Differential Fault Analysis of Streebog

Riham AlTawy and Amr M. Youssef

Concordia Institute for Information Systems Engineering,
Concordia University, Montréal, Québec, Canada

Abstract. In August 2012, the Streebog hash function was selected as the new Russian federal hash function standard (GOST R 34.11-2012). In this paper, we present a fault analysis attack on this new hashing standard. In particular, our attack considers the compression function in the secret key setting where both the input chaining value and the message block are unknown. The fault model adopted is the one in which an attacker is assumed to be able to cause a bit-flip at a random byte in the internal state of the underlying cipher of the compression function. We also consider the case where the position of the faulted byte can be chosen by the attacker. In the sequel, we propose a two-stage approach that recovers the two secret inputs of the compression function using an average number of faults that varies between 338-1640, depending on the assumptions of our employed fault model. Moreover, we show that the attack can be extended to the iterated hash function using a feasible pre-computation stage. Finally, we analyze Streebog in different MAC settings and demonstrate how our attack can be used to recover the secret key of HMAC/NMAC-GOST.

Keywords: Differential fault analysis, Hash functions, Cryptanalysis, HMAC, NMAC, GOST R 34.11-2012, Streebog.

1 Introduction

Streebog is a Russian hash function that was originally proposed in 2010 [24] as a replacement for the previous standard GOST R 34.11-94 which has been theoretically broken in [26,25]. Later, in 2012, the Russian Federation Technical Committee for Standardization (TC 26) announced Streebog as the new hash standard GOST R 34.11-2012. The function supports digest sizes of 256 and 512 bits. The compression function employs a 12-round AES-like cipher with 8×8-byte internal state. The compression function operates in Miyaguchi-Preneel (MP) mode and is plugged in a modified Merkle-Damgård domain extender with a modular checksum finalization step [1]. The new GOST is also standardized by IETF as RFC 6896 [17].

Due to the significance of this standard, its security has been thoroughly investigated in a series of works appearing in a relatively short time. These works include the analysis of the collision resistance of its compression function and internal cipher by AlTawy *et al.* [2] and Wang *et al.* [30]. An integral analysis of the compression function has been presented by AlTawy and Youssef where

© Springer International Publishing Switzerland 2015
J. Lopez and Y. Wu (Eds.): ISPEC 2015, LNCS 9065, pp. 35–49, 2015.
DOI: 10.1007/978-3-319-17533-1_3

integral distinguishers for the reduced compression function was proposed [3]. Moreover, preimage attacks on the reduced hash function have been independently proposed by Altawy and Youssef [4] and Zou *et al.* [31], and later the attacks were improved by Bingka *et al.* [23]. Also, Kazymyrov and Kazymyrova presented an analysis of the algebraic aspects of the function [19], and a long second preimage attack was proposed by Guo *et al.* [15]. Finally, a malicious version of the whole hash function was presented in [5].

In this work, we present a practical differential fault analysis attack (DFA) on Streebog. The attack considers the compression function when operating with secret inputs which is the default setting when the function is used in a message authentication code (MAC) scheme. In other words, we consider that both the input chaining value and message block are unknown and that we can only observe the output of the compression function. In the sequel, we propose a two-stage attack using the one-bit fault model where the attacker is able to cause a bit flip at a chosen or random byte in the internal state of the function. Employing a specific property of the Streebog Sbox and by observing several correct and faulty compression function outputs, the first stage of the attack bypasses the final feedforward and retrieves the state of the internal cipher. Since all inputs are unknown, the retrieved state does not allow us to invert the internal cipher of the compression function because its round keys are dependant on the input chaining value which is a secret. Accordingly, in the second stage of the attack, we recover one of the round keys which enables the recovery of both the chaining value and message block of the attacked compression function. To this end, we are restricted to the processing of the last compression function in the iterated hash function as it is the only one which we can observe both its correct and faulty outputs. For that, we employ two precomputed tables which allows us to extend the attack to the whole hash function. Finally, we analyze the GOST hash function in different MAC [7] settings and show how to use our attack to recover the secret MAC key of simple prefix and secret-IV MACs [27], HMAC, and NMAC [7].

The rest of the paper is organized as follows. In the next section, a brief overview on fault analysis attacks is given. The description of the Streebog hash function along with the notation used throughout the paper are provided in section 3. Afterwards, in section 4, we provide a detailed description of the used fault model, our two-stage approach, and show how to extend the attack from the compression function to the whole hash function. In section 5, we consider Streebog operating in different MAC settings and present the approaches used in the key recovery of simple prefix, secret-IV, HMAC, and NMAC. Simulation results and analysis of the number of required faults for different attack scenarios are given in section 6. Finally, the paper is concluded in section 7.

2 Fault Analysis

In mathematical attacks, such as differential and linear cryptanalysis, the attacker tries to exploit any weakness in the underlying mathematical structure

of the cryptographic primitive. In fault analysis, which is an implementation dependant attack, the attacker faults the state of the primitive during its computation to deduce information about its secret material. In particular, the attacker applies some kind of physical intervention during the computation of the internal state of the primitive which corrupts random or known bits in the state. Consequently, the attacker observes the correct and the faulty outputs and performs differential fault analysis [9]. During this analysis, the attacker gains non negligible information about the secret material embedded in the hardware by comparing the correct and faulty outputs. Fault injection can be done in many ways which include power glitches, clock pulses, and laser radiation. The reader is referred to [28,12] for more details about the practical experimentation with different methods of fault injection.

Fault analysis was first introduced when Boneh *et al.* showed how the private key of the RSA-CRT-algorithm can be successfully recovered by observing the correct ciphertext and then injecting a fault and acquiring the faulty ciphertext [10]. Later on, Biham and Shamir combined fault analysis with differential cryptanalysis and presented differential fault analysis [9] against DES. Their attack works by observing the difference between the correct and faulty ciphertexts and exploiting this relation to recover the key of DES. DFA attacks have been used for the analysis of the hardware security of many ciphers (e.g., see [6,14,29]). In particular and due to its significance as a standard, AES has received a lot of attention with regards to DFA where some of the works used fault injection in the encryption process [29,14], and others attacked the key schedule [21]. DFA attacks vary in the number of required faults depending on the employed fault model. Generally, all models assume that the attacker has access to the physical device, and is able to reset the device to the same unknown initial settings as often as needed. Furthermore, different assumptions with respect to the amount of control the attacker has over the position and the Hamming weight of the induced faults are employed.

While most of the DFA work in the literature is targeted towards block and stream ciphers, only few researchers considered hash functions. This fact might seem logical at first glance because ciphers have a secret key input. On the other hand, hash functions are usually analyzed with known inputs. However, lately, DFA attacks have been considered on hash functions with secret inputs, which is the default setting for the hash function when used in a MAC scheme. In general, adapting DFA attacks against hash functions operating in the secret key setting is somewhat inherently more difficult than adapting it against stream and block ciphers. In fact, unlike block and stream ciphers where one assumes that only the input key material is unknown, when a hash compression function is used in a MAC setting, we consider all its inputs as secrets. Additionally, when a hash function is employed in a MAC scheme, there are usually several applications of the hash function and even a single application of the hash function uses a domain extender with occasionally a complex finalization stage.

Literature related to DFA attacks on hash functions include the analysis of SHACAL [22], which is the internal cipher of the SHA1 compression function. Later, the attack was adapted to deal with the feedforward which masks the output of the internal cipher and both the secret chaining value and message block were retrieved [16]. Afterwards, DFA was used to analyze HAS1-60 [18], and Grøstl [13]. In particular, in the analysis of Grøstl [13], the authors have used the one-bit fault model to invert the truncated output transformation, and to retrieve the input chaining value and message block of its permutation based compression function. In our attack on Streebog, we employ some of the concepts introduced in [13]. In the following section, we give the description of the Streebog hash function.

3 Specification of Streebog

Streebog outputs a 512 or 256-bit hash value, where half the last state is truncated when adopting the 256-bit output. The standard [1] specifies two different *IVs*, one for each output length. Streebog can process messages of length up to $2^{512} - 1$. The compression function iterates over 12 rounds of an AES-like cipher with an 8×8 byte internal state and a final round of key mixing. The compression function operates in Miyaguchi-Preneel mode and is plugged in Merkle-Damgård domain extender with a finalization step. The input message M is padded into a multiple of 512 bits by appending one followed by zeros. The message length for MD-strengthening is further included as an extra separate block, followed by a block of a checksum evaluated by the modulo 2^{512} addition of all message blocks as a finalization step. More precisely, let $n = \lfloor \frac{|M|}{512} \rfloor$ and the input message $M = x \| m_n \| \cdots \| m_1 \| m_0$, where $|M|$ is length of M, and x is a non complete or an empty block. Let $m_{n+1} = 0^{511-|x|} \| 1 \| x$, then the padded message $M = m_{n+1} \| m_n \| \cdots \| m_1 \| m_0$. As depicted in Figure 1, the compression function g_N is fed with three inputs: the chaining value h_{i-1}, a message block m_{i-1}, and the counter of bits hashed so far $N_{i-1} = 512 \times i$. Let h_i be a 512-bit

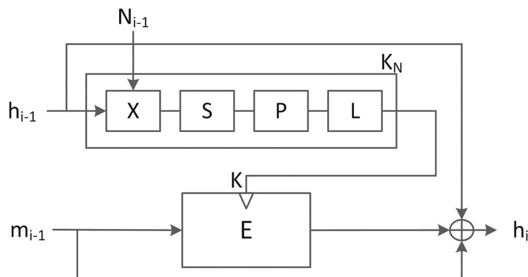

Fig. 1. Streebog's compression function g_N

chaining variable. The first state is loaded with the initial value IV and assigned to h_0. The hash value of M is computed as follows:

$$h_i \leftarrow g_N(h_{i-1}, m_{i-1}, N_{i-1}) \text{ for } i = 1, 2, .., n+2$$
$$h_{n+3} \leftarrow g_0(h_{n+2}, |M|, 0)$$
$$h(M) \leftarrow g_0(h_{n+3}, \Sigma, 0),$$

where $\Sigma = m_{n+1} + \cdots + m_1 + m_0$, $h(M)$ is the hash value of M, and g_0 is g_N with $N = 0$. As depicted in Figure 1, the compression function g_N consists of:

- K_N: a nonlinear whitening round of the chaining value. It takes a 512-bit chaining variable h_{i-1} and a counter of the bits hashed so far N_{i-1} and outputs a 512-bit key K.
- E: an AES-based cipher that iterates over the message for 12 rounds in addition to a finalization key mixing round. The cipher E takes a 512-bit key K and a 512-bit message block m as a plaintext. As shown in Figure 2, it consists of two similar parallel flows for the state update and the key scheduling.

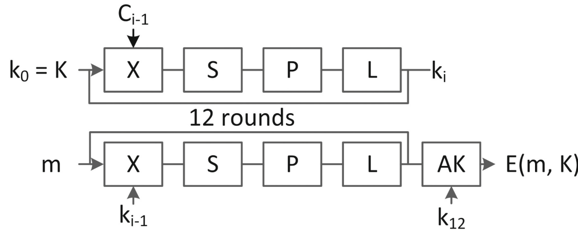

Fig. 2. The internal block cipher (E)

Both K_N and E operate on an 8×8 byte key state K. E updates an additional 8×8 byte message state M. In one round, a given state is updated by the following sequence of transformations:

- AddKey(X): XOR with either a round key, a constant, or the counter of bits hashed so far (N).
- SubBytes (S): A nonlinear byte bijective mapping.
- Transposition (P): Byte permutation.
- Linear Transformation (L): Row multiplication by an MDS matrix in GF(2).

Initially, state K is loaded with the chaining value h_{i-1} and updated by K_N as follows:

$$k_0 = L \circ P \circ S \circ X[N_{i-1}](K).$$

Now K contains the key k_0 to be used by the cipher E. The message state M is initially loaded with the message block m and $E(k_0, m)$ runs the key scheduling function on state K to generate 12 round keys, $k_1, k_2, .., k_{12}$, as follows:

$$k_i = L \circ P \circ S \circ X[C_{i-1}](k_{i-1}), \text{ for } i = 1, 2, .., 12,$$

where C_{i-1} is the ith round constant. The state M is updated as follows:

$$M_i = L \circ P \circ S \circ X[k_{i-1}](M_{i-1}), \text{ for } i = 1, 2, ..., 12.$$

The final round output is given by $E(k_0, m) = M_{12} \oplus k_{12}$. The output of g_N in the Miyaguchi-Preneel mode is $E(K_N(h_{i-1}, N_{i-1}), m_{i-1}) \oplus m_{i-1} \oplus h_{i-1}$ as shown in Figure 1. For further details, the reader is referred to [1].

Let M and K be (8×8)-byte states denoting the message and key state, respectively. The following notation is used throughout the paper:

- M_i: The message state at the beginning of round i.
- M_i^U: The message state after the U transformation at round i, where $U \in X, S, P, L$.
- $M_i[r,\ c]$: A byte at row r and column c of state M_i.
- $M_i[\text{row } r]$: Eight bytes located at row r of M_i state.
- $M_i[\text{col } c]$: Eight bytes located at column c of M_i state.

Same notation applies to K. In the following section, we give the details of our attack on Streebog.

4 Differential Fault Analysis Attack on Streebog

Our attack on the Streebog compression function aims to recover the secret input chaining value and message block. We proceed in a two-stage approach. In the first stage, given the compression function output, we recover the internal state of the last round of the internal cipher. Unlike the attack on the permutation based Grøstl, the knowledge of the internal state is not sufficient to recover the secret inputs since Streebog employs an internal cipher with secret round keys additions and not known constants. Hence, we adopt a second stage for the attack where we use the knowledge of the retrieved state from stage one to successfully recover one of the secret round keys, thus inverting the cipher and acquiring both the secret inputs of the compression function. In what follows, we give the definition of the used fault model and one of the Streebog Sbox properties that we are going to use in our attack.

Fault model: In our attack, we use the one-bit fault model which is used in [13,14]. For each fault injection, the attacker is assumed to be able to flip one bit in a given byte of the processed state whose position at row r and column c may be known or not. The practicality of this model has been demonstrated in [12], where the authors showed how tuning the laser injection parameters enables them to control with a 100% success rate the fault injection effect on a single bit: 0 to 1 or 1 to 0. Let M be a correctly computed state and M' a faulty state with a fault induced during its computation, then $M' = M \oplus \Delta$ where Δ is the

error state with only one non-zero byte. Formally, the employed fault model is defined as follows:

$$\Delta[r,c] = \begin{cases} \delta \in E & \text{for only one byte position,} \\ 0 & \text{otherwise,} \end{cases}$$

where $\Delta[r,c]$ denotes the error at the byte in row r and column c, and the set $E = \{0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80\}$.

For the Streebog Sbox, if x is a random input byte, $\delta x_i \in E$ for $i = 1,..,n$, $n \leq 8$ is a randomly chosen but distinct one bit faults, and

$$\delta y_i = S(x) \oplus S(x \oplus \delta x_i),$$

then x is uniquely identified by the values of δy_i only. In other words, the value of the Sbox input byte x can be recovered by observing n output differences δy_i corresponding to n one-bit distinct input faults. According to our exhaustive simulation, depending on x, the average number of unique fault insertions which affect different bits δx_i required to identify x varies between 2.071418 - 4.86861, and the overall average is $\overline{n} \approx 2.635$ faults per byte. For the case when fault insertions δx_i are not uniquely selected, the overall average is $\overline{n} \approx 3.077$ faults per byte. Another observation is that, for all x, there always exist two unique δx_i that would identify x. In what follows, we give the details of the first stage of the attack.

4.1 Stage One

In this stage, we recover the message state of the internal cipher of the compression function $g_N(h_{i-1}, m)$. We first observe the value of the correct compression function output $h_i = g_N(h_{i-1}, m)$. Afterwards and as depicted in Figure 3, we induce one-bit fault in a given byte of M_{11}, which is the input to the last round of the cipher, such that, $M'_{11} = M_{11} \oplus \Delta$, and Δ has only one non zero byte at position $[r,c]$. This fault results in a faulty h'_i that differs from the correct h_i state in one row as shown in Figure 3.

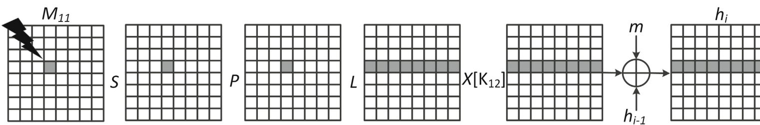

Fig. 3. Fault injection in the first stage of the attack

For a fault in a given byte position $M_{11}[r,c]$ we get the following two equations:

$$h_i = (X[k_{12}] \circ L \circ P \circ S(M_{11})) \oplus h_{i-1} \oplus m,$$
$$h'_i = (X[k_{12}] \circ L \circ P \circ S(M_{11} \oplus \Delta)) \oplus h_{i-1} \oplus m.$$

Since, $X, L,$ and P are bijective linear functions, we can propagate the difference at h_i backwards until the state after the Sbox as follows:

$$h_i \oplus h'_i = L \circ P \circ (S(M_{11}) \oplus S(M_{11} \oplus \Delta)),$$
$$P \circ L^{-1}(h_i \oplus h'_i) = S(M_{11}) \oplus S(M_{11} \oplus \Delta).$$

To this end, the difference state at the output of the Sbox of the last round of the internal cipher is given by $\Delta_{out} = S(M_{11}) \oplus S(M_{11} \oplus \Delta)$, where $\Delta_{out} = P \circ L^{-1}(h_i \oplus h'_i)$ and has only one non-zero value at row r and column c. Since, the substitution transformation operates on the state bytes independently, then the knowledge of the difference state Δ_{out} reveals the position $[r, c]$ of the induced fault. Accordingly, if we assume that we have enough faulty compression function outputs h'_i such that we know enough Δ_{out} states for each byte position in state M^S_{11}, then using the Sbox property presented in the previous section, we can recover the value of the entire state M_{11}.

4.2 Stage Two

Although in stage one, we are able to bypass the effect of the feedforward and recover state M_{11} of the internal cipher, we are still not able to invert the compression function and retrieve the secret input chaining value and message block. This is due to the fact that unlike other AES-based hash functions such as Grøstl which employs an internal permutation where known round constant additions are used, the Streebog internal cipher employs round key addition. These round keys are derived from the secret input chaining value and consequently they are not known to the attacker. For that reason, the knowledge of a round state of the compression function is not sufficient to invert it.

Our strategy in this stage is to recover the value of round key k_{11}, which is the key used in the round before the last one. Once we retrieve the value of k_{11}, we invert the key schedule to compute all previous round keys and finally, using the knowledge of the compression function counter N, the secret input chaining value is recovered. The employed approach depends on the knowledge of state M_{11} which we have recovered in the first stage of the attack. To recover the value of k_{11}, we first retrieve the value of state M_{10}, then evaluate $k_{11} = L \circ P \circ S(M_{10}) \oplus M_{11}$. Since, we know the value of M_{11}, we can inject one-bit faults in M_{10} and propagate the resulting differences in state h_i back to state M_{11} which is then used as h_i in stage one to recover M_{10}. As depicted in Figure 4, we inject one-faults in M_{10} and acquire the corresponding faulty h'_i which differs from the correct h_i in the whole state. The difference state $h_i \oplus h'_i$ is then propagated backward through the linear transformations until state M^S_{11}. Accordingly, the value of the faulty state M'_{11} is given by:

$$M'_{11} = S^{-1}(S(M_{11}) \oplus (P \circ L^{-1}(h_i \oplus h'_i))).$$

To this end, we get the difference at M_{11} which is then propagated backward to state M^S_{10}. The difference at M^S_{10} is the output difference of the Sbox at

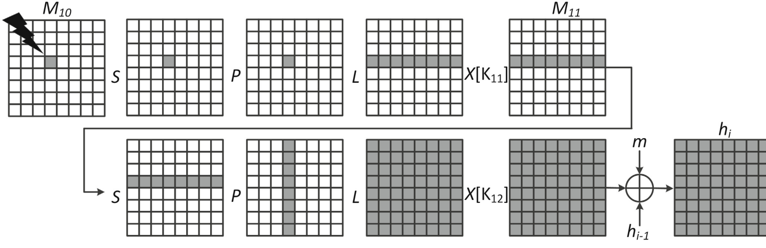

Fig. 4. Fault injection in the second stage of the attack

the eleventh round corresponding to the fault that we injected at state M_{10}. Consequently, this difference has only one active byte which reveals the byte position of the injected fault. The difference at state M_{10}^S is denoted by Δ_{out} and is given by:

$$\Delta_{out} = P \circ L^{-1}(M_{11} \oplus M'_{11}).$$

Now, if we repeat stage two such that we get enough Δ_{out} values for each of the 64 positions in state M_{10}^S, we can recover the value of state M_{10}. Consequently, the value of k_{11} is computed by the following equation:

$$k_{11} = (L \circ P \circ S(M_{10})) \oplus M_{11}).$$

In the sequel, using k_{11} we invert the key schedule and acquire all the round keys. Then by utilizing the knowledge of the compression function counter N within the hash function, the input chaining value h_{i-1} is recovered. Since, we only observe the output of the last compression function call of the hash function, we always assume that we are processing $g_0(h_{i-1}, \Sigma)$ so that $N = 0$. However, the attack can work on any g_N within the hash function as described in the following subsection. Finally, with the knowledge of the round keys and state M_{10}, we invert the message encryption and recover the input message block m of $g_N(h_{i-1}, m)$.

4.3 Extending the Attack to the Hash Function

The two-stage attack presented in the previous section works on a compression function that one can observe the effect of the induced fault on its output. When Streebog is used in various MAC applications, full hash function application is used and, as depicted in Figure 5, one can only observe the output $H(M)$ of the last compression function call $g_0(h_{t+1}, \Sigma)$ of the hash function. Accordingly, to be able to retrieve the inputs of the previous compression function, we first launch the two-stage attack on $g_0(h_{t+1}, \Sigma)$. Because we observe both the correct and faulty values of $H(M)$, we can retrieve the values of Σ and h_{t+1}. To attack $g_0(h_t, |M|)$, the first stage of the attack requires the difference at h_{t+1} in addition to its value which cannot be deduced from observing the faulty $H'(M)$.

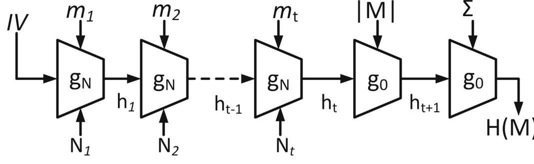

Fig. 5. The Streebog iterated hash function

This requirement can be fulfilled with a precomputed table T_1 for all the possible differences at h_{t+1} that result from injecting any fault at all the 64 byte positions of state M_{11} and their corresponding faulty $H'(M)$. Building this table is quite feasible for the fact that whatever the value of the induced fault at M_{11}, as depicted in Figure 3, each byte position at M^S_{11} may have up to 255 difference values which linearly maps to 255 one row differences Δh_{t+1} at state h_{t+1}. Accordingly, for each byte position in M^S_{11}, we linearly propagate the 255 possible differences forward to get Δh_{t+1}. Using our knowledge of the values of h_{t+1} and Σ, we evaluate the faulty $H'(M)$ corresponding to each difference. Finally, table T_1 will have 64×255 pairs of Δh_{t+1} and their corresponding $H'(M)$. Consequently, table T_1 enables us to complete the first stage of our attack because, when we inject a fault in M_{11}, the value of Δh_{t+1} corresponding to the resulting observed faulty $H'(M)$ is obtained from T_1. This step allows the recovery of the value of state M_{11} of $g_0(h_t, |M|)$.

The second stage of our attack requires the knowledge of the difference ΔM_{11} at state M_{11}. As depicted in Figure 4, a fault at a given position in M_{10} may have up to 255 difference after the Sbox which internally linearly map to 255 one row difference ΔM_{11} at state M_{11}. Since, we already know the value of state M_{11} from the previous step, we can get the corresponding 255 output differences after the Sbox at state M^S_{11} and linearly propagate them to get the full active state differences Δh_{t+1} at state h_{t+1}. Similar to the previous step, we build a second table, T_2 with all the 64×255 differences Δh_{t+1} and their corresponding $H'(M)$. This table allows us to finish stage two of our DFA and recover the values of h_t and $|M|$ of $g_0(h_t, |M|)$. The knowledge of $|M|$ reveals the number of the processed message blocks and accordingly the number of compression function calls and their corresponding counter values. Finally, we repeat the previous two-steps for each compression function and hence invert all of the compression function calls within the iterated hash function and retrieve all their secret inputs. Although we consider the 512-bit version of the hash function in our 2-stage attack, it also works on the 256-bit version where the last four rows of the last compression function are truncated. We only have to add an initial stage that deals with the truncation. We utilize the fact that the position and value of a single byte difference in a given row can be uniquely identified from the knowledge of the difference in any two bytes in the same row after the linear transformation (*cf.* Lemma 3 in [13]). In the added initial stage, we retrieve half of the state of the last round. Then, in stage one of our attack, we recover the whole state in the round before the last one with the knowledge of half of the difference state after the linear transformation, then continue with the rest of the attack.

5 DFA on Streebog in Different MAC Settings

One of the prospective applications of the new Russian standard is using it in MAC schemes. Despite the fact that both the simple prefix and the secret-IV MACs [27] are vulnerable to length extension attacks, Streebog is by design not vulnerable to length extension attacks due to its finalization stage. This property may tempt users to adopt one of the simpler MAC constructions. Indeed, the designers of the NIST SHA-3 hash function, Keccak [8,11] state on their website that since Keccak is not vulnerable to length extension attacks, it does not need HMAC and propose that MAC computation can be done by concatenating the key with the message [20]. Accordingly, in what follows, we consider Streebog in both the simple and standardized MAC settings, and show how our attack can be used to recover the secret MAC key.

Simple prefix/Secret-IV MACs: As depicted in Figure 6, in the simple prefix MAC, the secret key is used as the first message block of the processed message in the iterative construction of the hash function. More formally, $MAC(M) = H(K||M)$. On the other hand, in the secret-IV MAC, the standard initial value is replaced by the secret key in the iterative construction of the hash function. More formally, $MAC(M) = H_K(M)$, where $H_K(M)$ is the keyed hash value of the message M using the secret key K as the IV. The knowledge of the authenticated message reveals its corresponding message blocks and accordingly their modular sum. We can retrieve the secret key of the simple prefix MAC using the two-stage DFA on the last compression function call. The attack recovers Σ which is the modular summation of all processed message blocks including the secret key. Accordingly, to recover the key, we simply subtract the summation of the known message blocks of the authenticated message from the retrieved Σ. As for secret-IV MAC, we use our DFA and invert the compression function calls until the first one with $N = 0$, the retrived chaining value is the secret key. In both schemes, if we do not know the authenticate message, we can easily retrieve the number of message blocks from $|M|$ and iterate the attack backwards until the compression function with $N = 0$ to recover the key.

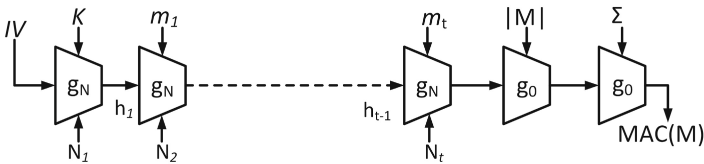

Fig. 6. Simple prefix MAC using Streebog

HMAC/NMAC: HMAC [7] is defined as:

$$HMAC(M) = H((K \oplus opad)||H((K \oplus ipad)||M)),$$

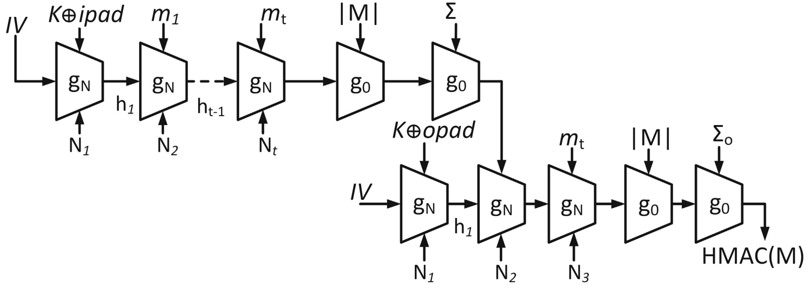

Fig. 7. HMAC using Streebog

where *opad* and *ipad* are known padding constants and H denotes a hash function call. The algorithm is standardized by ANSI, IETF, ISO and NIST, and is widely deployed in many Internet security protocols (e.g. SSL, SSH, IPSec). As depicted in Figure 7, the Streebog hash function is called twice. Our analysis works on the outer hash function call where $K \oplus opad$ is used as the first message block. Accordingly, our DFA is applied on the outer hash function and using the observed HMAC(M), we iterate the attack backwards to invert five compression function calls. The retrieved message block of the fifth backward compression function reveals the key value after xoring it with *opad*.

NMAC [7] employs two keys and is defined as:

$$NMAC(M) = H_{K_2}(H_{K_1}(M)),$$

where the keys are used as the initial values in the outer and inner hash function calls. The algorithm has a similar structure to HMAC but differs in that the first compression function call in both hash function calls in HMAC is omitted, and K_1 and K_2 are used as the IV for the following compression function call. Accordingly, if Figure 7 is to describe NMAC, we omit $g_N(IV, K \oplus ipad)$ and replace the resulting h_1 by K_1, and remove $g_N(IV, K \oplus opad)$ and use K_2 as the IV for the following compression function call. In the sequel, our attack works first to recover K_2 by iterating the two-stage attack backwards for four compression function calls. Afterwards, the retrieved message block corresponding to the output of the inner hash function is used to further recover K_1 from the inner hash function application.

6 Simulation Results

Since the attack has a very low complexity, we have simulated three scenarios of the attack on the compression function on an 8-core Intel i7 CPU running at 2.67GHz and the secret inputs were recovered in less than one minute. The scenarios vary in the assumptions of whether the attacker can control the injection of distinct faults and if the faulted byte position can be chosen or not. The provided average fault requirements are the result of running our simulation using

1000 different inputs to the compression function. As shown by our simulations, the number of required faults to retrieve 128 bytes in both stages depends on the assumptions used during fault injections. In what follows, we give the results of our simulation:

1. When the faults are selected distinctly and the byte position $[r, c]$ is chosen by the attacker, then one needs an average of 338 faults which is equivalent to an average of 2.635 faults per byte.
2. If we randomly induce non distinct one-bit faults and select the byte positions, then the attack requires an average of 394 fault injections in total with an average of 3.077 faults per byte.
3. In the case where both the byte position and the induced one-bit faults are randomly chosen, the attack requires an average of 1640 fault injections in total, and accordingly an average 12.807 fault per byte.

7 Conclusion

In this paper, we have investigated the security of the new Russian hash function standard GOST R 34.11-2012 with respect to differential fault analysis. In particular, we have proposed a two-stage approach that considers the compression function operating with secret inputs. Accordingly, using one-bit faults, the first stage of our attack bypasses the final feedforward and retrieves the internal state of the cipher used in the compression function. The second stage retrieves one of the round keys used in the cipher which enables the generation of the rest of the round keys and consequently, the input chaining value and message block are recovered. We have simulated the attack on the compression function with different assumptions regarding the control of the attacker over the induced faults and the faulted position. The results show that our two-stage attack requires between 338 and 1640 faults on average, depending on what are the assumptions of the employed fault model. Moreover, we have proposed a feasible precomputation step where we require two tables of size 2^{14} state each to enable the extension of the attack to the whole hash function. Finally, we have shown how our proposed approach is used to recover the secret MAC key when Streebog is used in simple prefix, secret-IV, HMAC, and NMAC settings. A naive approach to prevent our attack is to use spatial/temporal algorithm level redundancy and to disable the device output if the two produced MAC tags do not match. Another approach is to add parity bits to detect corruptions of the inner state registers and disable the device output if any of these parity checks is violated. Efficient fault analysis resistant implementations for Streebog, as well as for other hash functions deployed in MAC schemes, need to be addressed in future research.

Acknowledgment. The authors would like to thank the anonymous reviewers for their valuable comments and suggestions that helped improve the quality of the paper. This work is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

1. The National Hash Standard of the Russian Federation GOST R 34.11-2012. Russian Federal Agency on Technical Regulation and Metrology report (2012), https://www.tc26.ru/en/GOSTR34112012/GOST_R_34_112012_eng.pdf
2. AlTawy, R., Kircanski, A., Youssef, A.M.: Rebound attacks on stribog. In: Lee, H.-S., Han, D.-G. (eds.) ICISC 2013. LNCS, vol. 8565, pp. 175–188. Springer, Heidelberg (2014)
3. AlTawy, R., Youssef, A.M.: Integral distinguishers for reduced-round Stribog. Information Processing Letters 114(8), 426 (2014)
4. AlTawy, R., Youssef, A.M.: Preimage attacks on reduced-round stribog. In: Pointcheval, D., Vergnaud, D. (eds.) AFRICACRYPT. LNCS, vol. 8469, pp. 109–125. Springer, Heidelberg (2014), http://dx.doi.org/10.1007/978-3-319-06734-6_7
5. AlTawy, R., Youssef, A.M.: Watch your Constants: Malicious Streebog. IET Information Security (2015) (to appear)
6. Banik, S., Maitra, S., Sarkar, S.: A differential fault attack on the Grain family of stream ciphers. In: Prouff, E., Schaumont, P. (eds.) CHES 2012. LNCS, vol. 7428, pp. 122–139. Springer, Heidelberg (2012)
7. Bellare, M., Canetti, R., Krawczyk, H.: Keying hash functions for message authentication. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 1–15. Springer, Heidelberg (1996)
8. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Keccak sponge function family main document. Submission to NIST, Round 2 (2009)
9. Biham, E., Shamir, A.: Differential fault analysis of secret key cryptosystems. In: Kaliski Jr., B.S. (ed.) CRYPTO 1997. LNCS, vol. 1294, pp. 513–525. Springer, Heidelberg (1997)
10. Boneh, D., DeMillo, R.A., Lipton, R.J.: On the importance of checking cryptographic protocols for faults. In: Fumy, W. (ed.) EUROCRYPT 1997. LNCS, vol. 1233, pp. 37–51. Springer, Heidelberg (1997), http://dx.doi.org/10.1007/3-540-69053-0_4
11. Chang, S.-J., Perlner, R., Burr, W.E., Turan, M.S., Kelsey, J.M., Paul, S., Bassham, L.E.: Third-round report of the SHA-3 cryptographic hash algorithm competition (2012)
12. Courbon, F., Loubet-Moundi, P., Fournier, J.J.A., Tria, A.: Adjusting laser injections for fully controlled faults. In: Prouff, E. (ed.) COSADE 2014. LNCS, vol. 8622, pp. 229–242. Springer, Heidelberg (2014)
13. Fischer, W., Reuter, C.A.: Differential fault analysis on Grøstl. In: IEEE Workshop on Fault Diagnosis and Tolerance in Cryptography, pp. 44–54 (2012)
14. Giraud, C.: DFA on AES. In: Dobbertin, H., Rijmen, V., Sowa, A. (eds.) AES 2005. LNCS, vol. 3373, pp. 27–41. Springer, Heidelberg (2005)
15. Guo, J., Jean, J., Leurent, G., Peyrin, T., Wang, L.: The usage of counter revisited: Second-preimage attack on new russian standardized hash function. In: Joux, A., Youssef, A. (eds.) SAC 2014. LNCS, vol. 8781, pp. 195–211. Springer, Heidelberg (2014)
16. Hemme, L., Hoffmann, L.: Differential fault analysis on the SHA1 compression function. In: IEEE Workshop on Fault Diagnosis and Tolerance in Cryptography, pp. 54–62 (2011)
17. IETF. GOST R 34.11-2012: Hash Function, RFC6896 (2013)

18. Zou, J., Wu, W., Wu, S.: Cryptanalysis of the round-reduced GOST hash function. In: Lin, D., Xu, S., Yung, M. (eds.) Inscrypt 2013. LNCS, vol. 8567, pp. 307–320. Springer, Heidelberg (2014)
19. Kazymyrov, O., Kazymyrova, V.: Algebraic aspects of the russian hash standard GOST R 34.11-2012. In: CTCrypt, pp. 160–176 (2013), `http://eprint.iacr.org/2013/556`
20. Keccak team. Strengths of Keccak - Design and security, `http://keccak.noekeon.org/` (last accessed: December 2, 2014)
21. Kim, C.H., Quisquater, J.-J.: New differential fault analysis on AES key schedule: Two faults are enough. In: Grimaud, G., Standaert, F.-X. (eds.) CARDIS 2008. LNCS, vol. 5189, pp. 48–60. Springer, Heidelberg (2008)
22. Li, R., Li, C., Gong, C.: Differential fault analysis on SHACAL-1. In: IEEE Workshop on Fault Diagnosis and Tolerance in Cryptography, pp. 120–126 (2009)
23. Ma, B., Li, B., Hao, R., Li, X.: Improved cryptanalysis on reduced-round GOST and Whirlpool hash function. In: Boureanu, I., Owesarski, P., Vaudenay, S. (eds.) ACNS 2014. LNCS, vol. 8479, pp. 289–307. Springer, Heidelberg (2014)
24. Matyukhin, D., Rudskoy, V., and Shishkin, V. A perspective hashing algorithm. In: RusCrypto (2010) (in Russian)
25. Mendel, F., Pramstaller, N., Rechberger, C.: A (Second) preimage attack on the GOST hash function. In: Nyberg, K. (ed.) FSE 2008. LNCS, vol. 5086, pp. 224–234. Springer, Heidelberg (2008)
26. Mendel, F., Pramstaller, N., Rechberger, C., Kontak, M., Szmidt, J.: Cryptanalysis of the GOST hash function. In: Wagner, D. (ed.) CRYPTO 2008. LNCS, vol. 5157, pp. 162–178. Springer, Heidelberg (2008)
27. Preneel, B., van Oorschot, P.C.: On the security of iterated message authentication codes. IEEE Transactions on Information Theory 45(1), 188–199 (1999)
28. Skorobogatov, S., Anderson, R.: Optical fault induction attacks. In: Kaliski Jr., B.S., Koç, Ç.K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 2–12. Springer, Heidelberg (2003)
29. Tunstall, M., Mukhopadhyay, D., Ali, S.: Differential fault analysis of the Advanced Encryption Standard using a single fault. In: Ardagna, C.A., Zhou, J. (eds.) WISTP 2011. LNCS, vol. 6633, pp. 224–233. Springer, Heidelberg (2011)
30. Wang, Z., Yu, H., Wang, X.: Cryptanalysis of GOST R hash function. Information Processing Letters 114(12), 655–662 (2014)
31. Zou, J., Wu, W., Wu, S.: Cryptanalysis of the round-reduced GOST hash function. In: Lin, D., Xu, S., Yung, M. (eds.) Inscrypt 2013. LNCS, vol. 8567, pp. 307–320. Springer, Heidelberg (2014)