Possible lightweight modification of hashing algorithms GOST R 34.11-94 and GOST R 34.11-2012

S A Bystrevskii¹, A E Borshevnikov¹, M I Starodubov¹

¹Far Eastern Federal University, Vladivostok, Russky Island, 10 Ajax Bay, 10, 690922, Russia

E-mail serg.by.and97@mail.ru

Abstract. The paper discusses the development of a lightweight modification of the hashing algorithms GOST R 34.11-2012 and GOST R 34.11-94. The quality of optimization of the obtained algorithms and their safety are considered.

Keywords: Lightweight cryptography; Hash function; Complexity of the algorithm; Algorithm optimization

1. Introduction

Most modern algorithms for protecting information and, in particular, encryption, are designed for use in a PC as part of software systems without taking into account optimization at the hardware level. This fact makes it impossible to apply most of the existing cryptographic algorithms in devices with limited computing power, small volume and low power consumption. Cryptographic data protection techniques in low cost systems have become the backbone of lightweight cryptography.

Lightweight cryptography is gaining special relevance in the light of the development of the idea of the Internet of Things, which is a wireless self-configuring network between objects of various classes, examples of which are household appliances, vehicles, smart sensors and radio frequency identification (RFID) tags [1].

The goal of this work is to develop and analyze lightweight implementations of cryptographic hash functions standardized in the Russian Federation: GOST R 34.11-2012 and GOST R 34.11-94.

2. Development of lightweight hashing algorithms

Let us first consider the optimization of the GOST R 34.11-2012 algorithm.

First of all, it is worth reducing the block size of the hash function. Perhaps this will slow down the algorithm, but it will clearly reduce its size. Let the block size be 64 bits, let's call this algorithm Stribog-64 accordingly. Moreover, this block size should be throughout the entire algorithm, and not just cut at the end, as in Stribog-256. Let's consider the modified transformations for *Stribog-64.

The nonlinear bijective transformation π: V₈ → V₈ will not change. But it is worth replacing the permutation r: Z₆₄ → Z₆₄ with r: Z₁₆ → Z₁₆, we will set it below as an array r = (r(0), r(1), ..., r(16)): (0, 4, 8, 12, 1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15).

To convert l: V₆₄ → V₆₄, it is worth changing the constants, or rather, generating them, since they take up a lot of space. 64 constants, each 64 bits in size, totaling 4096 bits only for storing these constants.
constants. It makes sense to store only one constant, and generate the rest from it. Since the transformation \(I\) is used frequently in the algorithm, this will slow down the speed of the algorithm, so the generation should not be difficult.

Also, in order for the transformation result to be any vector from \(V_{64}\), it is necessary that the newly formed constants form the basis of the \(V_{64}\) space. This result can be achieved if we take \(0x8e20faa72ba0b470\) as the first number, and each next constant is obtained from the previous one by cyclic shift by any prime number less than 64. For definiteness, take the number 5, then the process of generating a constant vector from the standard can be expressed below as:

\[
A_i = \begin{cases} 
8e20faa72ba0b470, & \text{if } i = 0 \\
A_{i-1} < < 5, & \text{if } 1 \leq i \leq 63 
\end{cases}, 
\]

where \(A_i \in V_{64}\) are elements of the constant vector \(A\), defined in GOST R 34.11-2012.

The transformation \(X[k]: V_{512} \rightarrow V_{512}\) will be changed only in dimension, that is, it will become \(X[k]: V_{64} \rightarrow V_{64}\).

The transformation \(S: V_{512} \rightarrow V_{512}\) will accordingly become \(S: V_{64} \rightarrow V_{64}\), and will definitely be like:

\[
S(a) = S(a_7)||...||a_1||a_0) = \pi(a_7)||...||\pi(a_1)||\pi(a_0), 
\]

where \(a = a_7||...||a_0 \in V_{64}\), \(a_7, a_0 \in V_8\).

The permutation conversion \(P: V_{512} \rightarrow V_{512}\) is redefined as \(P: V_{64} \rightarrow V_{64}\) as follows:

\[
P(a) = P(a_{15})||...||a_1||a_0) = a_r(15)||...||a_r(1)||a_r(0), 
\]

where \(a = a_{15}||...||a_0 \in V_{64}\), \(a_{15}, a_0 \in V_4\), \(r\) is a new permutation.

The linear transformation \(L: V_{512} \rightarrow V_{512}\) becomes \(L: V_{64} \rightarrow V_{64}\) and will be equal to the new linear transformation \(L: V_{64} \rightarrow V_{64}\).

Thus, the basic transformation used in the Stribog algorithm was simplified. Now we need to define a one-step compression function.

It makes no sense to make major changes in it. But it's worth understanding the generation of thirteen round keys using twelve constants for the algorithm. These constants take up a lot of space. Each is 512 bits and thus they take up 6144 bits in total. In key generation, they are used only for modulo two addition, which is not justified memory consumption. It makes sense to leave one constant and use it only when adding modulo two with the first key. So now in the one-step compression function, instead of \(N\), some special vector will be used. But in order to have the effect of changes, this vector must change every run of the hash function. To achieve this goal, you can use some light pseudo-random number generator, for example, created on the basis of the linear shift register proposed in the cipher KATAN [2,3].

To implement the idea described above, we introduce a new transformation \(C_{RI_1,RI_2}: V_{64} \rightarrow V_{64}\), which we define as follows:

\[
C_{RI_1,RI_2}(N) = C_{RI_1,RI_2}(N_{63})||...||N_0) = N_{62}||...||N_{32}||RI_2 \oplus N_{31} \oplus N_{24} \oplus (N_{20} \& N_{15}) \oplus \\
(N_{3} \& N_0)||N_{30}||...||N_0||RI_1 \oplus N_{63} \oplus N_{56} \oplus (N_{52} \& N_{47}) \oplus (N_{35} \& N_{33}), 
\]

where \(RI_1, RI_2 \in V_1\) are stochastic bits obtained from the hash, \(N = N_{63}||...||N_0 \in V_{64}\), \(N_{63},...,N_0 \in V_1\).

This function consists of two linear shift registers that write new data to each other. The peculiarity is that it uses stochastic bits. The bits \(RI_1, RI_2\) will be the last and first bits of the hash, respectively. And the first value of \(N\) will be \(b1085bd1a1cdacae9\).

Now let's review the transformations of the one-step hash function.

Key generation is now set like this:

\[
K_i = \begin{cases} 
K, & \text{for } i = 1 \\
\text{LPS(K}_{i-1}), & \text{for } i = 2, ..., 13 
\end{cases}, 
\]

where \(K \in V_{64}\) is the value for the first key.

The transformation \(E: V_{512} \times V_{512} \rightarrow V_{512}\) will remain the same, except for the change in dimension \(E: V_{64} \times V_{64} \rightarrow V_{64}\):

\[
E(K, m) = X[K_{13}]LPSX[K_{12}]...LPSX[K_1](m),
\]
where $K, m \in V_{64}$, $K_1, ..., K_8$ are generated keys, where the value of the first key is $K$.

Compression function $g_N : V_{512} \times V_{512} \rightarrow V_{512}$, also changes only in dimension $g_N : V_{64} \times V_{64} \rightarrow V_{64}$:

$$g_N(h, m) = E(LPS(h \oplus N), m) \oplus h \oplus m \quad (7)$$

where $h, m, N \in V_{64}$.

Finally, by redefining all the functions, you can define the algorithm itself. The algorithm for generating the value of the hash function Stribog-64 is defined as follows.

Input: $M \in V^*$.
1. $h = 0^{64}$.
2. $N = \text{Vec}_{64}((0xb1085bda1ecadae9), \Sigma = 0^{64}$.
3. If the condition $|M| \geq 64$ is satisfied, then go to step 4, otherwise go to step 10.
4. Calculate $M', m; M = M' || m$, where $m \in V_{64}$.
5. $h = g_N(h, m)$.
6. $N = C_{h_{63}, h_0}(N)$, where $h = h_{63} || ... || h_0, h_{63}, ..., h_0 \in V_1$.
7. $\Sigma = \text{Vec}_{64}(\text{Int}_{64}(\Sigma) \leftarrow \text{Int}_{64}(m))$.
8. $M = M'$.
9. Go to step 3.
10. $m = 0^{64-|M|} || 1 || M$.
11. $h = g_N(h, m)$.
12. $N = C_{h_{63}, h_0}(N)$, where $h = h_{63} || ... || h_0, h_{63}, ..., h_0 \in V_1$.
13. $\Sigma = \text{Vec}_{64}(\text{Int}_{64}(\Sigma) \leftarrow \text{Int}_{64}(m))$.
14. $h = g_N(h, \Sigma)$.
Output: $h$ – hash value.

Now let's start considering the algorithm for generating the value of the hash function GOST R 34.11-94.

If the block size is reduced, then this algorithm meets the size requirements for lightness. Therefore, it is necessary to increase the performance of this algorithm.

Since the algorithm uses the Magma cipher, the performance of the GOST R 34.11-94 algorithm will directly depend on the performance of the Magma cipher. This problem can be solved by reducing the number of rounds in the encryption algorithm. According to research, the number of rounds can be reduced to sixteen without a strong loss of cryptographic strength.

In what follows, the transformation $E_K : V_{64} \rightarrow V_{64}$ will mean the sixteen-round cipher "Magma" with the key $K \in V_{256}$.

In the algorithm GOST R 34.11-94, we will reduce the block length to 64 bits. And we will redefine transformations accordingly.

First, let's define an algorithm for generating a key. Here we have to change something, since we need to collect one key from 64 bits with a length of 256 bits. We will use the same principle as in the original algorithm and generate four 64-bit keys, and then glue them into one.

We redefine the transformation $A : V_{256} \rightarrow V_{256}$ as $A : V_{64} \rightarrow V_{64}$. This transformation is a linear register shift and can be expressed as:

$$A(X) = A(x_3 || x_2 || x_1 || x_0) = (x_0 \oplus x_1) || x_3 || x_2 || x_1, \quad (8)$$

where $X = x_3 || x_2 || x_1 || x_0 \in V_{64}, x_3, x_2, x_1, x_0 \in V_{16}$.

We redefine the supporting permutation $P : V_{256} \rightarrow V_{256}$ as $P : V_{64} \rightarrow V_{64}$. Let's set it as follows:

$$P(X) = P(x_0 || ... || x_1) = x_{\varphi(0)} || ... || x_{\varphi(1)} \quad (9)$$

where $X = x_8 || ... || x_1 \in V_{64}, x_8, ..., x_1 \in V_{8}; \varphi(i + 1 + 4(k - 1)) = 2i + k, i = 0, ..., 3, k = 1, 2$.

Now, let's start redefining the key generation algorithm.

Key generation is a 256-bit key generation from vectors $M, H \in V_{64}$. We define the algorithm as follows.
Input: $M, H \in V_{256}$.
1. $C_2 = C_4 = 0^{64}$, $C_3 = 1^{16}0^{24}1^{16}0^{8}$.
2. $i = 1$, $U = H, V = M$.
3. $W = U \oplus V$, $K_1 = P(W)$.
4. $i = i + 1$.
5. If the condition $i = 5$ is satisfied, then go to step 9, otherwise go to step 6.
6. $U = A(U) \oplus C_i, V = A(A(V))$.
7. $W = U \oplus V$, $K_1 = P(W)$.
8. Go to step 4.
9. $K = K_4|K_3|K_2|K_1$.
Output: $K \in V_{256}$ – generated key.

Now consider mixing transformation $\Psi: V_{256} \rightarrow V_{256}$, which needs to be redefined as $\Psi: V_{64} \rightarrow V_{64}$.
Let's set it up as follows:
\[
\Psi(X) = \Psi(x_0) ... x_1 = x_1 \oplus x_3 \oplus x_7 \oplus x_8 | x_0 | ... | x_2, \tag{10}
\]
where $X = x_0 | ... | x_1 \in V_{64}, x_0, ... , x_1 \in V_8$.

Since the algorithm uses the function $\Psi(X)$ 61 times and 12 times in a row, it makes sense to calculate the values $\Psi^{61}(X)$ and $\Psi^{12}(X)$ in advance, rather than using a loop in the one-step compression function.
The composition of these functions can be set as follows:
\[
\Psi^{12}(X) = \Psi^{12}(x_0) ... x_1 = x_3 \oplus x_7 \oplus x_8 \oplus x_2 \oplus x_6 \oplus x_0 \oplus x_7 \oplus x_3 \oplus x_8 \oplus x_2 \oplus x_4 \oplus x_5 \oplus x_6 \oplus x_7 \oplus x_1 \oplus x_3 \oplus x_4 \oplus x_5 \oplus x_6 \oplus x_7 \oplus x_1 \oplus x_3 \oplus x_4 \oplus x_5 \oplus x_6 \oplus x_7 \oplus x_1 \oplus x_3 \oplus x_4 \oplus x_5 \oplus x_6 \oplus x_7 \oplus x_1 \oplus x_3 \oplus x_4 \oplus x_5 \oplus x_6 \oplus x_7 \oplus x_1 \oplus x_3 \oplus x_4 \oplus x_5 \oplus x_6 \oplus x_7, \tag{11}
\]
\[
\Psi^{61}(X) = \Psi^{61}(x_0) ... x_1 = x_2 \oplus x_3 \oplus x_5 \oplus x_6 \oplus x_8 \oplus x_3 \oplus x_4 \oplus x_5 \oplus x_6 \oplus x_8 \oplus x_3 \oplus x_4 \oplus x_5 \oplus x_6 \oplus x_7 \oplus x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_5 \oplus x_6 \oplus x_7 \oplus x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_5 \oplus x_6 \oplus x_7 \oplus x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_5 \oplus x_6 \oplus x_7 \oplus x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_5 \oplus x_6 \oplus x_7, \tag{12}
\]
where $X = x_0 | ... | x_1 \in V_{64}, x_0, ... , x_1 \in V_8$.

Now we need to define a one-step compression function. It makes no sense to make major changes here, other than to reduce the use of mixing transforms, since the number of mixing blocks has changed.
Finally, given the above, let's redefine the one-step hashing function as:
\[
\chi(M, H) = \Psi^{61}(H \oplus \Psi(M \oplus \Psi^{12}(E_k(H)))), \tag{13}
\]
where $M, H \in V_{64}$.
The algorithm for calculating the value of the hash function $h$ will change only by the dimensions of the blocks and the use of a new one-step hashing function. Specifically, the algorithm is defined as follows.

Input: $M, H \in V^*$, $H \in V_{64}$
1. $\Sigma = 0^{64}$, $L = 0^{64}$.
2. If the condition $|M| > 64$ is satisfied, then go to step 3, otherwise go to step 8.
3. Calculate $M_p, M_s$: $M = M_p || M_s$, where $M_s \in V_{64}$.
4. $H = \chi(M_s, H)$.
5. $\Sigma = \Sigma^+ || M_s$, $L = L^+ | M_s$.
6. $M = M_p$.
7. Go to step 2.
8. $M' = 0^{64-|M|} || M$.
9. $\Sigma = \Sigma^+ || M'$, $L = L^+ | M$.
10. $H = \chi(M', H)$.
11. $H = \chi(L, H)$.
12. $H = \chi(\Sigma, H)$.
Output: $H$ – hash value.
3. Optimization score

Let's consider the optimization of the GOST R 34.11-2012 algorithm.

First, let's look at the total amount of memory saved. Saved 1728 bits of dynamic memory and 10240 bits of static memory. This is a very large volume and if we translate this into the size of the algorithm, then about 65024 gate equivalent (GE) will be released.

In terms of speed, not such results have been achieved. However, a plus can be noted in that the processing speed of short data is several times higher than that of the original. Thus, if the input messages are less than 64 bytes, then the average data processing speed is twice that of the original. Note that if the input messages are less than 16 bytes in length, then the processing speed is 60000000 bytes per second, when the original has 100000000 bytes per second. This is of course short data, but such data includes passwords, which are usually 8 to 16 bytes long. Also, the flogs are constantly compared, so this algorithm is very suitable for them. However, if we take a conditionally infinite string as input, then for the original the data processing speed is 250000000 bytes per second, while the new algorithm has 1250000000 bytes per second. That is, twice as slow.

The new algorithm meets the requirements of lightness in that it has a small size and a high speed of processing short input data. But hash function is not suitable for other purposes, since it has a low processing speed for big data.

Let us evaluate the optimization of the GOST R 34.11-94 algorithm.

First, let's estimate the data processing speed. It increased greatly, but, as for the previous algorithm, it turned out to be different for different lengths. If we consider conventionally infinite lines, then the speed of the original algorithm is 3000000000 bytes per second, while the speed of the new algorithm is 6000000000 bytes per second. That is, even for a conditionally infinite string, the new algorithm is twice as efficient in time as the original.

If we consider the input data of the order of 16 bytes, then the speed of the original algorithm is 8500000000 bytes per second, and the new algorithm is 3830000000 bytes per second. That is, for small input data, the speed of the algorithm has increased by about 4.5 times.

Now let's look at optimization for the amount of stored information. 768 bits of dynamic memory was saved and 128 bits of static memory were added, thus reducing the algorithm size by about 6272 GE.

As a result, we can say that the optimization of this algorithm is successful and is suitable both for processing long messages and for processing a large number of short messages. It can also be noted that, although not by a little, the amount of stored information has decreased, which still slightly affects the size of the hardware implementation of the algorithm and, accordingly, the cost.

4. Safety assessment

It remains only to consider the safety of the developed algorithms. We have considered algorithms for generating a hash, so we will consider the possibility of selecting collisions and modifying data, as well as finding a second preimage.

It is worth adding a number of statements about hash functions that were not covered in the first chapter. First, if the hash function is built on a one-step compression function $f$, then the collision resistance of the function $f$ implies the collision resistance of the entire hash function. The second statement is that if the function is resistant to collisions, then it is resistant to finding the second preimage. The third statement - the condition of resistance to finding the inverse image of a function is stronger than the condition of collision resistance.

Let's start by considering the cryptographic strength of the GOST R 34.11-2012 algorithm [4].

Before starting the research, we note that the algorithm has an avalanche effect, as a result of which, even with small changes in the initial data, serious changes occur in the final result. It is also worth noting that the end result is equally likely. Consider the estimate of the complexity of finding the second preimage. Since the algorithm is equally probable, the probability that a random input value will have a
specific desired hash is \( \frac{1}{|Y|} = \frac{1}{2^{64}} \). The probability that out of \( n \) randomly taken messages at least one of the messages will coincide in the hash value with the original is equal to:

\[
P = 1 - \left( \frac{2^{64} - 1}{2^{64}} \right)^n = 1 - \left( 1 - \frac{1}{2^{64}} \right)^n \approx 1 - e^{-\frac{n}{2^{64}}}
\] (14)

Thus, in order to find the second preimage with a given probability \( P \), it is necessary to take the following amount of input data:

\[
n = -2^{64} \ln(1 - P)
\] (15)

That is, in order to find the second preimage with a probability of 0.5, it is necessary to take approximately \( n = 2^{63} \) input data. And in order to almost guaranteed to find the second preimage, you need to take \( n = 2^{70} \) input data.

Consequently, the algorithm lost its resistance to the search for the second preimage, due to the decrease in the hash block, but it still remained acceptable.

Next, we will consider the stability of this algorithm to the “Birthdays” paradox. Following the paradox of "Birthdays", we can say that to find a collision with a probability of 0.5, you need to take \( 2^{32} \) inputs. Hence the algorithm is not very collision resistant. This is a small hash length problem.

All of the above applies only to hash length, but not to the algorithm itself. The security of the algorithm itself has great cryptographic strength, since in the lightweight version, almost everything remains from the original.

In general, the functions used in the algorithm make it quite cryptographically strong. But the algorithm is weak for collision detection and a list attack due to a small hash value. Again, this was expected and justified, because lightweight algorithms are used on low-power devices and, as a rule, do not use expensive information and the cost of cracking them is more expensive than the information itself, even with such a low cryptographic strength. In general, this algorithm demonstrates the main advantage of constructing a lightweight algorithm based on the existing one - it is high cryptographic strength due to proven functions.

Now let's move on to studying the safety of the optimized algorithm GOST R 34.11-94 [5,6].

To begin with, we note that the algorithm has a hash dimension of 64 bits. The Stribog-64 algorithm, considered in the previous paragraph, has the same hash dimension. This means that the entire estimate of the optimized GOST R 34.11-94 algorithm with respect to the length of its hash will be the same as for the Stribog-64 algorithm.

Similar to the previous considered algorithm, this algorithm is not sufficiently cryptographically strong, since it has a small hash length, but is resistant to attacks of search for a preimage. In principle, this robustness is sufficient for lightweight algorithms. Perhaps, for further work on this topic, we can consider blocks of greater length than 64 bits to increase the cryptographic strength of the hash function.

These algorithms turned out to be not resistant to collision search, but rather resistant to searching for images.

5. Conclusion

In this paper, issues related to lightweight cryptography, methods for the implementation of lightweight hashing algorithms, and methods for their analysis were considered.

These algorithms have shown good results in terms of lightness. That is, these algorithms require less stored information and perform operations faster than the originals. At the same time, the cryptographic strength of these algorithms turned out to be low, but sufficient for lightweight algorithms.

References

[1] Weis S A, Sarma S E, Rivest R L and Engels D W 2004 Security and privacy aspects of low-cost radio frequency identification systems (In Security in Pervasive Computing 2003, Lecture Notes in Computer Science) v 2802 pp 201-212

[2] Leander G, Paar C, Poschmann A and Schram K 2007 New Lightweight DES Variants (FSE 2007, LNCS) A. Biryukov (Ed.) 4593 pp 196–210
[3] Bard G V, Courtois N T, Nakahara J, Sepehrdad P and Zhang B 2010 Algebraic, AIDA/Cube and Side Channel Analysis of KATAN Family of Block Ciphers (INDOCRIPT 2010, LNCS) G. Gong and K.C. Gupta (Eds.) 6498 pp 176–196

[4] Sedov G K 2015 The security of GOST R 34.11-2012 against preimage and collision attacks (Mathematical Aspects of Cryptography) vol 6 pp 79–98

[5] Levin V Y 2011 Increasing the security of hash functions (J Math Sci) 172 pp 734–739

[6] Poschmann A, Ling S and Wang H 2010 256 Bit Standardized Crypto for 650 GE GOST Revisited (In CHES 2010, Lecture Notes in Computer Science) v 6225 pp 219-233