

ESIEA - CVO

Test Plan GostCrypt

Author:

Sebastiaan GROOT



June 13, 2014

Contents

1	Introduction	1
2	Static tests	2
3	Unit tests	4

1 Introduction

GostCrypt is a cryptography and steganography application based on TrueCrypt 7.1a. The main difference between TrueCrypt 7.1a and GostCrypt is the replacement of the existing cipher suite with the GOST 28147-89 block cipher, the removal of the SHA-512 and RIPEMD-160 hash functions and the introduction of the GOST R 34.11-94 and GOST R 34.11-2012 hash functions.

In order to publish a stable release of GostCrypt, this document provides a number of static code tests, as well as unit tests that can expose bugs in critical (cryptographic) parts of the application. Bugs encountered during the testing phase can then be resolved before publishing a first public release version of the application.

A large part of the non-cryptographic code in TrueCrypt 7.1a has already been audited by the Open Crypto Audit Project (2014) [1]. Because of this audit, only code that was modified or added for GostCrypt will be tested in an effort to reduce the scope of the testing phase to only include untested code.

2 Static tests

When the GostCrypt application is compiled, the compiler automatically runs various static tests on the source code. This is mainly used to detect syntax errors and provide the programmer with quick feedback about potential errors in his or her code. Compilers do not run very comprehensive static tests on the source code however, as they have to balance code testing with compilation speed. Several tools have been developed that provide a more comprehensive static testing environment. This class of tools is usually referred to as “lint” tools.

The application used for static analysis is the “lint” tool “splint”, a static analysis tool for C developed by the University of Virginia, Department of Computer Science [2] with a focus on detecting security-related errors.

The use of lint tools in static code analysis of large projects creates a problem. Lint tools give warnings for every piece of code that can potentially be dangerous. In many cases, warnings given by a lint tool do not translate to actual defects in the code. The following example demonstrates this behaviour:

```
unsigned char x = 0;
```

In this example, variable x is of the type *unsigned char*, which can contain any value between 0 and 255 (both inclusive) and it is initialized at 0. However, as splint interprets x as a non-numeric variable intended to store characters, it gives the following warning:

```
/*
splintexample.c(3,19): Variable j initialized to type int, expects
    unsigned char: 0
Types are incompatible. (Use -type to inhibit warning)
*/
```

Because of this, a first selection is made on the splint output by the code reviewer before reporting potential issues in the resulting report.

Table 1 shows the source files that will be included in the static analysis phase. This selection only includes source files that contain code that was modified for GostCrypt.

Source file	Description
Mount\Mount.c	Main code for the GostCrypt Mount application.
Format\Format.c	In place encryption algorithms.
Driver\NtDriver.c	Entry point for the GostCrypt driver.
Common\Xts.c	XTS mode of operation algorithms.
Common\Volumes.c	Code concerning volume headers.
Common\Tests.c	Built-in automatic tests for cryptographic algorithms.
Common\Random.c	Pseudo-random number generator.
Common\Pkcs5.c	Key derivation algorithms.
Common\Language.c	Language-pack loading code.
Common\EncryptionThreadPool.c	Crypto interface for multi-threaded encryption.
Common\Dlgcode.c	Windows dialog related code.
Common\Crypto.c	Interface to cryptographic algorithms.
Crypto\GostCipher.c	GOST 28147-89 implementation.
Crypto\GostHash.c	GOST R 34.11-94 implementation.
Crypto\Stribog.c	GOST R 34.11-2012 implementation.

Table 1: Source files included for static analysis

3 Unit tests

This chapter describes the unit tests that will be performed during the testing phase of GostCrypt. Unit tests are designed to test the functionality of functions or algorithms. Each unit test prepares the required input for the function it is testing, executes the function and then asserts that the function output is as expected. This is sometimes done by preparing the required arguments for a function and checking its return values, but it might also involve setting up global or external resources and asserting them afterwards for functions with side-effects.

The following subsections contain the different unit tests that are part of the test phase of GostCrypt, grouped by source file.

Common\Xts.c

The modified encryption and decryption variant of the XTS algorithm for block ciphers with a block size of 64-bit need to be tested.

There are two reasons why a complete unit test on the XTS encryption and decryption functions is infeasible. First of all, the output generated by these functions is dependent on the used block cipher. Secondly, this modified version of the XTS algorithm has no known test vectors. As we cannot derive specific properties from the output (by design), the unit tests will focus on small sections of the algorithm at a time, comparing the output to the output of the same sections in the 128-bit XTS algorithms.

In order to keep the used block cipher the same when comparing the XTS algorithm output, the GOST 28147-89 block cipher is used in both the 64-bit and 128-bit version of the algorithm during testing. Note that this is undesirable in normal operation, as using a 64-bit block cipher with the 128-bit XTS mode of operation results in half the plaintext not being transformed into ciphertext.

Setup The first four tests assert that the initial variables are set up properly before the encryption or decryption begins. The values of the 64-bit functions use the results of the 128-bit function tests as reference.

Scope	
Function	EncryptBufferXTS8Byte
Lines	80 - 114
Input variables	
buffer	(hexidecimal) 4c 6f 72 65 6d 20 69 70 73 75 6d 20 64 2e 2e 2e
length	16
startDataUnitNo	0
startCipherBlockNo	0
ks	key: (hexidecimal) 54 4f d7 b6 90 b3 5a cc 9f 7c e5 c5 c8 a5 04 79 82 41 0e 89 09 23 f2 c6 cd a2 d8 8c 42 4b be ef
ks2	key: (hexidecimal) a2 0e 0e f5 5b 38 52 46 0e 57 e7 24 20 af 9b bc 00 c3 fc 94 f8 23 2c 21 37 ea 30 55 34 fe eb df
Expected output variables	
whiteningValue	Equal to the first two bytes of the whiteningValue of the 128-bit XTS algorithm.
blockCount	Double the value of blockCount of the 128-bit XTS algorithm.
endBlock	blockCount - 1

Table 2: Unit Test: Setup EncryptBufferXTS8Byte

Scope	
Function	EncryptBufferXTSNonParallel
Lines	320 - 360
Input variables	
buffer	(hexidecimal) 4c 6f 72 65 6d 20 69 70 73 75 6d 20 64 2e 2e 2e
length	16
startDataUnitNo	0
startCipherBlockNo	0
ks	key: (hexidecimal) 54 4f d7 b6 90 b3 5a cc 9f 7c e5 c5 c8 a5 04 79 82 41 0e 89 09 23 f2 c6 cd a2 d8 8c 42 4b be ef
ks2	key: (hexidecimal) a2 0e 0e f5 5b 38 52 46 0e 57 e7 24 20 af 9b bc 00 c3 fc 94 f8 23 2c 21 37 ea 30 55 34 fe eb df
Expected output variables	
whiteningValue	(hexidecimal) cb 26 f8 f0 39 5a 28 93 00
blockCount	1
endBlock	blockCount - 1

Table 3: Unit Test: Setup EncryptBufferXTSNonParallel

Scope	
Function	DecryptBufferXTS8Byte
Lines	454 - 488
Input variables	
buffer	(hexidecimal) 8d c9 fa 9c 5e 9a 45 5d 00 00 00 00 00 00 00 00
length	16
startDataUnitNo	0
startCipherBlockNo	0
ks	key: (hexidecimal) 54 4f d7 b6 90 b3 5a cc 9f 7c e5 c5 c8 a5 04 79 82 41 0e 89 09 23 f2 c6 cd a2 d8 8c 42 4b be ef
ks2	key: (hexidecimal) a2 0e 0e f5 5b 38 52 46 0e 57 e7 24 20 af 9b bc 00 c3 fc 94 f8 23 2c 21 37 ea 30 55 34 fe eb df
Expected output variables	
whiteningValue	Equal to the first two bytes of the whiteningValue of the 128-bit XTS algorithm.
blockCount	Double the value of blockCount of the 128-bit XTS algorithm.
endBlock	blockCount - 1

Table 4: Unit Test: Setup DecryptBufferXTS8Byte

Scope	
Function	DecryptBufferXTSNonParallel
Lines	689 - 722
Input variables	
buffer	(hexidecimal) 8d c9 fa 9c 5e 9a 45 5d 00 00 00 00 00 00 00 00
length	16
startDataUnitNo	0
startCipherBlockNo	0
ks	key: (hexidecimal) 54 4f d7 b6 90 b3 5a cc 9f 7c e5 c5 c8 a5 04 79 82 41 0e 89 09 23 f2 c6 cd a2 d8 8c 42 4b be ef
ks2	key: (hexidecimal) a2 0e 0e f5 5b 38 52 46 0e 57 e7 24 20 af 9b bc 00 c3 fc 94 f8 23 2c 21 37 ea 30 55 34 fe eb df
Expected output variables	
whiteningValue	(hexidecimal) cb 26 f8 f0 39 5a 28 93 00
blockCount	1
endBlock	blockCount - 1

Table 5: Unit Test: Setup DecryptBufferXTSNonParallel

Encryption The following two unit tests observe the encryption of a single block in the 64-bit and 128-bit versions of the XTS algorithm. Again, the 128-bit version is used as a reference

for the 64-bit version.

Scope	
Function	EncryptBufferXTS8Byte
Lines	117 - 166
Input variables	
buffer	(hexidecimal) 4c 6f 72 65 6d 20 69 70 73 75 6d 20 64 2e 2e 2e
length	16
startDataUnitNo	0
startCipherBlockNo	0
ks	key: (hexidecimal) 54 4f d7 b6 90 b3 5a cc 9f 7c e5 c5 c8 a5 04 79 82 41 0e 89 09 23 f2 c6 cd a2 d8 8c 42 4b be ef
ks2	key: (hexidecimal) a2 0e 0e f5 5b 38 52 46 0e 57 e7 24 20 af 9b bc 00 c3 fc 94 f8 23 2c 21 37 ea 30 55 34 fe eb df
Expected output variables	
buffer	Equal to the value of buffer in the 128-bit XTS algorithm.

Table 6: Unit Test: Encryption EncryptBufferXTS8Byte

Scope	
Function	EncryptBufferXTSNonParallel
Lines	364 - 423
Input variables	
buffer	(hexidecimal) 4c 6f 72 65 6d 20 69 70 73 75 6d 20 64 2e 2e 2e
length	16
startDataUnitNo	0
startCipherBlockNo	0
ks	key: (hexidecimal) 54 4f d7 b6 90 b3 5a cc 9f 7c e5 c5 c8 a5 04 79 82 41 0e 89 09 23 f2 c6 cd a2 d8 8c 42 4b be ef
ks2	key: (hexidecimal) a2 0e 0e f5 5b 38 52 46 0e 57 e7 24 20 af 9b bc 00 c3 fc 94 f8 23 2c 21 37 ea 30 55 34 fe eb df
Expected output variables	
buffer	(hexidecimal) 8d c9 fa 9c 5e 9a 45 5d 73 75 6d 20 64 2e 2e 2e

Table 7: Unit Test: Encryption EncryptBufferXTSNonParallel

Decryption The following two unit tests observe the decryption of a single block in the 64-bit and 128-bit versions of the XTS algorithm. Again, the 128-bit version is used as a reference for the 64-bit version.

Scope	
Function	DecryptBufferXTS8Byte
Lines	490 - 543
Input variables	
buffer	(hexidecimal) 8d c9 fa 9c 5e 9a 45 5d 00 00 00 00 00 00 00
length	16
startDataUnitNo	0
startCipherBlockNo	0
ks	key: (hexidecimal) 54 4f d7 b6 90 b3 5a cc 9f 7c e5 c5 c8 a5 04 79 82 41 0e 89 09 23 f2 c6 cd a2 d8 8c 42 4b be ef
ks2	key: (hexidecimal) a2 0e 0e f5 5b 38 52 46 0e 57 e7 24 20 af 9b bc 00 c3 fc 94 f8 23 2c 21 37 ea 30 55 34 fe eb df
Expected output variables	
buffer	Equal to the value of buffer in the 128-bit XTS algorithm.

Table 8: Unit Test: Decryption DecryptBufferXTS8Byte

Scope	
Function	DecryptBufferXTSNonParallel
Lines	726 - 785
Input variables	
buffer	(hexidecimal) 8d c9 fa 9c 5e 9a 45 5d 00 00 00 00 00 00 00 00
length	16
startDataUnitNo	0
startCipherBlockNo	0
ks	key: (hexidecimal) 54 4f d7 b6 90 b3 5a cc 9f 7c e5 c5 c8 a5 04 79 82 41 0e 89 09 23 f2 c6 cd a2 d8 8c 42 4b be ef
ks2	key: (hexidecimal) a2 0e 0e f5 5b 38 52 46 0e 57 e7 24 20 af 9b bc 00 c3 fc 94 f8 23 2c 21 37 ea 30 55 34 fe eb df
Expected output variables	
buffer	(hexidecimal) 4c 6f 72 65 6d 20 69 70 00 00 00 00 00 00 00 00

Table 9: Unit Test: Setup DecryptBufferXTSNonParallel

Common\Volumes.c

No algorithmic changes were made to Common\Volumes.c. The only changes are the replacement of the hash functions (Crypt\GostHash.c and Crypto\Stribog.c) and key derivation functions (Common\Pkcs5.c). If the hash functions and key derivation functions pass the unit tests, Common\Volumes.c is considered to work as intended.

Common\Random.c

No algorithmic changes were made to Common\Random.c. Only the hash functions were replaced. If the hash functions pass the unit tests, Common\Random.c is considered to work as intended.

Common\Pkcs5.c

The PBKDF2 algorithm has a set of official test vectors, as defined in RFC 6070 [3]. Seeing how the only change between the different key derivation functions in Common\Pkcs5.c is in the calls to the underlying hash functions, the unit tests will replace the underlying hash function to SHA-1 to match the official test vectors. If the unit test succeeds, the same result can be expected with different hash functions, given that those hash functions succeeded their respective unit tests.

Scope	
Function	derive_key_sha1
Input variables	
pwd	"password"
pwd_len	8
salt	"salt"
salt_len	4
iterations	1
dk	0 repeated 20 times
dklen	20
Expected output variables	
dk	(hexidecimal) 0c 60 c8 0f 96 1f 0e 71 f3 a9 b5 24 af 60 12 06 2f e0 37 a6

Table 10: Unit Test: PBKDF2 #1

Scope	
Function	derive_key_sha1
Input variables	
pwd	“password”
pwd_len	8
salt	“salt”
salt_len	4
iterations	2
dk	0 repeated 20 times
dklen	20
Expected output variables	
dk	(hexidecimal) 4b 00 79 01 b7 65 48 9a be ad 49 d9 26 f7 21 d0 65 a4 29 c1

Table 11: Unit Test: PBKDF2 #2

Scope	
Function	derive_key_sha1
Input variables	
pwd	“password”
pwd_len	8
salt	“salt”
salt_len	4
iterations	4096
dk	0 repeated 20 times
dklen	20
Expected output variables	
dk	(hexidecimal) 4b 00 79 01 b7 65 48 9a be ad 49 d9 26 f7 21 d0 65 a4 29 c1

Table 12: Unit Test: PBKDF2 #3

Scope	
Function	derive_key_sha1
Input variables	
pwd	“password”
pwd_len	8
salt	“salt”
salt_len	4
iterations	16777216
dk	0 repeated 20 times
dklen	20
Expected output variables	
dk	(hexidecimal) ee fe 3d 61 cd 4d a4 e4 e9 94 5b 3d 6b a2 15 8c 26 34 e9 84

Table 13: Unit Test: PBKDF2 #4

Scope	
Function	derive_key_sha1
Input variables	
pwd	“passwordPASSWORDpassword”
pwd_len	24
salt	“saltSALTsaltSALTsaltSALTsaltSALTsalt”
salt_len	36
iterations	4096
dk	0 repeated 25 times
dklen	25
Expected output variables	
dk	(hexidecimal) 3d 2e ec 4f e4 1c 84 9b 80 c8 d8 36 62 c0 e4 4a 8b 29 1a 96 4c f2 f0 70 38

Table 14: Unit Test: PBKDF2 #5

Scope	
Function	derive_key_sha1
Input variables	
pwd	“pass\0word”
pwd_len	9
salt	“sa\0lt”
salt_len	5
iterations	4096
dk	0 repeated 16 times
dklen	16
Expected output variables	
dk	(hexidecimal) 56 fa 6a a7 55 48 09 9d cc 37 d7 f0 34 25 e0 c3

Table 15: Unit Test: PBKDF2 #6

Crypto\GostCipher.c

Testing of the GOST 28147-89 block cipher is split into two parts. Because the block cipher uses a non-standard S-Box, namely a key-dependent S-Box, unit tests of the implementation can’t be compared to other implementations. Therefore, Crypto\GostCipher.c is first tested with the key-dependent nature disabled against other implementations. Secondly, the S-Box mixing function used to generate key-dependent S-Boxes is tested independently.

Implementation comparison In order to compare the GostCrypt GOST 28147-89 implementation to other implementations, the key-dependent S-Box generation is first disabled. This leaves the block cipher with the “GOST R 34.11-94 CryptoProParamSet” S-Box, specified in RFC 4357 [4]. Using this same S-Box in the OpenSSL implementation and the SCV CryptoManager, we can compare the output on a set of input parameters and see if the resulting ciphertext (when encrypting) or plaintext (when decrypting) matches.

Table 16 shows the input parameters that will be used during this test.

Mode	Input	Key
Encrypting	00 00 00 00 00 00 00 00	00 00
Encrypting	ff ff ff ff ff ff ff ff	ff ff
Encrypting	5b ef 60 12 16 f2 0e 81	64 29 0d 4f 9a 03 0e 21 6b 24 bb cc 93 25 0c 8d 6b 5a 5e a9 52 45 23 fe 30 40 0d ae 5d 85 6a 77
Encrypting	9d 68 ae cb 5a f7 7f 25	a1 46 d6 88 60 1d b0 83 d6 07 74 4e 73 48 9d 67 10 54 fc 7c ce 88 0b ad eb df 62 99 62 5b ad 8c
Decrypting	00 00 00 00 00 00 00 00	00 00
Decrypting	ff ff ff ff ff ff ff ff	ff ff
Decrypting	5b ef 60 12 16 f2 0e 81	64 29 0d 4f 9a 03 0e 21 6b 24 bb cc 93 25 0c 8d 6b 5a 5e a9 52 45 23 fe 30 40 0d ae 5d 85 6a 77
Decrypting	9d 68 ae cb 5a f7 7f 25	a1 46 d6 88 60 1d b0 83 d6 07 74 4e 73 48 9d 67 10 54 fc 7c ce 88 0b ad eb df 62 99 62 5b ad 8c

Table 16: GOST 28147-89 comparison test

S-Box key mixing The S-Box key mixing algorithm mixes the 512-bit Stribog digest of the 256-bit key with the GOST S-Box, which also contains 512 bits of informations (8 boxes * 16 values * 4 bits). In the algorithm, the Stribog digest is xor-ed with the S-Box. In order to test that each of the bits in the S-Box and digest are being used against each other, we use a digest equal to the S-Box. After the algorithm, we would expect the entire S-Box to contain nothing but zero's, as a xor operation on equal values results in zero.

S-Box	Digest
GOST R 34.11-94 CryptoProParamSet [4]	1d 77 47 5a 3e 66 af f4 a4 24 7c 45 91 4b ce 06 57 d9 09 28 b0 9c f4 d1 45 f2 21 b3 fa 0a 80 97 83 a1 e3 1d 6c 18 1b 7c 78 50 65 6e ef be 52 30 d6 8f d6 c9 02 ed ba e2 29 c3 98 ab cb 35 3d 8f

Table 17: GOST 28147-89 xor_s_box

Crypto\GostHash.c

The GOST R 34.11-94 hash function will be tested in two ways: using the example computations from RFC 5831 [5] as test vectors and by comparing it to other implementations.

Example computations Table 18 shows the input parameters and the expected output according to RFC 5831 [5].

Input message	Digest
73 65 74 79 62 20 32 33 3d 68 74 67 6e 65 6c 20 2c 65 67 61 73 73 65 6d 20 73 69 20 73 69 68 54	fa ff 37 a6 15 a8 16 69 1c ff 3e f8 b6 8c a2 47 e0 95 25 f3 9f 81 19 83 2e b8 19 75 d3 66 c4 b1
73 65 74 79 62 20 30 35 20 3D 20 68 74 67 6E 65 6C 20 73 61 68 20 65 67 61 73 73 65 6D 20 6C 61 6E 69 67 69 72 6F 20 65 68 74 20 65 73 6F 70 70 75 53	08 52 F5 62 3B 89 DD 57 AE B4 78 1F E5 4D F1 4E EA FB C1 35 06 13 76 3A 0D 77 0A A6 57 BA 1A 47

Table 18: GOST R 34.11-94 example computations

Implementation comparison The GOST R 34.11-94 implementation of GostCrypt is compared against the OpenSSL and SCV CryptoManager implementations. Table 19 shows the input message used in this test.

Input message
02 b6 cc 2e 0a 46 d8 e1 08 99 9f f3 58 98 cf 70 b6 ca b0 99 bc 5a 2b 16 63 06 e5 8f 72 35 63 3b 0d be ce 4e 8b 71 0c dc 3d d4 b5 32 6c 7b 9c bd 00 8b 96 61 80 c7 75 b8 59 e4 a7 b8 39 67 90 3b 12 a2 92 e4 2f 61 9f e3 eb 31 5d 39 0e 88 44 13 be 06 e2 30 85 f8 01 60 fa f0 27 0e 1b 6c 7c cd 40 45 6c d3 ee d3 0b 60 6b a8 2e 16 f1 9f e5 e9 c2 68 d7 07 f7 b2 8c eb 27 d9 cf cc 96 d8 a7 05 be 91 36 7e 5c 32 25 90 c4 76 69 a0 97 b9 8b a9 a8 95 88 69 fb 95 70 72 19 03 f1 62 b1 11 5a f0 a0 68 ef 5e 19 72 88 05 dc 37 6e 68 1c 5d 0c 67 e8 2f e7 61 5c f3 da ca b8 b1 ec aa 5e 70 9c 70 e6 ab 06 ac 4a 61 f0 3f fb f3 99 ad 74 94 61 08 4b a6 2a 6f 86 b1 5c d3 de f3 97 7d a7 f1 a3 8c 52 db e8 1f a4 74 0b c9 eb 18 85 41 04 ed 3c c4 d7 9f 70 9c a1 84 c2 f9 38 b6 fa 4d 6b 25 01 31 6d 3c e7 23 b1 4e 50 1c be 02 8b aa bd fe 1b b1 95 53 d5 6c f1 12 46 ba 35

Table 19: GOST R 34.11-94 implementation comparisons

Crypto\Stribog.c

The GOST R 34.11-2012 hash function, also known as Stribog, is tested by using the example computations from RFC 6986 as test vectors.

Input message	Digest
32 31 30 39 38 37 36 35 34 33 32 31 30 39 38 37 36 35 34 33 32 31 30 39 38 37 36 35 34 33 32 31 30 39 38 37 36 35 34 33 32 31 30 39 38 37 36 35 34 33 32 31 30 39 38 37 36 35 34 33 32 31 30	48 6f 64 c1 91 78 79 41 7f ef 08 2b 33 81 a4 e2 11 c3 24 f0 74 65 4c 38 82 3a 7b 76 f8 30 ad 00 fa 1f ba e4 2b 12 85 c0 35 2f 22 75 24 bc 9a b1 62 54 28 8d d6 86 3d cc d5 b9 f5 4a 1a d0 54 1b
fb e2 e5 f0 ee e3 c8 20 fb ea fa eb ef 20 ff fb f0 e1 e0 f0 f5 20 e0 ed 20 e8 ec e0 eb e5 f0 f2 f1 20 ff f0 ee ec 20 f1 20 fa f2 fe e5 e2 20 2c e8 f6 f3 ed e2 20 e8 e6 ee e1 e8 f0 f2 d1 20 2c e8 f0 f2 e5 e2 20 e5 d1	28 fb c9 ba da 03 3b 14 60 64 2b dc dd b9 0c 3f b3 e5 6c 49 7c cd 0f 62 b8 a2 ad 49 35 e8 5f 03 76 13 96 6d e4 ee 00 53 1a e6 0f 3b 5a 47 f8 da e0 69 15 d5 f2 f1 94 99 6f ca bf 26 22 e6 88 1e

Table 20: GOST R 34.11-2012 example computations

References

- [1] *TrueCrypt - Security Assessment*. (2014). Open Crypto Audit Project.
- [2] Evans, D., & Larochelle, D. (2002). Improving security using extensible lightweight static analysis. *software, IEEE*, 19(1), 42-51.
- [3] Josefsson, S. (2011). PKCS# 5: Password-Based Key Derivation Function 2 (PBKDF2) Test Vectors.
- [4] Popov, V., Kurepkin, I., & Leontie, S. (2006). RFC 4357: Additional Cryptographic Algorithms for Use with GOST 28147-89, GOST R 34.10-94, GOST R 34.10-2001, and GOST R 34.11-94 Algorithms.
- [5] Dolmatov, V. (2010). GOST R 34.11-94: Hash Function Algorithm.