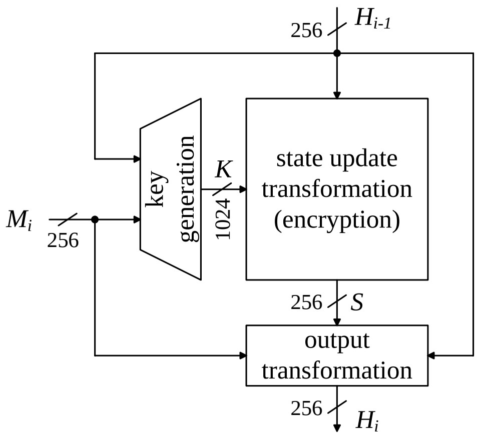

Fig. 2. The compression function of GOST

2.1 State Update Transformation

The state update transformation of GOST consists of 4 parallel instances of the GOST block cipher, denoted by E. The intermediate hash value H_{i-1} is split into four 64-bit words $h_3\|h_2\|h_1\|h_0$. Each 64-bit word is used in one stream of the state update transformation to construct the 256-bit value $S = s_3\|s_2\|s_1\|s_0$ in the following way:

$$s_0 = E(k_0, h_0) \tag{5}$$
$$s_1 = E(k_1, h_1) \tag{6}$$
$$s_2 = E(k_2, h_2) \tag{7}$$
$$s_3 = E(k_3, h_3) \tag{8}$$

where $E(k, p)$ denotes the encryption of the 64-bit plaintext p under the 256-bit key k. We refer to the GOST standard, for a detailed description of the GOST block cipher.

2.2 Key Generation

The key generation of GOST takes as input the intermediate hash value H_{i-1} and the message block M_i to compute a 1024-bit key K. This key is split into four 256-bit keys k_i, *i.e.* $K = k_3\| \cdots \|k_0$, where each key k_i is used in one stream as the key for the GOST block cipher E in the state update transformation. The four keys k_0, k_1, k_2, and k_3 are computed in the following way:

$$k_0 = P(H_{i-1} \oplus M_i) \tag{9}$$
$$k_1 = P(A(H_{i-1}) \oplus A^2(M_i)) \tag{10}$$
$$k_2 = P(A^2(H_{i-1}) \oplus \texttt{Const} \oplus A^4(M_i)) \tag{11}$$
$$k_3 = P(A(A^2(H_{i-1}) \oplus \texttt{Const}) \oplus A^6(M_i)) \tag{12}$$

where A and P are linear transformations and `Const` is a constant. Note that $A^2(x) = A(A(x))$. For the definition of the linear transformation A and P as well as the value of `Const`, we refer to [10], since we do not need them for our analysis.

2.3 Output Transformation

The output transformation of GOST combines the initial value H_{i-1}, the message block M_i, and the output of the state update transformation S to compute the output value H_i of the compression function. It is defined as follows.

$$H_i = \psi^{61}(H_{i-1} \oplus \psi(M_i \oplus \psi^{12}(S))) \tag{13}$$

The linear transformation $\psi : \{0,1\}^{256} \to \{0,1\}^{256}$ is given by:

$$\psi(\Gamma) = (\gamma_0 \oplus \gamma_1 \oplus \gamma_2 \oplus \gamma_3 \oplus \gamma_{12} \oplus \gamma_{15})\|\gamma_{15}\|\gamma_{14}\| \cdots \|\gamma_1 \tag{14}$$

where Γ is split into sixteen 16-bit words, *i.e.* $\Gamma = \gamma_{15}\|\gamma_{14}\| \cdots \|\gamma_0$.

3 Constructing Pseudo-Preimages for the Compression Function of GOST

In this section, we present how to construct a pseudo-preimage for the compression function of GOST. The attack is based on structural weaknesses of the compression function. Since the transformation ψ is linear, (13) can be written as:

$$H_i = \psi^{61}(H_{i-1}) \oplus \psi^{62}(M_i) \oplus \psi^{74}(S) \tag{15}$$

Furthermore, ψ is invertible and hence (15) can be written as:

$$\underbrace{\psi^{-74}(H_i)}_{X} = \underbrace{\psi^{-13}(H_{i-1})}_{Y} \oplus \underbrace{\psi^{-12}(M_i)}_{Z} \oplus S \tag{16}$$

Note that Y depends linearly on H_{i-1} and Z depends linearly on M_i. As opposed to Y and Z, S depends on both H_{i-1} and M_i processed by the block cipher E. For the following discussion, we split the 256-bit words X, Y, Z defined in (16) into 64-bit words:

$$X = x_3\|x_2\|x_1\|x_0 \quad Y = y_3\|y_2\|y_1\|y_0 \quad Z = z_3\|z_2\|z_1\|z_0$$

Now, (16) can be written as:

$$x_0 = y_0 \oplus z_0 \oplus s_0 \tag{17}$$
$$x_1 = y_1 \oplus z_1 \oplus s_1 \tag{18}$$
$$x_2 = y_2 \oplus z_2 \oplus s_2 \tag{19}$$
$$x_3 = y_3 \oplus z_3 \oplus s_3 \tag{20}$$

For a given H_i, we can easily compute the value $X = \psi^{-74}(H_i)$. Now assume, that for the given $X = x_0\|x_1\|x_2\|x_3$, we can find two pairs (H^1_{i-1}, M^1_i) and (H^2_{i-1}, M^2_i), where $H^1_{i-1} \neq H^2_{i-1}$ or $M^1_i \neq M^2_i$, such that both pairs produce the value x_0. Then we know that with a probability of 2^{-192}, these two pairs also lead to the corresponding values x_1, x_2, and x_3. In other words, we have constructed a pseudo-preimage for the given H_i for the compression function of GOST with a probability of 2^{-192}. Therefore, assuming that we can construct 2^{192} pairs (H^j_{i-1}, M^j_i), where $H^j_{i-1} \neq H^k_{i-1}$ or $M^j_i \neq M^k_i$, such that all produce the value x_0, then we have constructed a pseudo-preimage for the compression function.

Based on this short description, we will show now how to construct pseudo-preimages for the compression function of GOST. We will first derive how to construct pairs (H^j_{i-1}, M^j_i), which all produce the same value x_0. This boils down to solving an underdetermined system of equations. Assume, we want to keep the value s_0 in (17) constant. Since $s_0 = E(k_0, h_0)$, we have to find pairs (H^j_{i-1}, M^j_i) such that the values k_0 and h_0 are the same for each pair. We know that h_0 directly depends on H_{i-1}. The key k_0 depends on $H_{i-1} \oplus M_i$. Therefore, we get the following equations:

$$h_0 = a \tag{21}$$

$$m_0 \oplus h_0 = b_0 \tag{22}$$

$$m_1 \oplus h_1 = b_1 \tag{23}$$

$$m_2 \oplus h_2 = b_2 \tag{24}$$

$$m_3 \oplus h_3 = b_3 \tag{25}$$

where a and the b_i's are arbitrary 64-bit values. Note that $k_0 = P(H_{i-1} \oplus M_i) = \bar{B}$, where $\bar{B} = P(B)$ and $B = b_3 \| \cdots \| b_0$, see (9). This is an underdetermined system of equations with $5 \cdot 64$ equations in $8 \cdot 64$ variables over $GF(2)$. Solving this system leads to 2^{192} solutions for which s_0 has the same value. To find pairs (H_{i-1}^j, M_i^j) for which x_0 has the same value, we still have to ensure that also the term $y_0 \oplus z_0$ in (17) has the same value for all pairs. This adds one additional equation (64 equations over $GF(2)$) to our system of equations, namely

$$y_0 \oplus z_0 = c \tag{26}$$

where c is an arbitrary 64-bit value. This equation does not add any new variables, since we know that y_0 depends linearly on $h_3\|h_2\|h_1\|h_0$ and z_0 depends linearly on $m_3\|m_2\|m_1\|m_0$, see (16). To summarize, fixing the value of x_0 boils down to solving an underdetermined equation system with $6 \cdot 64$ equations and $8 \cdot 64$ unknowns over $GF(2)$. This leads to 2^{128} solutions h_i and m_i for $0 \leq i < 4$ and hence 2^{128} pairs (H_{i-1}^j, M_i^j) for which the value x_0 is the same.

Now we can describe how the pseudo-preimage attack on the compression function of GOST works. In the attack, we have to find H_{i-1} and M_i such that $f(H_{i-1}, M_i) = H_i$ for a given value of H_i. The attack can be summarized as follows.

1. Choose random values for b_0, b_1, b_2, b_3 and a. This determines k_0 and h_0
2. Compute $s_0 = E(k_0, h_0)$ and adjust c accordingly such that

   $$x_0 = y_0 \oplus z_0 \oplus s_0 = c \oplus s_0$$

   holds with $X = \psi^{-74}(H_i)$.
3. Solve the set of $6 \cdot 64$ linear equations over $GF(2)$ to obtain 2^{128} pairs (H_{i-1}^j, M_i^j) for which x_0 is correct.
4. For each pair compute X and check if x_3, x_2 and x_1 are correct. This holds with a probability of 2^{-192}. Thus, after testing all 2^{128} pairs, we will find a correct pair with a probability of $2^{-192} \cdot 2^{128} = 2^{-64}$. Therefore, we have to repeat the attack about 2^{64} times for different choices of b_0, b_1, b_2, b_3 and a to find a pseudo-preimage for the compression function of GOST.

Hence, we can construct a pseudo-preimage for the compression function of GOST with a complexity of about 2^{192} instead of 2^{256} as expected for a compression function with an output size of 256 bits. In the next section, we will show how this attack on the compression function can be extended to a preimage attack on the hash function.

4 A Preimage Attack for the Hash Function

In a preimage attack, we want to find, for a given hash value h, a message M such that $H(M) = h$. As we will show in the following, for GOST we can construct preimages of h with a complexity of about 2^{225} evaluations of the compression function of GOST. Furthermore, the preimage consists of 257 message blocks, *i.e.* $M = M_1 \| \cdots \| M_{257}$. The preimage attack consists basically of four steps as also shown in Figure 3.

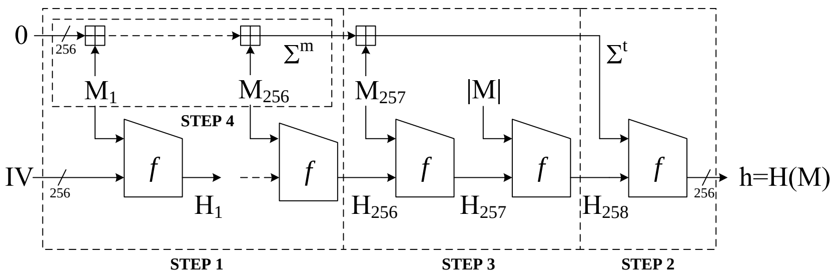

Fig. 3. Preimage Attack on GOST.

4.1 STEP 1: Multicollisions for GOST

In [5], Joux introduced multicollisions which can be constructed for any iterated hash function. A multicollision is a set of messages of equal length that all lead to the same hash value. As shown by Joux, constructing a 2^t multicollision, *i.e.* 2^t messages consisting of t message blocks which all lead to the same hash value, can be done with a complexity of about $t \cdot 2^{n/2}$, where n is the bit-size of the hash value. For the preimage attack on GOST, we construct a 2^{256} multicollision. This means, we have 2^{256} messages $M^* = M_1^{j_1} \| M_2^{j_2} \| \cdots \| M_{256}^{j_{256}}$ for $j_1, j_2, \ldots, j_{256} \in \{1, 2\}$ consisting of 256 blocks that all lead to the same hash value H_{256}. This results in a complexity of about $256 \cdot 2^{128} = 2^{136}$ evaluations of the compression function of GOST. Furthermore, the memory requirement is about $2 \cdot 256$ message blocks, *i.e.* we need to store 2^{14} bytes. With these multicollisions, we are able to construct the needed value of Σ^m in STEP 4 of the attack (where the superscript m stands for 'multicollision').

4.2 STEP 2: Pseudo-Preimages for the Last Iteration

We construct 2^{32} pseudo-preimages for the last iteration of GOST. For the given h, we proceed as described in Section 3 to construct a list L that consists of 2^{32} pairs (H_{258}, Σ^t) (where the superscript t stands for 'target'). Constructing the list L has a complexity of about $2^{32} \cdot 2^{192} = 2^{224}$ evaluations of the compression function of GOST. The memory requirements in this step come from the storage of 2^{32} pairs (H_{i-1}, M_i), *i.e.* we need to store 2^{32} 512-bit values or 2^{38} bytes.

4.3 STEP 3: Preimages Including the Length Encoding

In this step, we have to find a message block M_{257} such that for the given H_{256} determined in STEP 1, and for $|M|$ determined by our assumption that we want to construct preimages consisting of 257 message blocks, we find a H_{258} that is also contained in the list L constructed in STEP 2. Note that since we want to construct a message that is a multiple of 256 bits, we choose M_{257} to be a full message block and hence no padding is needed. We proceed as follows. Choose an arbitrary message block M_{257} and compute H_{258} as follows:

$$\begin{aligned} H_{257} &= f(H_{256}, M_{257}) \\ H_{258} &= f(H_{257}, |M|) \end{aligned}$$

where $|M| = (256 + 1) \cdot 256$. Then we check if the resulting value H_{258} is also in the list L. Since there are 2^{32} entries in L, we will find the right M_{257} with a probability of $2^{-256} \cdot 2^{32} = 2^{-224}$. Hence, after repeating this step of the attack about 2^{224} times, we will find an M_{257} and an according H_{258} that is also contained in the list L. Hence, this step of the attack requires 2^{225} evaluations of the compression function. Once we have found an appropriate M_{257}, also the value Σ^m is determined: $\Sigma^m = \Sigma^t \boxminus M_{257}$.

4.4 STEP 4: Constructing Σ^m

In STEP 1, we constructed a 2^{256} multicollision in the first 256 iterations of the hash function. From this set of messages that all lead to the same H_{256}, we now have to find a message $M^* = M_1^{j_1} \| M_2^{j_2} \| \cdots \| M_{256}^{j_{256}}$ for $j_1, j_2, \ldots, j_{256} \in \{1, 2\}$ that leads to the value of $\Sigma^m = \Sigma^t \boxminus M_{257}$. This can easily done by applying a meet-in-the-middle attack. First, we save all values for $\Sigma_1 = M_1^{j_1} \boxplus M_2^{j_2} \boxplus \cdots \boxplus M_{128}^{j_{128}}$ in the list L. Note that we have in total 2^{128} values in L. Second, we compute $\Sigma_2 = M_{129}^{j_{129}} \boxplus M_{130}^{j_{130}} \boxplus \cdots \boxplus M_{256}^{j_{256}}$ and check if $\Sigma^m \boxminus \Sigma_2$ is in the list L. After testing all 2^{128} values, we expect to find a matching entry in the list L and hence a message $M^* = M_1^{j_1} \| M_2^{j_2} \| \cdots \| M_{256}^{j_{256}}$ that leads to $\Sigma^m = \Sigma^t \boxminus M_{257}$. This step of the attack has a complexity of 2^{128} and a memory requirement of $2^{128} \cdot 2^5 = 2^{133}$ bytes. Once we have found M^*, we found a preimage for GOST consisting of 256+1 message blocks, namely $M^* \| M_{257}$.

The complexity of the preimage attack is determined by the computational effort of STEP 2 and STEP 3, *i.e.* a preimage of h can be found in about $2^{225} + 2^{224} \approx 2^{225}$ evaluations of the compression function. The memory requirements for the preimage attack are determined by finding M^* in STEP 4, since we need to store 2^{133} bytes for the standard meet-in-the-middle attack. Due to the high memory requirements of STEP 4, one could see this part as the bottleneck of the attack. However, the memory requirements of STEP 4 can be significantly reduced by applying a memory-less variant of the meet-in-the-middle attack introduced by Quisquater and Delescaille in [11].

4.5 A Remark on Second Preimages

Note that the presented preimage attack on GOST also implies a second preimage attack. In this case, we are not given only the hash value h but also a message M that results in this hash value. We can construct for any given message a second preimage in the same way as we construct preimages. The difference is, that the second preimage will always consist of at least 257 message blocks. Thus, we can construct a second preimage for any message M (of arbitrary length) with a complexity of about 2^{225} evaluations of the compression function of GOST.

Note that for long messages (more than 2^{32} message blocks) the generic second preimage attack of Gauravaram and Kelsey [4] is more efficient. For instance, a second preimage can be found for a message consisting of about 2^{54} message blocks with a complexity of 2^{203} evaluations of the compression function of GOST and 2^{142} bytes of memory.

5 Conclusion

In this article, we have presented a (second) preimage attack on GOST. Both the preimage and the second preimage attack have a complexity of about 2^{225} evaluations of the compression function and a memory requirement of about 2^{38} bytes. The internal structure of the compression function allows to construct pseudo-preimages with a complexity of about 2^{192}. This alone would not render the hash function insecure but would actually just constitute a certificational weakness. Nevertheless, the fact that we can construct multicollisions for any iterated hash function including GOST and the possibility of applying a meet-in-the-middle attack make the preimage and second preimage attack on GOST possible. More precisely, as opposed to most iterated hash functions, GOST additionally computes a checksum of the single message blocks which is then input to the final application of the compression function. For the preimage attack, we need a certain value in this chain after 256 iterations. The multicollision attack allows to generate a huge set of colliding messages such that we can generate any value for this checksum. Furthermore, a memory-less variant of meet-in-the-middle attack enables us to construct the specific value in an efficient way with respect to both running time and memory requirements.

Acknowledgements

The authors wish to thank Praveen Gauravaram, Mario Lamberger, Vincent Rijmen, and the anonymous referees for useful comments and discussions.

The work in this paper has been supported in part by the Austrian Science Fund (FWF), project P19863 and by the European Commission through the IST Programme under contract IST2002507 932 ECRYPT. The information in this paper is provided as is, and no guarantee or warranty is given or implied that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

References

1. Alex Biryukov and David Wagner. Advanced Slide Attacks. In Bart Preneel, editor, *EUROCRYPT*, volume 1807 of *LNCS*, pages 589–606. Springer, 2000.
2. John Black, Martin Cochran, and Trevor Highland. A Study of the MD5 Attacks: Insights and Improvements. In Matthew J. B. Robshaw, editor, *FSE*, volume 4047 of *LNCS*, pages 262–277. Springer, 2006.
3. Christophe De Cannière and Christian Rechberger. Finding SHA-1 Characteristics: General Results and Applications. In Xuejia Lai and Kefei Chen, editors, *ASIACRYPT*, volume 4284 of *LNCS*, pages 1–20. Springer, 2006.
4. Praveen Gauravaram and John Kelsey. Cryptanalysis of a Class of Cryptographic Hash Functions. Accepted at CT-RSA, 2008. Preprint available at `http://eprint.iacr.org/2007/277`.
5. Antoine Joux. Multicollisions in Iterated Hash Functions. Application to Cascaded Constructions. In Matthew K. Franklin, editor, *CRYPTO*, volume 3152 of *LNCS*, pages 306–316. Springer, 2004.
6. John Kelsey, Bruce Schneier, and David Wagner. Key-Schedule Cryptoanalysis of IDEA, G-DES, GOST, SAFER, and Triple-DES. In Neal Koblitz, editor, *CRYPTO*, volume 1109 of *LNCS*, pages 237–251. Springer, 1996.
7. Lars R. Knudsen and John Erik Mathiassen. Preimage and Collision Attacks on MD2. In Henri Gilbert and Helena Handschuh, editors, *FSE*, volume 3557 of *LNCS*, pages 255–267. Springer, 2005.
8. Youngdai Ko, Seokhie Hong, Wonil Lee, Sangjin Lee, and Ju-Sung Kang. Related Key Differential Attacks on 27 Rounds of XTEA and Full-Round GOST. In Bimal K. Roy and Willi Meier, editors, *FSE*, volume 3017 of *LNCS*, pages 299–316. Springer, 2004.
9. Mario Lamberger, Norbert Pramstaller, Christian Rechberger, and Vincent Rijmen. Second Preimages for SMASH. In Masayuki Abe, editor, *CT-RSA*, volume 4377 of *LNCS*, pages 101–111. Springer, 2007.
10. Markus Michels, David Naccache, and Holger Petersen. GOST 34.10 - A brief overview of Russia's DSA. *Computers & Security*, 15(8):725–732, 1996.
11. Jean-Jacques Quisquater and Jean-Paul Delescaille. How Easy is Collision Search. New Results and Applications to DES. In Gilles Brassard, editor, *CRYPTO*, volume 435 of *LNCS*, pages 408–413. Springer, 1989.
12. Markku-Juhani O. Saarinen. A chosen key attack against the secret S-boxes of GOST, 1998. unpublished manuscript.
13. Haruki Seki and Toshinobu Kaneko. Differential Cryptanalysis of Reduced Rounds of GOST. In Douglas R. Stinson and Stafford E. Tavares, editors, *Selected Areas in Cryptography*, volume 2012 of *LNCS*, pages 315–323. Springer, 2000.
14. Xiaoyun Wang, Xuejia Lai, Dengguo Feng, Hui Chen, and Xiuyuan Yu. Cryptanalysis of the Hash Functions MD4 and RIPEMD. In Ronald Cramer, editor, *EUROCRYPT*, volume 3494 of *LNCS*, pages 1–18. Springer, 2005.
15. Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding Collisions in the Full SHA-1. In Victor Shoup, editor, *CRYPTO*, volume 3621 of *LNCS*, pages 17–36. Springer, 2005.
16. Xiaoyun Wang and Hongbo Yu. How to Break MD5 and Other Hash Functions. In Ronald Cramer, editor, *EUROCRYPT*, volume 3494 of *LNCS*, pages 19–35. Springer, 2005.
17. Hongbo Yu, Xiaoyun Wang, Aaram Yun, and Sangwoo Park. Cryptanalysis of the Full HAVAL with 4 and 5 Passes. In Matthew J. B. Robshaw, editor, *FSE*, volume 4047 of *LNCS*, pages 89–110. Springer, 2006.

A A Pseudo-Collision for the Compression Function

In a similar way as we have constructed a pseudo-preimage in Section 3, we can construct a pseudo-collision for the compression function of GOST. In the attack, we have to find two pairs (H_{i-1}^1, M_i^1) and (H_{i-1}^2, M_i^2), where $H_{i-1}^1 \neq H_{i-1}^2$ or $M_i^1 \neq M_i^2$, such that $f(H_{i-1}^1, M_i^1) = f(H_{i-1}^2, M_i^2)$. The attack can be summarized as follows.

1. Choose random values for a, b_0, b_1, b_2, b_3 and c. This determines x_0.
2. Solve the set of $6 \cdot 64$ linear equations over $GF(2)$ to obtain 2^{128} pairs (H_{i-1}^j, M_i^j) for which x_0 in (17) is equal.
3. For each pair compute $X = x_3 \| x_2 \| x_1 \| x_0$ and save the the triple (X, H_{i-1}^j, M_i^j) in the list L. Note that x_0 is equal for all entries in the list L.
   After computing at most 2^{96} candidates for X one expect to find a matching entry (a collision) in L. Note that a collision is likely to exist due to the birthday paradox. Once, we have found a matching entry for X in the list L, we have also found a pseudo-collision for the compression function of GOST, since $H_i = \psi^{74}(X)$, see (16).

Note that memory-less variants of this attack can be devised [11]. Hence, we have a pseudo-collision for the compression function of GOST with a complexity of about 2^{96} instead of 2^{128} as expected for a compression function with an output size of 256 bits.