Cryptanalysis of the GOST Hash Function

Florian Mendel[1], Norbert Pramstaller[1], Christian Rechberger[1],
Marcin Kontak[2], and Janusz Szmidt[2]

[1] Institute for Applied Information Processing and Communications (IAIK),
Graz University of Technology, Inffeldgasse 16a, 8010 Graz, Austria
Florian.Mendel@iaik.tugraz.at
[2] Institute of Mathematics and Cryptology, Faculty of Cybernetics,
Military University of Technology, ul. Kaliskiego 2, 00-908 Warsaw, Poland

Abstract. In this article, we analyze the security of the GOST hash function. The GOST hash function, defined in the Russian standard GOST 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 iterative structure, a checksum computed over all input message blocks. This checksum is then part of the final hash value computation.
As a result of our security analysis of the GOST hash function, we present the first collision attack with a complexity of about 2^{105} evaluations of the compression function. Furthermore, we are able to significantly improve upon the results of Mendel *et al.* with respect to preimage and second preimage attacks. Our improved attacks have a complexity of about 2^{192} evaluations of the compression function.

Keywords: cryptanalysis, hash function, collision attack, second preimage attack, preimage attack

1 Introduction

A cryptographic hash function H maps a message M of arbitrary length to a fixed-length hash value h. Informally, 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. For a formal treatment of the security properties of cryptographic hash functions we refer to [18, 22].

Recent cryptanalytic results on hash functions mainly focus on collision attacks. Collisions have been shown for many commonly used hash functions (see for instance [5, 6, 16, 24–26]), but we are not aware of any published collision attack on the GOST hash function. In this article, we will present a security analysis of the GOST hash function with respect to both collision and (second) preimage resistance. The GOST hash function is widely used in Russia and is specified in the Russian national standard GOST 34.11-94 [3]. This standard has been developed by *GUBS of Federal Agency Government Communication and Information* and *All-Russian Scientific and Research Institute of Standardization*. The GOST hash function is the only hash function that can be used in the Russian digital signature algorithm GOST 34.10-94 [2]. Therefore, it is also used in several RFCs and implemented in various cryptographic applications (as for instance openSSL).

The GOST hash function 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 iterative structure, a checksum computed over all input message blocks. This checksum is then part of the final hash value computation. The GOST standard also specifies the GOST block cipher [1], which is the main building block of the hash function. Therefore, 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 [4, 11, 14, 19, 20]), only a few results regarding the hash function have been published to date. Note that for the remainder of this article we refer to the GOST hash function simply as GOST.

Related Work. In [7], Gauravaram and Kelsey show that the generic attacks on hash functions based on the Merkle-Damgård design principle can be extended to hash functions with linear/modular checksums independent of the underlying compression function. Hence, second preimages can be found for long messages (consisting of 2^t message blocks) for GOST with a complexity of 2^{n-t} evaluations of the compression function.

At FSE 2008, Mendel *et al.* have presented the first attack on GOST exploiting the internal structure of the compression function. The authors exploit weaknesses in the internal structure of GOST to construct pseudo-preimages for the compression function of GOST with a complexity of about 2^{192} compression function evaluations. Furthermore, they show how the 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} evaluations of the compression function of GOST. Both attacks are structural attacks in the sense that they are independent of the underlying block cipher.

Our Contribution. We improve upon the state of the art as follows. First, we show that for plaintexts of a specific structure, we can construct fixed-points in the GOST block cipher efficiently. Second, based on this property in the GOST block cipher we then show how to construct collisions in the compression function of GOST with a complexity of 2^{96} compression function evaluations. This collision attack on the compression function is then extended to a collision attack on the GOST hash function. The extension is possible by combining a multicollision attack and a generalized birthday attack on the checksum. The attack has a complexity of about 2^{105} evaluations of the compression function of GOST. Furthermore, we show that due to the generic nature of our attack we can construct meaningful collisions, *i.e.* collisions in the chosen-prefix setting with the same complexity. Note that in most cases constructing meaningful collisions is more complicated than constructing (random) collisions (see for instance MD5 [21]). Third, we show how the (second) preimage attack of Mendel *et al.* can be improved by additionally exploiting weaknesses in the GOST block cipher. The new improved (second) preimage attack has a complexity of 2^{192} evaluations of the compression function of GOST.

Table 1. Comparison of results for the GOST hash function.

source	attack complexity	attack
Gauravaram and Kelsey CT-RSA 2008 [7]	2^{256-t}	second preimages for long messages (2^t blocks)
Mendel *et al.* FSE 2008 [15]	2^{225}	preimages and second preimages
this work	2^{105}	collisions
	2^{105}	meaningful collisions (chosen-prefix)
	2^{192}	preimages and second preimages

The remainder of this article is structured as follows. In Section 2, we give a short description of the GOST hash function. In Section 3, we describe the GOST block cipher and show how to construct fixed-points efficiently. We use this in the collision attack on the hash function in Section 4. In Section 5, we show a new improved (second) preimage attack for the hash function. Finally, we present conclusions in Section 6.

2 The Hash Function 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,

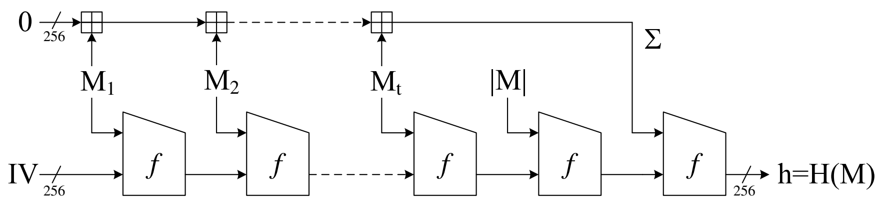

Fig. 1. Structure of the GOST hash function.

an unambiguous padding method is applied. For the description of the padding method we refer to [3]. Let $M = M_1 \| M_2 \| \cdots \| M_t$ be a t-block message (after padding). The hash value $h = H(M)$ is computed as follows (see Fig. 1):

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

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.

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 Section 3, for a detailed description of the GOST block cipher.

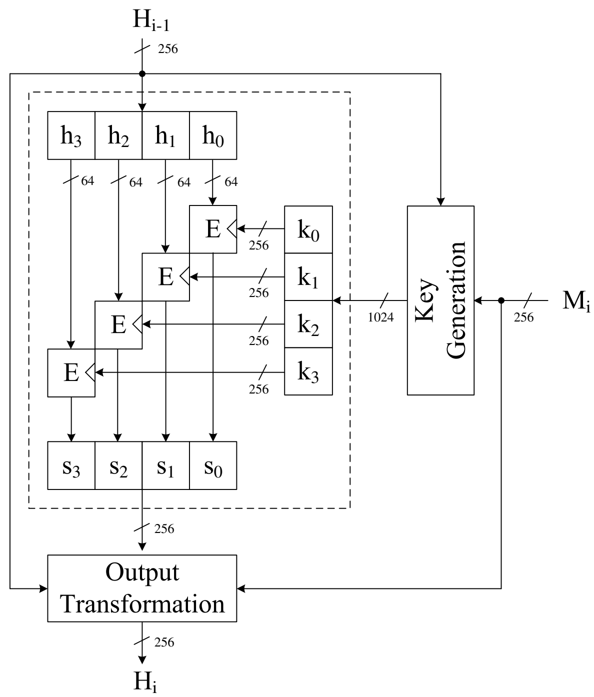

Fig. 2. The compression function of GOST

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 [3], since we do not need it for our analysis.

2.3 Output Transformation

The output transformation of GOST combines the intermediate hash 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 $\psi : \{0,1\}^{256} \rightarrow \{0,1\}^{256}$ 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 The GOST Block Cipher

The GOST block cipher is specified by the Russian government standard GOST 28147-89 [1]. Several cryptanalytic results have been published for the block cipher (see for instance [4, 11, 14, 19, 20]). However, if the block cipher is used in a hash function then we are facing a different attack scenario: the attacker has full control over the key. First results considering this fact for the security analysis of hash functions have been presented for instance in [13]. We will exploit having full control over the key for constructing fixed-points for the GOST block cipher.

3.1 Description of the block cipher

The GOST block cipher is a 32 round Feistel network with a block size of 64 bits and a key length of 256 bits. The round function of the GOST block cipher consists of a key addition, eight different 4×4 S-boxes S_j $(0 \leq j < 8)$ and a cyclic rotation (see also Figure 3). For the definition of the S-boxes we refer to [3], since we do not need them for our analysis.

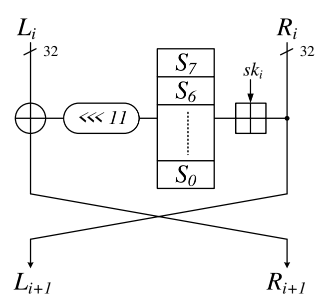

Fig. 3. One round of the GOST block cipher.

The key schedule of the GOST block cipher defines the subkeys sk_i derived from the 256-bit $K = k_7 \| k_6 \| \cdots \| k_0$ as follows

$$sk_i = \begin{cases} k_{i \bmod 8}, & i = 0, \ldots, 23, \\ k_{7-(i \bmod 8)}, & i = 24, \ldots, 31. \end{cases} \tag{15}$$

3.2 Constructing a Fixed-Point

In the following, we will show how to efficiently construct fixed-points in the GOST block cipher. It is based on the following observation. Note that a similar observation was used by Kara in [10] for a chosen plaintext attack on the GOST block cipher.

Observation 1 *Assume we are given a plaintext $P = L_0 \| R_0$ with $L_0 = R_0$. Then we can construct a fixed-point for the block cipher by constructing a fixed-point in the first 8 rounds.*

In the following, we refer to a plaintext $P = L_0 \| R_0$ with $L_0 = R_0$ as a *symmetric* plaintext (or for short as symmetric). Note that by using the block cipher for constructing a hash function, an attacker has full control over the key. Furthermore, each word of the key is only used once in the first 8 rounds of the block cipher. Hence, constructing a fixed-point in the first 8 rounds can be done efficiently. First, we choose random values for the first 6 words of the key (subkeys $sk_0, \ldots, sk_5$) and compute L_6 and R_6. Next, we choose the last 2 words of the key (subkeys sk_6 and sk_7) such that $L_8 = L_0$ and $R_8 = R_0$. With this method we can construct a fixed-point in the first 8 rounds of the block cipher with a computational cost of 8 round computations.

It is easy to see that if we have a fixed-point in the first 8 rounds, then this is also a fixed-point for rounds 9-16 and 17-24 since the same subkeys are used in these rounds. In the last 8 rounds the subkey is put in the opposite order, see (15). However, since the GOST block cipher is a Feistel network, we have here (rounds 25-32) a decryption if $L_{24} = R_{24}$. This implies that we have a fixed-point for the GOST block cipher (for all 32 rounds) if the plaintext is symmetric. Hence, for symmetric plaintexts we can efficiently construct fixed-points for the GOST block cipher.

4 Collision Attack on GOST

In this section, we present a collision attack on the GOST hash function with a complexity of about 2^{105} evaluations of the compression function. First, we will show how to construct collisions for the compression function of GOST, and based on this attack we then describe the collision attack for the hash function. For the remainder of this article we follow the notation of [15].

4.1 Constructing a Collision in the Compression Function

In the following, we show how to construct a collision in the compression function of GOST. The attack is based on structural weaknesses of the compression function. These weaknesses have been used in [15] to construct pseudo-collisions and pseudo-preimages for the compression function of GOST with a complexity of 2^{96} and 2^{192}, respectively.

Now we show a collisions attack on the compression function by additionally exploiting weaknesses in the underlying GOST block cipher. Since the transformation ψ is linear, (13) can be written as:

$$H_i = \psi^{61}(H_{i-1}) \oplus \psi^{62}(M_i) \oplus \psi^{74}(S) \tag{16}$$

Furthermore, ψ is invertible and hence (16) can be written as:

$$\underbrace{\psi^{-74}(H_i)}_{X} = \underbrace{\psi^{-13}(H_{i-1})}_{Y} \oplus \underbrace{\psi^{-12}(M_i)}_{Z} \oplus S \tag{17}$$

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 (17) 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, (17) can be written as:

$$x_0 = y_0 \oplus z_0 \oplus s_0 \tag{18}$$
$$x_1 = y_1 \oplus z_1 \oplus s_1 \tag{19}$$
$$x_2 = y_2 \oplus z_2 \oplus s_2 \tag{20}$$
$$x_3 = y_3 \oplus z_3 \oplus s_3 \tag{21}$$

Now assume, that we can find 2^{96} message blocks M_i^j, where $M_i^k \neq M_i^t$ with $k \neq t$, such that all message blocks produce the same value x_0. Then we know that due to the birthday paradox two of these message blocks also lead to the same values x_1, x_2, and x_3. In other words, we have constructed a collision for the compression function of GOST. The attack has a complexity of about 2^{96} evaluations of the compression function of GOST.

Based on this short description, we will show now how to construct message blocks M_i^j, which all produce the same value x_0. Assume, we want to keep the value s_0 in (18) constant. Since $s_0 = E(k_0, h_0)$ and k_0 depends linearly on the message block M_i, we have to find keys k_0^j and hence, message blocks M_i^j, which all produce the same value s_0. This can be done by exploiting the fact that in the GOST block cipher fixed-points can be constructed efficiently for symmetric plaintexts (see Section 3.2). In other words, if h_0 is symmetric then we can construct 2^{96} message blocks M_i^j where $s_0 = h_0$, and (18) becomes

$$x_0 = y_0 \oplus z_0 \oplus h_0\ . \tag{22}$$

However, to find message blocks 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 (22) has the same value for all message blocks. Therefore, we get the following equation (64 equations over $GF(2)$)

$$y_0 \oplus z_0 = c \tag{23}$$

where c is an arbitrary 64-bit value. We know that y_0 depends linearly on H_{i-1} and z_0 depends linearly on M_i, see (17). Therefore, the choice of the message block M_i and accordingly, the choice of the key k_0, is restricted by 64 equations over $GF(2)$. Hence, for constructing a fixed-point in the GOST block cipher we have to consider these restrictions. For the following discussion let

$$A \cdot k_0 = d \tag{24}$$

denote the set of 64 equations over $GF(2)$ which restricts the choice of the key k_0, where A is a 64×256 matrix over $GF(2)$ and d is a 64-bit vector. It follows from Observation 1 that for constructing a fixed-point in the GOST block cipher (for symmetric plaintexts), it is sufficient to construct a fixed-point in the first 8 rounds. Hence, one method to construct an appropriate fixed-point would be to construct many arbitrary fixed-points and then check if (24) holds. With this method we find an appropriate fixed-point with a complexity of about 2^{64}. Since we need 2^{96} such fixed-points for the collision attack, this would lead to a complexity of 2^{160} evaluations of the compression function of GOST. However, we can improve this complexity by using a meet-in-the-middle approach (see also Fig. 4).

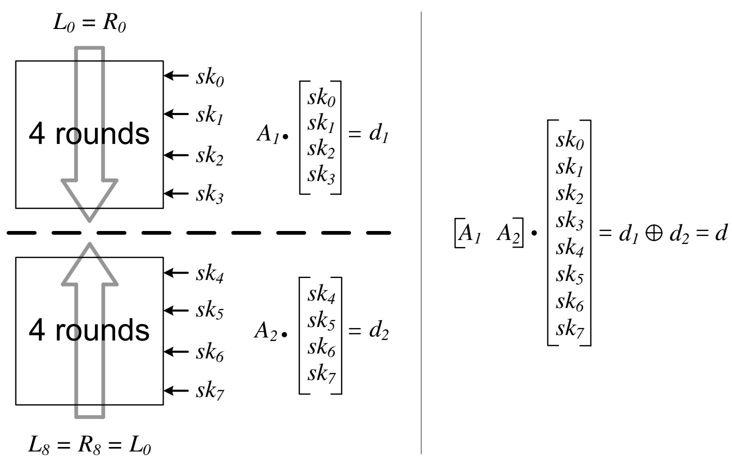

Fig. 4. Constructing a fixed-point in the GOST block cipher.

We split the first 8 rounds of the GOST block cipher into 2 parts P_1 (rounds 1-4) and P_2 (rounds 5-8). Since the subkey used in the first 8 rounds is restricted by $A \cdot k_0$, we also split this system of 64 equations over $GF(2)$ into two parts:

$$A_1 \cdot \begin{bmatrix} sk_0 \\ sk_1 \\ sk_2 \\ sk_3 \end{bmatrix} = d_1 \qquad A_2 \cdot \begin{bmatrix} sk_4 \\ sk_5 \\ sk_6 \\ sk_7 \end{bmatrix} = d_2 \tag{25}$$

where $A = [A_1 \; A_2]$ and $d = d_1 \oplus d_2$. Now we can apply a meet-in-the-middle attack to construct 2^{64} appropriate fixed-points for the GOST block cipher with a complexity of 2^{64}. It can be summarized as follows.

1. Choose a random value for d_1. This determines also $d_2 = d \oplus d_1$.
2. For all 2^{64} subkeys $sk_0, \ldots, sk_3$ which fulfill (25) compute L_4, R_4 and save the result in the list L.
3. For all 2^{64} subkeys $sk_4, \ldots, sk_7$ which fulfill (25) compute rounds 4-8 backward to get L_4, R_4 and check for a matching entry in the list L. Note that since there are 2^{64} entries in the list L we expect to always find a matching entry in the list L. Hence, we get 2^{64} appropriate fixed-points for the GOST block cipher with a complexity of about 2^{64} and memory requirements of $2^{64} \cdot 40 \approx 2^{70}$ bytes.

By repeating this attack about 2^{32} times for different choices of d_1, we get 2^{96} appropriate fixed-points. In other words, we found 2^{96} keys k_0^j which all produce the same value $s_0 = E(k_0^j, h_0)$ and additionally fulfill (24). Consequentially, we have 2^{96} message blocks M_i^j which all result in the same value x_0 with $X = \psi^{-74}(H_i)$. By applying a birthday attack we will find two message blocks M_i^k and M_i^t with $k \neq t$ where also x_1, x_2, and x_3 are equal. In other words, we can find a collision for the compression function of GOST with a complexity of about 2^{96} instead of 2^{128} evaluations of the compression function of GOST.

4.2 Constructing Collisions for the Hash Function

In this section, we show how the collision attack on the compression function can be extended to the hash function. The attack has a complexity of about 2^{105} evaluations of the compression function of GOST. Note that the hash function defines, in addition to the common iterative structure, a checksum computed over all input message blocks which is then part of the final hash computation. Therefore, to construct a collision in the hash function we have to construct a collision in the iterative structure (*i.e.* chaining variables) as well as in the checksum. To do this we use multicollisions.

A multicollision is a set of messages of equal length that all lead to the same hash value. As shown in [9], constructing a 2^t collision, *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 $t \cdot 2^x$ for any iterated hash function, where 2^x is the cost of constructing a collision in the compression function. As shown in Section 4.1, collisions for the compression function of GOST can be constructed with a complexity of 2^{96} if h_0 is symmetric in $H_{i-1} = h_3 \| h_2 \| h_1 \| h_0$. Note that by using an additional message block M_{i-1} we find a chaining variable $H_{i-1} = f(H_{i-2}, M_{i-1})$, where h_0 is symmetric with a complexity of 2^{32} compression function evaluations. Hence, we can construct a 2^{128} collision with a complexity of about $128 \cdot (2^{96} + 2^{32}) \approx 2^{103}$ evaluations of the compression function of GOST. With this method we get 2^{128} messages M^* that all lead to the same value H_{256} as depicted in Figure 5.

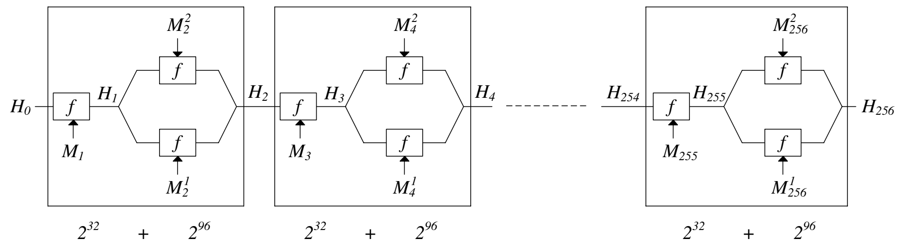

Fig. 5. Constructing a multicollision for GOST.

To construct a collision in the checksum of GOST we have to find 2 distinct messages which produce the same value $\Sigma = M_1 \boxplus M_2^{j_2} \boxplus \cdots \boxplus M_{255} \boxplus M_{256}^{j_{256}}$ with $j_2, j_4, \ldots j_{256} \in \{1, 2\}$. By applying a birthday attack we can find these 2 messages with a complexity of about 2^{127} additions over $GF(256)$ and memory requirements of 2^{134} bytes. Due to the high complexity and memory requirements of the birthday attack, one could see this part as the bottleneck of the attack. However, the runtime and memory requirements can significantly be reduced by applying a generalized birthday attack introduced by Wagner in [23]. Wagner shows that if ℓ is a power of two then the memory requirements and the running time for the generalized birthday problem is given by $2^{n/(1+\lg \ell)}$ and $\ell \cdot 2^{n/(1+\lg \ell)}$, respectively. Note that in the standard birthday attack we have $\ell = 2^1$.

Let us now consider the case $\ell = 2^3$. Then the birthday attack in the second part of the attack has a complexity of $2^3 \cdot 2^{256/4} = 2^{67}$ and uses lists of size $2^{256/4} = 2^{64}$. In detail, we need to construct 8 lists of size 2^{64} in the first step of the attack. Hence, we need to construct a $2^{8 \cdot 64}$ collision in the first part of the attack to get 8 lists of the needed size. Constructing this multicollision has a complexity of about $8 \cdot 64 \cdot (2^{32} + 2^{96}) = 2^{105}$ compression function evaluations and memory requirements of $8 \cdot 64 \cdot (2 \cdot 64) = 2^{16}$ bytes. Hence, we can construct a collision for the GOST hash function with a complexity of about 2^{105} and memory requirements of $2^{64} \cdot 2^6 = 2^{70}$ bytes by using a generalized birthday attack with $\ell = 8$ lists. Furthermore, the colliding message pair consists of $8 \cdot (2 \cdot 64) = 1024$ message blocks. Note that $\ell = 8$ is the best choice for the attack. On one hand if we choose $\ell > 8$ then the memory requirements of the attack would decrease but the attack complexity would increase. Since we need about 2^{70} bytes of memory for constructing fixed-points in the GOST block cipher, this does not improve the attack. On the other hand if we choose $\ell < 8$ then the memory requirements of the attack would be significantly higher.

A Remark on the Length Extension Property. Once, we have found a collision, *i.e.* collision in the iterative part (chaining variables) and the checksum, we can construct many more collisions by appending an arbitrary message block. Note that this is not necessarily the case for a straight-forward birthday attack. By applying a birthday attack we construct a collision in the final hash value (after the last iteration) and appending a message block is not possible. Hence, we need a collision in the iterative part as well as in the checksum for the extension property. Note that by combining the generic birthday attack and multicollisions, one can construct collisions in both parts with a complexity of about $128 \cdot 2^{128} = 2^{135}$ while our attack has a complexity of 2^{105}.

A Remark on Meaningful Collisions. In a chosen-prefix setting, an attacker searches for a message pair (M, M^*) such that for a given hash function H

$$H(M_{pre} \| M) = H(M^*_{pre} \| M^*) \tag{26}$$

for any pair (M_{pre}, M^*_{pre}). In [21], Stevens *et al.* show that such a more powerful attack exists for MD5. Furthermore, they describe an application of this attack for colliding X.509 certificates. Note that their attack (in a chosen-prefix setting) has a complexity of 2^{50}, while the currently best published collision attack for MD5 has a complexity of about 2^{30} evaluations of the compression function [12].

However, in the case of GOST the collision attack in the chosen-prefix setting has the same complexity as the collision attack. Due to the generic nature of the collision attack, differences in the chaining variables can be canceled efficiently. Assume that the chosen prefix (M_{pre}, M^*_{pre}) consists of t message blocks resulting in the chaining variables H_t and H^*_t. Then the attack can be summarized as follows.

1. We have to find two message blocks M_{t+1} and M^*_{t+1} such that $h_0 = h^*_0 = 0$, where $H_{t+1} = h_3 \| h_2 \| h_1 \| h_0$ and $H^*_{t+1} = h^*_3 \| h^*_2 \| h^*_1 \| h^*_0$. This has a complexity of about $2 \cdot 2^{64}$ evaluations of the compression function of GOST.
2. Now we have to find two message blocks M_{t+2} and M^*_{t+2} such that $H_{t+2} = H^*_{t+2}$. This can be done similar as constructing a collision in the compression function of GOST (see Section 4.1). First, we choose a random value for c in (22) and construct 2^{96} message blocks M_{t+2}, where x_0 is equal. Second, we construct 2^{96} message blocks M^*_{t+2}, where $x^*_0 = x_0$. To guarantee that $x_0 = x^*_0$ we have to adjust c^* in (22) such that the following equation holds.

$$x_0 = x^*_0 = y^*_0 \oplus z^*_0 \oplus h^*_0 = c^* \oplus h^*_0$$

   By applying a meet-in-the-middle attack we will find two message blocks M_{t+2} and M^*_{t+2} which produce the same chaining variables ($H_{t+2} = H^*_{t+2}$). This step of the attack has a complexity of $2 \cdot 2^{96}$ evaluations of the compression function of GOST.
3. Once we have constructed a collision in the iterative part (chaining variables), we have to construct a collision in the checksum as well. Therefore, we proceed as described in Section 4.2. By generating a 2^{512} collision and applying a generalized birthday attack with $\ell = 8$ we can construct a collision in the checksum of GOST with a complexity of 2^{105} compression function evaluations and memory requirements of 2^{70} bytes.

Hence, we can construct meaningful collisions, *i.e.* collisions in the chosen-prefix setting, for the GOST hash function with a complexity of about 2^{105} compression function evaluations.

5 Improved 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^{192} evaluations of the compression function of GOST. Before describing the attack, we will first show how to construct preimages for the compression function of GOST. Based on this attack we then present the preimage attack for the hash function.

5.1 A Preimage Attack for the Compression Function

In a similar way as we have constructed a collision in Section 4.1, we can construct a preimage for the compression function of GOST. In the attack, we have to find a message block M_i, such that $f(H_{i-1}, M_i) = H_i$ for the given values H_{i-1} and H_i. Note that the value of H_i determines $x_3, \ldots, x_0$, since $X = \psi^{-74}(H_i)$. Furthermore, assume that h_0 (in $H_{i-1} = h_3 \| \cdots \| h_0$) is symmetric. Then the attack can be summarized as follows.

1. Since we will construct fixed-points for the GOST block cipher such that $s_0 = E(k_0, h_0) = h_0$, we have to adjust c in (22) such that

$$x_0 = y_0 \oplus z_0 \oplus h_0 = c \oplus h_0$$

   holds with $X = \psi^{-74}(H_i)$. Once c is fixed, this also determines d in (24).
2. Choose a random value for d_1 (this also determines $d_2 = d \oplus d_1$) and apply a meet-in-the-middle attack to obtain 2^{64} message blocks M_i^j for which x_0 is correct. Note that this step of the attack has memory requirements of 2^{70} bytes.
3. For each message block 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^{64} message blocks, we will find a correct message block with a probability of $2^{-192} \cdot 2^{64} = 2^{-128}$. Note that we can repeat the attack about 2^{64} times for different choices of d_1.

Hence, we will find a preimage for the compression function of GOST with a probability of about 2^{-64} and a complexity of about 2^{128} evaluations of the compression function of GOST and memory requirements of 2^{70} bytes.

5.2 Extending the Attack to the Hash Function

Now, we show how the preimage attack on the compression function can be extended to the GOST hash function. The attack has a complexity of about 2^{192} evaluations of the compression function of GOST. Moreover, the preimage consists of 257 message blocks, *i.e.* $M = M_1 \| \cdots \| M_{257}$. The preimage attack consists of four steps as also shown in Figure 6.

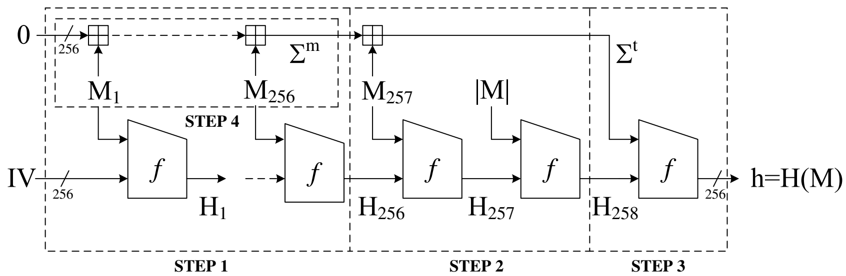

Fig. 6. Preimage Attack on GOST.

STEP 1: Multicollisions for GOST. For the preimage attack on the hash function, we construct a 2^{256} collision. 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').

STEP 2: Constructing $\boldsymbol{H_{258}}$ 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 $H_{258} = h_3\|\cdots\|h_0$ where h_0 is symmetric. 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:

$$H_{257} = f(H_{256}, M_{257})$$
$$H_{258} = f(H_{257}, |M|)$$

where $|M| = (256 + 1) \cdot 256$. Then we check if h_0 in the resulting value H_{258} is symmetric. This has a probability of 2^{-32}. Hence, this step of the attack requires $2 \cdot 2^{32}$ evaluations of the compression function of GOST.

STEP 3: Preimages for the Last Iteration. To construct a preimage for the last iteration of GOST we proceed as described in Section 5.1. Since h_0 in H_{258} is symmetric, we will find a preimage for the last iteration of GOST with a probability of 2^{-64} (and a complexity of about 2^{128}). Therefore, we have to repeat this step of the attack about 2^{64} times for different values of H_{258} (where h_0 is symmetric) to find a preimage for the last iteration. Hence, finishing this step of the attack has a complexity of about $2^{64} \cdot (2 \cdot 2^{32} + 2^{128}) \approx 2^{192}$ evaluations of the compression function of GOST. Once we have found a preimage for the last iteration, also the value Σ^m is determined, since $\Sigma^m = \Sigma^t \boxminus M_{257}$.

STEP 4: Constructing Σ^m. In STEP 1, we constructed a 2^{256} collision 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 be 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 Attack Complexity. 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^{192} 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 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 significantly be reduced by applying a memory-less variant of the meet-in-the-middle attack introduced by Quisquater and Delescaille in [17]. Hence, a preimage can be constructed for the GOST hash function with a complexity of 2^{192} evaluations of the compression function and memory requirements of about 2^{70} bytes.

5.3 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^{192} evaluations of the compression function of GOST.

6 Conclusion

In this article, we have presented a collision attack and a (second) preimage attack on the GOST hash function. Both the collision and the (second) preimage attack are based on weaknesses in the GOST block cipher, namely fixed-points can be constructed efficiently for plaintexts of a specific structure. The internal structure of the compression function allows to construct collisions with a complexity of about 2^{96} evaluations of the compression function. This alone would not render the hash function insecure. The fact that we can construct multicollisions for any iterated hash function including the GOST hash function and the possibility of applying a (generalized) birthday attack to construct also a collision in the checksum make the collision attack on the hash function possible. The attack has a complexity of about 2^{105} compression function evaluations. Furthermore, the generic nature of the attack allows us to construct meaningful collisions, *i.e.* collisions in the chosen-prefix setting, with the same complexity. In a similar way as we construct collisions for the hash function, we can construct (second) preimages for the hash function with a complexity of about 2^{192} evaluations of the compression function. This improves the previous (second) preimage attack of Mendel *et al.* by a factor of 2^{33}. Even though the complexities of our attacks are far beyond of being practical, they point out weaknesses in the design principles of the hash function GOST.

Acknowledgements

The authors wish to thank 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 Secure Information Technology Center - Austria (A-SIT) and by the Austrian Science Fund (FWF), project P19863.

References

1. GOST 28147-89, Systems of the Information Treatment. Cryptographic Security. Algorithms of the Cryptographic Transformation, 1989. (In Russian).
2. GOST 34.10-94, Information Technology Cryptographic Data Security Produce and Check Procedures of Electronic Digital Signature Based on Asymmetric Cryptographic Algorithm, 1994. (In Russian).
3. GOST 34.11-94, Information Technology Cryptographic Data Security Hashing Function, 1994. (In Russian).
4. Alex Biryukov and David Wagner. Advanced Slide Attacks. In Bart Preneel, editor, *EUROCRYPT*, volume 1807 of *LNCS*, pages 589–606. Springer, 2000.
5. Christophe De Cannière, Florian Mendel, and Christian Rechberger. Collisions for 70-Step SHA-1: On the Full Cost of Collision Search. In Carlisle M. Adams, Ali Miri, and Michael J. Wiener, editors, *Selected Areas in Cryptography*, volume 4876 of *LNCS*, pages 56–73. Springer, 2007.
6. 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.
7. Praveen Gauravaram and John Kelsey. Linear-XOR and Additive Checksums Don't Protect Damgård-Merkle Hashes from Generic Attacks. In Tal Malkin, editor, *CT-RSA*, volume 4964 of *LNCS*, pages 36–51. Springer, 2008.
8. Daniel Joscák and Jirí Tuma. Multi-block Collisions in Hash Functions Based on 3C and 3C+ Enhancements of the Merkle-Damgård Construction. In Min Surp Rhee and Byoungcheon Lee, editors, *ICISC*, volume 4296 of *LNCS*, pages 257–266. Springer, 2006.

9. 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.
10. Orhun Kara. Reflection Attacks on Product Ciphers. Cryptology ePrint Archive, Report 2007/043, 2007. http://eprint.iacr.org/.
11. 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.
12. Vlastimil Klima. Tunnels in Hash Functions: MD5 Collisions Within a Minute. Cryptology ePrint Archive, Report 2006/105, 2006. http://eprint.iacr.org/.
13. Lars R. Knudsen and Vincent Rijmen. Known-Key Distinguishers for Some Block Ciphers. In Kaoru Kurosawa, editor, *ASIACRYPT*, volume 4833 of *LNCS*, pages 315–324. Springer, 2007.
14. 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.
15. Florian Mendel, Norbert Pramstaller, and Christian Rechberger. A (Second) Preimage Attack on the GOST Hash Function. In Kaisa Nyberg, editor, *FSE*, volume 5086 of *LNCS*, pages 224–234. Springer, 2008.
16. Florian Mendel and Vincent Rijmen. Cryptanalysis of the Tiger Hash Function. In Kaoru Kurosawa, editor, *ASIACRYPT*, volume 4833 of *LNCS*, pages 536–550. Springer, 2007.
17. 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.
18. Phillip Rogaway and Thomas Shrimpton. Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance. In Bimal K. Roy and Willi Meier, editors, *FSE*, volume 3017 of *LNCS*, pages 371–388. Springer, 2004.
19. Markku-Juhani O. Saarinen. A chosen key attack against the secret S-boxes of GOST, 1998. unpublished manuscript.
20. 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.
21. Marc Stevens, Arjen K. Lenstra, and Benne de Weger. Chosen-Prefix Collisions for MD5 and Colliding X.509 Certificates for Different Identities. In Moni Naor, editor, *EUROCRYPT*, volume 4515 of *LNCS*, pages 1–22. Springer, 2007.
22. Douglas R. Stinson. Some Observations on the Theory of Cryptographic Hash Functions. *Des. Codes Cryptography*, 38(2):259–277, 2006.
23. David Wagner. A Generalized Birthday Problem. In Moti Yung, editor, *CRYPTO*, volume 2442 of *LNCS*, pages 288–303. Springer, 2002.
24. 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.
25. 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.
26. 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.