Noname manuscript No.
(will be inserted by the editor)

Influence of Addition Modulo 2^n on Algebraic Attacks

Oleksandr Kazymyrov · Roman Oliynykov · Håvard Raddum

the date of receipt and acceptance should be inserted later

Abstract Many modern ciphers have a substitution-permutation (SP) network as a main component. This design is well researched in relation to Advanced Encryption Standard (AES). One of the ways to improve the security of cryptographic primitives is the use of additional nonlinear layers. However, this replacement may not have any effect against particular cryptanalytic attacks. In this paper we use algebraic attacks to analyze an SP network with addition modulo 2^n as the key mixing layer. In particular, we show how to reduce the number of intermediate variables in round functions based on SP networks. We also apply the proposed method to the GOST 28147-89 block cipher that allows us to break reduced 8- and 14-round versions with complexity at most 2^{155} and $2^{215.4}$, respectively.

Keywords Block cipher · Addition modulo 2^n · Algebraic attack · Binary decision diagram

Mathematics Subject Classification (2000) 94A60 · 68P25 · 94C10

1 Introduction

Some modern advanced cryptographic primitives use addition modulo 2^n as an extra nonlinear layer in additions to substitutions (S-boxes). Most of these algorithms can be attributed to one of two groups. The first one is based on modular additions, rotations and XORs (ARX schemes). The second group use substitution-permutation (SP), Feistel or other networks. In some cases crypto primitives from the last group replace the XOR operation by addition modulo 2^n in a key mixing layer. On the one hand using addition modulo 2^n may increase the nonlinearity of a round function, but on the other hand it decreases the performance. Therefore finding the balance between these two parameters is one issue for designing crypto primitives. In this paper we will contribute to assessing the security of this design with respect to algebraic attacks.

Oleksandr Kazymyrov
University of Bergen,
E-mail: Oleksandr.Kazymyrov@ii.uib.no

Roman Oliynykov
Kharkiv National University of Radioelectronics,
E-mail: roliynykov@gmail.com

Håvard Raddum
Simula Research Laboratories,
E-mail: haavardr@simula.no

Intuitively, one could assume that adding an extra nonlinear layer increases the complexity of cryptanalytic attacks. One assumption which is sometimes used in cryptanalysis is to replace addition modulo 2^n by the XOR operation which creates a "weaker" cipher that is easier to analyze.

In this paper we investigate how the use of addition modulo 2^n in round functions influences algebraic attacks. In particular, we introduce a much more efficient method describing SP-like round functions in comparison with a naive approach. This allows to decrease the number of variables and as a result reduce the complexity of algebraic attacks. In addition, we describe the GOST 28147-89 block cipher [1] (from here on GOST) using the proposed approach. Solving a system of equations of an 8-round GOST using a binary decision diagram (BDD) method has complexity at most 2^{155}. For 14-round GOST the complexity is $2^{215.4}$ using the same technique.

The rest of the paper is organized as follows. Section 2 describes the general idea of algebraic attacks as well as an approach to solve systems of equations using BDDs. Section 3 explains our description method of SP-like round functions. Section 4 gives the results and details of the BDD algebraic attack against GOST. Finally, the conclusions of the paper are presented in Section 5.

2 Algebraic attacks using a BDD approach

This section describes a general method of an algebraic attack on a block cipher and the BDD approach used in practice.

2.1 The general idea of algebraic attack on block ciphers

Let us consider an algebraic attack on an example of a general crypto primitive based on a substitution-permutation network (SPN). At a high level the cipher has a structure depicted in Figure 1. It consists of three main parts: a key mixing layer, a nonlinear layer and a linear layer. A plaintext P is encrypted over two rounds producing a ciphertext C. At the first stage of the algebraic attack an adversary represents the nonlinear layer as a system of equations over a finite field $\mathbb{F}_{q^n}$. In most cases q equals 2, which means that an equation system is created over a binary field. To represent the entire encryption algorithm the intermediate state Z must be represented as variables. This helps to connect two adjacent nonlinear layers together without increasing the degree of polynomials. However, this works perfectly only when the key mixing layer is linear. For example, in the Advanced Encryption Standard (AES) the key mixing layer is done using the XOR operation [2]. This operation is linear with respect to MixColumns and ShiftRows [3,4]. In this paper we consider a nonlinear key mixing layer, represented by addition modulo 2^n.

When the system of equations representing an entire encryption algorithm, including the key schedule, is created one can apply a solving method to it. The promising method that is used in this paper is an approach based on BDDs [5,7].

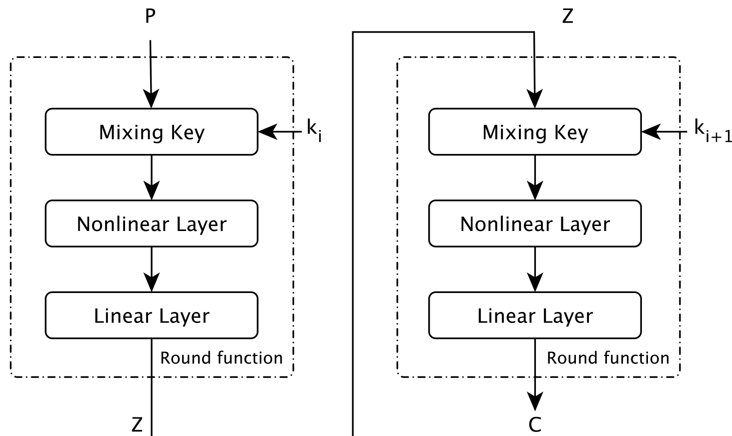


Fig. 1: 2-round SPN block cipher

2.2 Binary decision diagrams

We only give a brief overview of BDDs here, more thorough descriptions can be found in several other sources, for instance in [8]. For clarity, when we talk about BDDs in this paper we mean a zero-suppressed, reduced, and ordered BDD.

A BDD is a class of directed acyclic graphs. There are two particular nodes that must be present in a BDD, called the *source* and the *sink* nodes. Other nodes are called *internal* nodes. We visualize and draw a BDD by placing the source node at the top, the sink node at the bottom, and all internal nodes in between. The internal nodes are structured in horizontal *levels*, where nodes on the same level are drawn next to each other. The levels are numbered from top to bottom, with the source node being the only node on level 1, and the sink node the only node on level L, for some L.

Each node except for the sink node has either one or two out-going edges called the 0-edge and the 1-edge. There are no edges between nodes on the same level, and all edges are directed downwards. Hence an edge going from a node on level i always points to some node on level j, for $j > i$. When drawing a BDD we draw the 1-edges as solid lines and the 0-edges as dotted lines.

One important purpose for studying BDDs is that they can represent a very efficient encoding of complex Boolean functions. In earlier works this is done by associating a variable with each level, except for the bottom level (containing the sink node). We follow the work in [5,7], and associate each level with some **linear combination** of variables. An example of a BDD representing a substitution on four bits can be seen in Figure 2.

2.3 Operations on BDDs

It is well known that it is possible to swap the linear combinations at two adjacent levels without changing the underlying function encoded by the BDD [6]. To do this one may have to introduce new nodes on one of the two levels, and the edges between nodes on these levels must be rearranged. One important thing to note is that only the subgraph where the swap is taking place needs to be acted on, the rest of the BDD will be unchanged.

As we are dealing with linear combinations associated with the levels, we are also interested in adding (XOR-ing) the linear combinations of two adjacent levels together. This can be done using

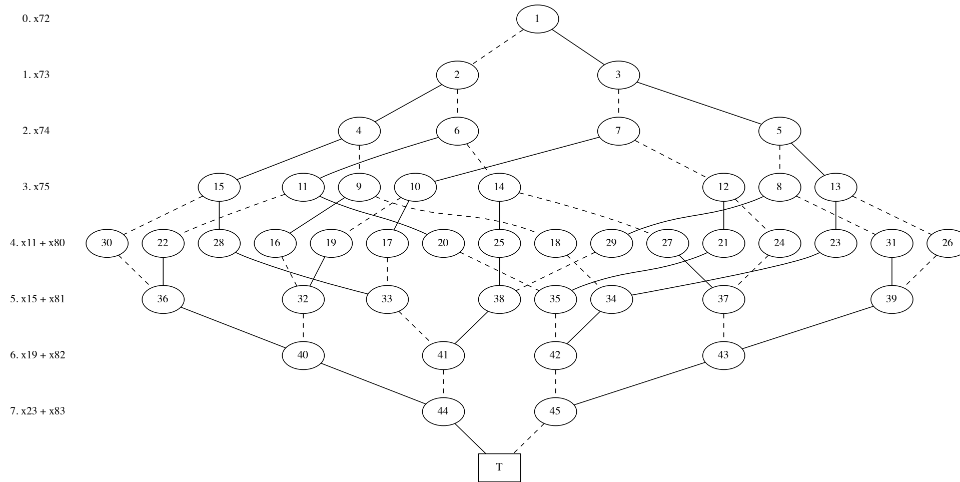


Fig. 2: The BDD representation of a 4-bit S-box

an algorithm similar to that of swapping levels [5]. Again, only the two levels where the addition is taking place is touched while the rest of the BDD remains intact.

We assume our BDDs to be reduced, but after applying a swap or addition of levels, the resulting BDD may be in an unreduced state. The reduction algorithm [8, pp. 14-15] has a running time linear in the number of nodes, and we run the reduction algorithm whenever needed to bring the BDD back to a reduced state. It has been proven that a reduced BDD is unique for a given set of linear combinations of variables and their order in the BDD.

We finally remark that two BDDs may be *joined*, simply by identifying the sink node of one with the source node of the other. Visually this is just stacking them on top of each other in the natural way, resulting in a single BDD.

2.4 Linear absorption

A path from the source to the sink node will assign values to the linear combinations associated with the levels in a BDD. If we choose a 0-edge out from a node on a level associated with the linear combination l, we see this as generating the linear equation $l = 0$. Selecting a path will hence generate a linear system of equations.

In our definition of BDDs, it is fully possible to have linear combinations for the levels that are linearly dependent. Especially after joining some BDDs together the resulting BDD may have dependencies among its linear combinations. When selecting a path in a BDD with dependent linear combinations, we may get a system of linear equations with no solutions, that is, an inconsistent system.

We would like to remove all linear dependencies present in a BDD. This can be done using an algorithm called *linear absorption* [5, 7]. The idea of the algorithm is to identify a set of linearly dependent linear combinations and use swap and addition of levels repeatedly to create a level where these linear combinations have been added together. Due to the dependency, the linear combination for this level will be $\mathbf{0}$. Selecting a 1-edge out from a node on this level would immediately yield an inconsistency. We may therefore remove all 1-edges from nodes on this level, and afterwards

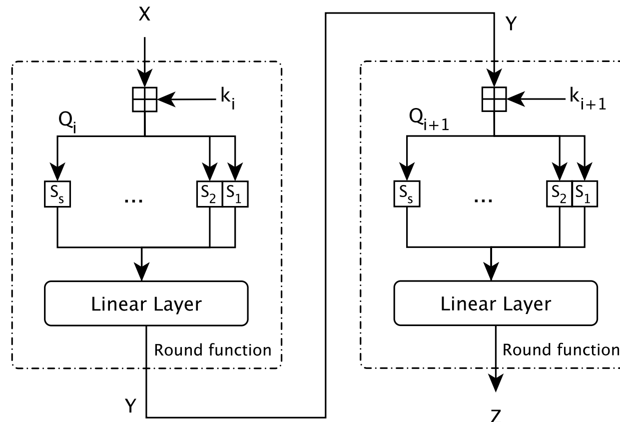


Fig. 3: Two rounds of an SP cipher

remove the whole level. We say the linear dependency we started with has been *absorbed* into the BDD.

We may repeat this process until all linear dependencies have been absorbed resulting in a BDD where all linear combinations associated with the levels are independent. Selecting a path in such a BDD is guaranteed to give a consistent linear system of equations.

2.5 Solving method

We can create a set of relatively small BDDs representing an encryption algorithm. The general method we can use to solve such a system is to repeatedly join some of the BDDs and absorb all linear dependencies we get along the way. If we can do this until all BDDs have been joined into one, finding the solution is simply to select one of the remaining paths and solve the linear system of equations we get. Because of the linear absorption we have done, we know that this system will be consistent and will give a solution for all the variables, including the variables for the secret key.

When applying swap and addition of levels, the number of nodes in the BDD may grow. We may therefore run into BDDs that are too big for a computer to handle, and hence we can (of course) not solve any system in practice. Finding how to keep the sizes of BDDs low when joining and absorbing is currently an active research topic.

3 Addition modulo 2^n in SP network

In Section 2 we gave a general overview of a block cipher based on the SP network. Now let us assume that we have a more specific nonlinear layer represented by substitutions used in parallel (Figure 3). At the same time a linear layer is linear with respect to addition modulo 2 (XOR). These two layers are widely used in modern crypto primitives. The last part of the round function is addition modulo 2^n that differs from the usually applied XOR operation. This difference causes a problem for the algebraic attack because of its nonlinearity.

In naive applications of the algebraic attack one needs to introduce extra variables between two nonlinear layers (i.e., substitutions and the addition). The solving complexity depends on both the number of equations and the number of variables [9], and a natural question is how to increase the ratio of equations to variables. One of the ways is to use several plaintext/ciphertext pairs (PCPs)

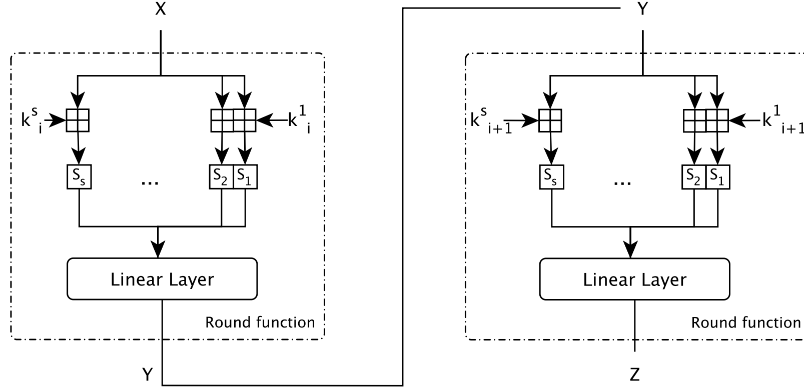
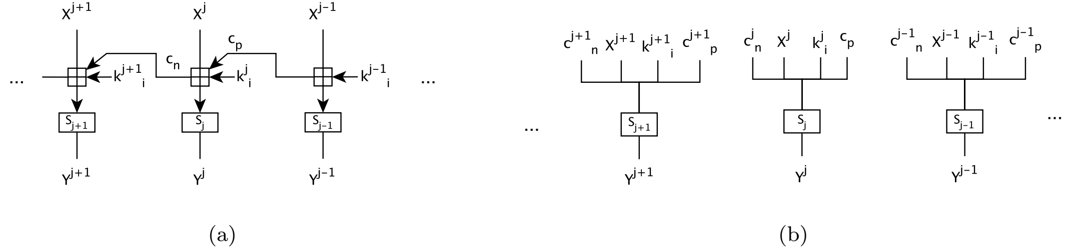


Fig. 4: Alternative representation of the round function

with the reuse of variables across different instances of PCPs. This approach does not help directly in our case (especially with increasing number of rounds), but can be used at further steps. Another way is to create a bigger S-box. The best we can do is to represent the entire round function as a system of equations over $\mathbb{F}_2$. In theory this works fine [10, 11], but in practice we need an enormous amount of computation and storage resources.

Instead of creating a big system of equations we propose to split addition modulo 2^n into the size of the substitutions. In other words, use addition modulo $2^{|S_i|}$ followed by the substitution S_i (Figure 4). However, this leads to an issue related to carry bits (Figure 5a). To solve this issue let us define a new substitution $F_2^{2|S_i|+1} \to F_2^{|S_i|}$, where the extra bits are the previous carry bit (c_p) and the round key (Figure 5b). Each next carry bit (c_n) is generated by other bits of the same block. Since we describe the same nonlinear layer, then $c_p^{j+1} = c_n^j$.



(a) (b)

Fig. 5: Representation of addition modulo 2^n and substitutions using larger S-boxes

This approach to represent the round function allows us to use the general method for describing the encryption algorithm stated in Section 2. It also allows to reduce the number of variables. However, increasing the input space of a substitution may also increase the degree of polynomials [11]. Unlike most other methods the complexity of BDD approach does not depend on the degree of the polynomials.

Decreasing the number of variables in the considered round function leads to increasing the complexity of the algebraic description. Since the degree of polynomials has an exponential effect on the solving complexity in many known methods the BDD approach has advantages in our case.

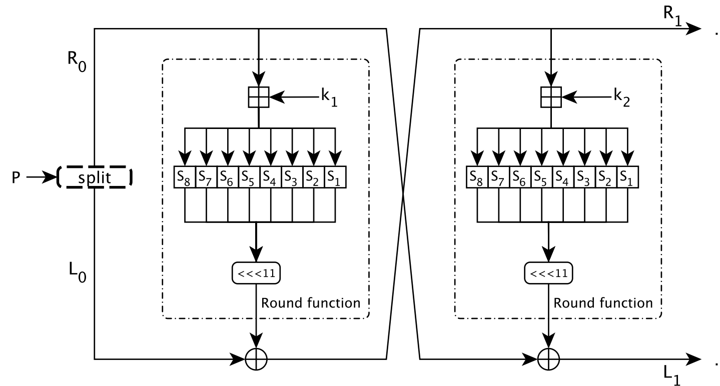


Fig. 6: Two rounds of the GOST cipher

4 Application of the proposed method on GOST

The GOST block cipher was developed in the 1970's as a standard for securing Soviet communications. The English description of the algorithm was released to the public in 1994 [12]. In 2014 a draft version of the new standard has been published, which is planned to be used in the near future [13]. The draft includes two block ciphers that are oriented on software and hardware implementations. For the hardware version the algorithm of GOST 28147-89 with fixed substitutions will be used.

4.1 Structure of the GOST algorithm

GOST is a Feistel cipher with 32 rounds without swapping after the last round. The block size is 64 bits and the key size is 256 bits. The round function takes a 32-bit input and a 32-bit round key to produce an output of 32 bits. The components of the round function are addition modulo 2^{32}, denoted by $\boxplus$, a group of eight 4-bit S-boxes, and a cyclic rotation.

At the start of the round the round key is added to the input modulo 2^{32}. The result of the addition is then split into 8 nibbles, which are substituted according to the set of S-boxes. The output of the S-box layer is assembled back into a 32-bit word and rotated by 11 bits to the left. The rotated word is the output of the round function. A diagram describing GOST can be seen in Figure 6.

4.2 Specification of the substitutions

The S-boxes to be used in GOST 28147-89 are not specified in the original standard. The S-boxes are supposed to be set up by the user, and may be kept secret to be regarded as additional key material. The cipher should be secure also with known S-boxes. Several sets of substitutions that were recommended to be used in the algorithm have been published [14, 13]. We took the set of substitutions defined in the draft standard. This set is also known as "id-tc26-gost-28147-param-A" and have the object identifier 1.2.643.7.1.2.5.1.1. All 8 substitutions are listed in Table 1.

Table 1: The set of S-boxes used in the draft standard

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
S_8	1	7	E	D	0	5	8	3	4	F	A	6	9	C	B	2
S_7	8	E	2	5	6	9	1	C	F	4	B	0	D	A	3	7
S_6	5	D	F	6	9	2	C	A	B	7	8	1	4	3	E	0
S_5	7	F	5	A	8	1	6	D	0	9	3	E	B	4	2	C
S_4	C	8	2	1	D	4	F	6	7	0	A	5	3	E	9	B
S_3	B	3	5	8	2	F	A	D	E	1	7	4	C	9	6	0
S_2	6	8	2	3	9	A	5	C	1	E	4	7	B	D	0	F
S_1	C	4	6	2	A	5	B	9	E	8	D	7	0	3	F	1

4.3 Key expansion

The key expansion routine of GOST is extremely simple and one can argue that GOST has no key expansion. The user-selcted key has 256 bits, and is seen as eight 32-bit numbers $k_0, \ldots, k_7$. For the first 24 rounds of GOST the k_i's are used as round keys in sequence, starting with k_0 and ending with k_7. For the last 8 rounds the round keys are still $k_0, \ldots, k_7$, but they are used in reversed order, starting with k_7 in round 25 and ending with k_0 in round 32.

4.4 Algebraic-differential cryptanalysis of GOST

A general concept of an algebraic-differential attack for the 4-round GOST is presented in Figure 7. This attack is based on the fact that some variables can be reused for several plaintext/ciphertext pairs. Reusing variables increases the ratio of equations to variables. An important goal for an attacker is to maximize the number of equations keeping the number of variables at low level.

The main stages of the attack are as follows. At the first step one describes the entire encryption algorithm for 1 PCP using the method stated in Section 3. Next, a difference is added to the left half of the plaintext in the 8th nibble. The application of this difference allows us to minimize the propagation of the input difference for several rounds. In Figure 7 the white and green blocks depict known (defined for the first plaintext) variables. At the same time the blue color describes unknown (varying) values of the bits. In the 3rd round the gradation of the blue color from dark to light corresponds to decreasing uncertainty. This uncertainty can be evaluated as the probability that the propagation of an input difference stops after some nibbles. Theorem 1 describing this with a proof is given in Appendix A.

The best found propagation is presented in Figure 8. According to Theorem 1 with $s = 4$ and $e = 5$, the probability that the propagation of the input value stops after the 3rd nibble is 0.999985. This fact allows to reuse variables up to 5 rounds. We use the scheme together with the BDD method to attack GOST up to 14 rounds.

4.5 Computational results

GOST has 256 unknown key bits, but only a 64-bit block. This means that one must use at least 4 known PCPs to uniquely determine the key, which leads to rather big equation systems. Since the final BDD contains *all* solutions of a system, it becomes difficult to solve GOST systems directly.

To test the solving of GOST systems in practice it is necessary to fix (guess) parts of the key in the system before solving. When fixing k bits of the key, the total complexity of the attack becomes $\mathcal{O}(c2^k)$, where c is the complexity of solving the system. For $c < 2^{256-k}$ we obtain a valid algebraic attack.

Increasing the number of fixed key bits to the correct values, we decrease the number of PCPs needed to define the system uniquely. We have done some experiments on GOST systems using the BDD solver, and report on two successful attacks below.

4.5.1 Fixing the first 192 bits of the key.

In this system only the 64 bits in k_6 and k_7 used in round 7 and 8 are considered to be unknown. The values for $k_0, \ldots, k_5$ are guessed (known), and these subkeys are used in rounds $1, \ldots, 6$ and rounds $9, \ldots, 14$. The attack therefore fits a 14-round version of GOST, where the guessed key bits reduces the cipher to only two actual rounds.

Not surprisingly, this system is very easy to solve, and the memory consumption of the solver is negligible. The solver used 0.7 seconds on a PC which is equivalent to $2^{23.4}$ encryptions of 14-round GOST. In total we get an algebraic attack with complexity $2^{215.4}$.

4.5.2 Fixing the last 96 bits of the key.

We consider an 8-round version of GOST, where we have guessed the 96 bits in k_5, k_6, k_7. When constructing the system, we essentially cover the first five rounds of GOST, involving 160 unknown key bits. This system is not trivial to solve. Also, because of the reuse of variables and equations as explained in Section 4.4, we get an underdefined system that does not give a unique solution.

When using 8 PCP pairs, the solver returns a BDD with 2^{59} paths. However, due to the memory constrains of the PC there are still 21 dependencies remaining among the linear combinations for

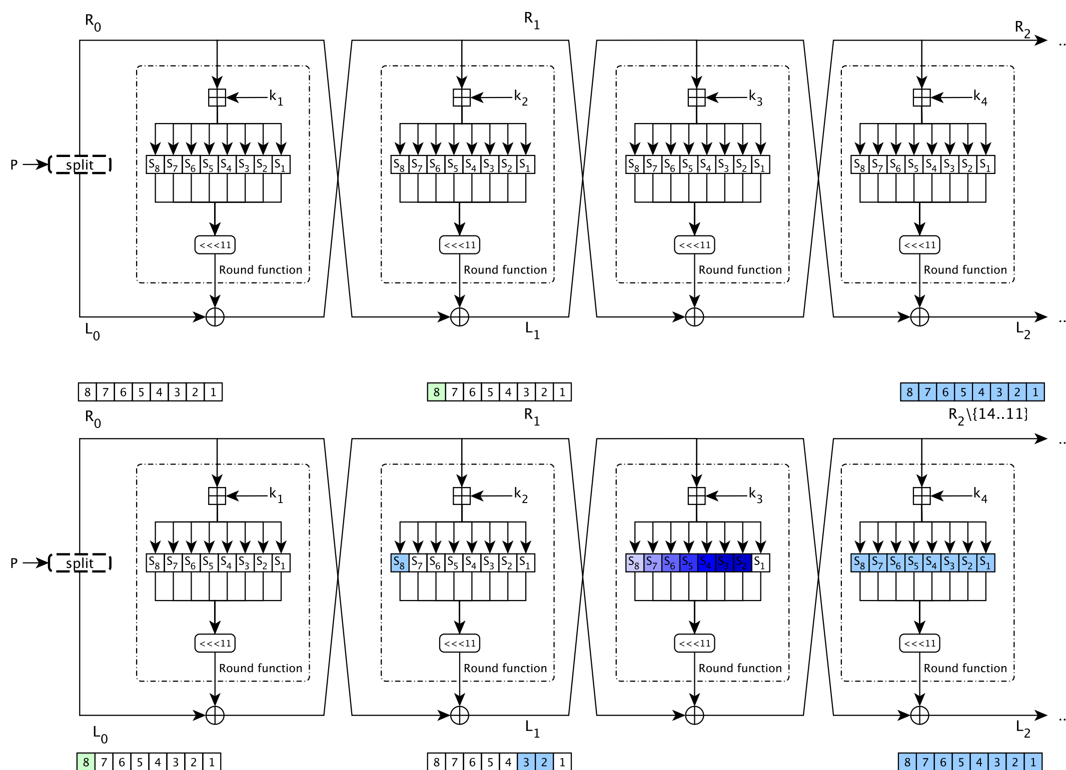


Fig. 7: An algebraic-differential attack of 4-round GOST

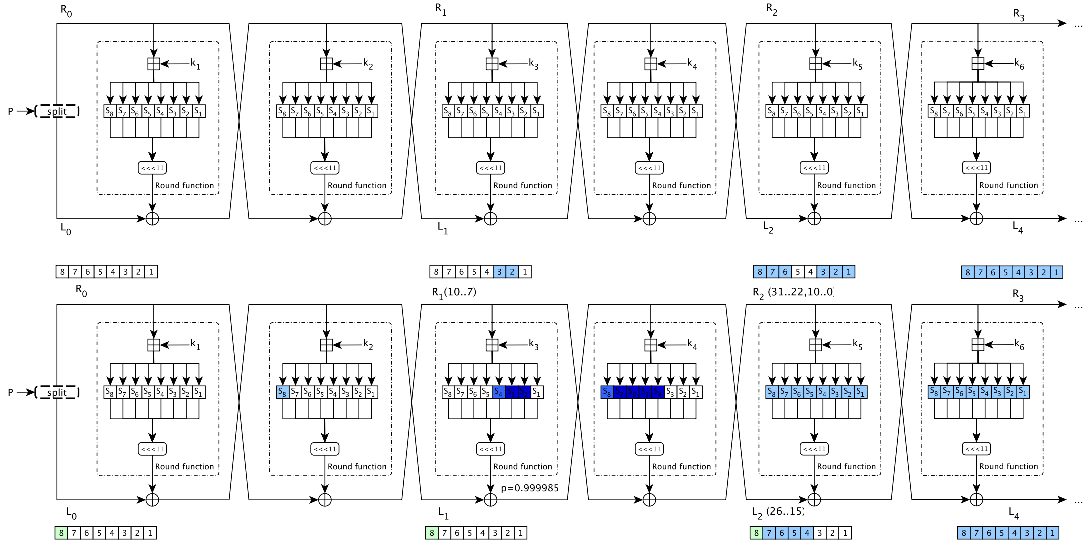


Fig. 8: Algebraic-differential analysis of 6-round GOST

the levels. One path will give a consistent linear system (see Sec. 2.4) with probability 2^{-21}, so we expect 2^{38} valid paths that give solutions to the system.

The time complexity for the solver to find this BDD corresponds to $2^{30.1}$ 8-round encryptions of GOST. Searching in the BDD to find any one solution to the system has the complexity of searching through approximately 2^{21} paths. Finding one particular solution requires searching through the whole BDD, giving a complexity of searching through 2^{59} paths. Checking one path in the BDD has lower complexity than one GOST encryption, but for simplicity we assume that checking one path takes the same time as doing one GOST encryption.

The complexity for building the final BDD is $2^{30.1}$ GOST encryptions. The complexity of finding a unique key, that is finding the unique path giving this key, has complexity 2^{59}. These two tasks must be done for every guess of the 96 bits of key, so the complexity for the attack becomes at most $2^{96+30.1} + 2^{96+59} \approx 2^{155}$ GOST encryptions, which is a lot better than brute-forcing 256 unknown key bits.

5 Conclusions

The main idea that we are trying to convey in this work is that any changes in the structure of crypto primitives must be re-verified. Intuitive assumptions may be wrong. On the one hand changes increase the resistance against some attacks, but on the other hand they may give little or no added protection against other attacks.

We decided to analyze an SP-network with addition modulo 2^n against algebraic attacks. The application of the classical algebraic description over $\mathbb{F}_2$ of modern ciphers gives lots of intermediate variables that increase the total solving complexity. However, the number of variables can be reduced by the use of larger substitutions where the non-linear layers are combined. A negative consequence of this method may be that the degrees of the polynomials increase. Higher degrees create a problem for lots of solving methods. Nonetheless, the BDD method does not depend on the degree of polynomials and seems to be perfectly suited for solving systems of equations describing the considered SP networks. Applying the proposed method to GOST gives the opportunity to find a 256-bit secret key for the 8-round and 14-round versions with the complexities of 2^{155} and $2^{215.4}$ encryptions, respectively.

On the whole, the proposed method of describing addition modulo 2^n followed by substitutions is universal and helps to estimate the security level of crypto primitives against algebraic attacks more precisely.

References

1. Dolmatov, V.: GOST 28147-89: Encryption, Decryption, and Message Authentication Code (MAC) Algorithms. RFC 5830 (Informational), March 2010.
2. FIPS PUB 197: Advanced Encryption Standard (AES). *National Institute of Standards and Technology,* 2001.
3. Knudsen, L. R., Robshaw, M.: *The block cipher companion.* Information Security and Cryptography. Springer Berlin Heidelberg, 2011.
4. Kazymyrov, O., Kazymyrova, V.: Extended criterion for absence of fixed points. *In Pre-proceedings of 2nd Workshop on Current Trends in Cryptology (CTCrypt 2013),* pp. 177–191, 2013.
5. Schilling, T., Raddum, H.: Solving compressed right hand side equation systems with linear absorption. In Helleseth, T., Jedwab, J. (eds.), *Sequences and Their Applications – SETA 2012*, vol. 7280 of *Lecture Notes in Computer Science*, pp. 291–302. Springer Berlin Heidelberg, 2012.
6. Rudell, R.: Dynamic variable ordering for ordered binary decision diagrams. *Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design* **12**, pp. 42–47, 1993.
7. Kazymyrov, O., Raddum, H.: Algebraic attacks using binary decision diagrams. *In Pre-proceedings of BalkanCryptSec 2014,* pp. 31–44, 2014.
8. Knuth, D. E.: *The Art of Computer Programming. Bitwise Tricks & Techniques. Binary Decision Diagrams*, vol. 4. Addison-Wesley, 2009.
9. Albrecht, M.: *Algorithmic algebraic techniques and their application to block cipher cryptanalysis.* Ph.D. thesis, Royal Holloway, University of London, the United Kingdom, 2010.
10. Courtois, N., Pieprzyk, J.: Cryptanalysis of block ciphers with overdefined systems of equations. In Zheng, Y. (ed.), *Advances in Cryptology — ASIACRYPT 2002*, vol. 2501 of *Lecture Notes in Computer Science*, pp. 267–287. Springer Berlin Heidelberg, 2002.
11. Kazymyrov, O., Oliynykov, R.: Choosing substitutions for symmetric cryptographic algorithms based on the analysis of their algebraic properties. In *Mathematical modeling. Information Technology. Automated control systems.*, vol. 925, pp. 79–86. V. N. Karazin Kharkov National University, Ukraine, 2010. (In Russian).
12. Pieprzyk, J., Tombak, L.: Soviet encryption algorithm. *Electronic source,* 1994. `https://www.thc.org/root/phun/stego-challenge/gost-spec.pdf`.
13. GOST R __.___-20__ (draft): Information technology. Cryptographic data security. Block ciphers. *Electronic source,* 2014. `http://www.tc26.ru/standard/draft/GOSTR-bsh.pdf`. (In Russian).
14. Popov, V., Kurepkin, I., Leontiev, S.: 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. RFC 4357 (Informational), January 2006.

A Propagation of carry bits in a modular adder

Theorem 1 *Let Σ_n be an n-bit length adder implementing addition in the group $\mathbb{Z}_{2^n}$. The first input to Σ_n is a random uniformly chosen group element, and the second input is a group element with the bit representation $I_0 = i_n i_{n-1} \dots i_{s+b+1} 00 \dots 0 i_b i_{b-1} \dots i_1$, where $i_j \in \{0,1\}$. Let $S_0 = C_0 || E_0 || V_0 || B_0$ be an output value of the adder with $S_0 \in \mathbb{Z}_{2^n}$, $C_0 = r_c^{(C_0)} r_{c-1}^{(C_0)} \dots r_1^{(C_0)}$, $E_0 = r_e^{(E_0)} r_{e-1}^{(E_0)} \dots r_1^{(E_0)}$, $V_0 = r_s^{(V_0)} r_{s-1}^{(V_0)} \dots r_1^{(V_0)}$ and $B_0 = r_b^{(B_0)} r_{b-1}^{(B_0)} \dots r_1^{(B_0)}$, where $c > 1$, $e > 1, s > 1, b > 1$, and $c + e + s + b = n$. If values $I_J = i_n i_{n-1} \dots i_{s+b+1} j_s j_{s-1} \dots j_1 i_b i_{b-1} \dots i_1$ with $j_s j_{s-1} \dots j_1 = J \in \{1, \dots, 2^s - 1\}$ are sequentially taken as the second adder input, then for the output values $S_J = C_J || E_J || V_J || B_J$, where $B_J = B_0$ (i.e. constant), the probability of the event that $C_J = C_0$ for all inputs I_J $(J \in \{1, \dots, 2^s - 1\})$ is equal to $1 - \frac{2^s - 1}{2^{se}}$.*

Proof Assume that an output value of the adder Σ_n is divided into the following 4 groups of bits

- B is a constant number;
- V is a variable part of the sum;
- E is a very likely variable part of the sum because of carry bits generated by V when zero input bits are changed to ones;
- C is an unlikely changed part of the sum, since carry bits generated by V is very likely to be stopped at E.

Thus, the output values definitely changes at block V and may affect higher blocks E and C through carry bits from V. It is necessary to estimate the probability of the event that there is no carry bit from V to E, or the propagation of the carry bits will not affect C via E, so C remains constant ($C_J = C_0$).

From the theorem description follows that at least one different value $C_J \neq C_0$ for any J breaks the theorem conditions. It means that it is sufficient to take into consideration only the value $J = 1_s 1_{s-1} \dots 1_2 1_1$ as long as J (the block of the second input corresponding V) is taking all values from $0_s 0_{s-1} \dots 0_2 0_1$ till $1_s 1_{s-1} \dots 1_2 1_1$. If a carry bit appears for any other value of J, it definitely appears for $J = 1_s 1_{s-1} \dots 1_2 1_1$. The probability of the event that a given random block of s bit length with the second input $1_s 1_{s-1} \dots 1_2 1_1$ of the adder Σ_n will trigger E is equal to $\frac{2^s-1}{2^s}$.

Now it is necessary to estimate the probability of the event that for I_0 there is no carry bit from E to C, but it appears for $I_J \neq I_0$. For this case the corresponding bits of the first and the second input of the adder must form the output value $1_e 1_{e-1} \dots 1_2 1_1$. The probability of this event is 2^{-e} (taking into account all pairs of input values given the necessary sum). Since the first argument of the adder is chosen randomly and independently, the propagation probability of carry bits from E to C with the second input I_J, and with absence of carry bits for I_0, is calculated as the product $2^{-e} \cdot \frac{2^s-1}{2^s}$. Accordingly, the probability of the complement event, that is the block C remains constant, is $1 - \frac{2^s-1}{2^{se}}$.

Finally, to obtain the situation when the block B has influence on the probability the following condition must be satisfied: the carry bits of V_0 and V_J are different. The case when the carry bit is already presented for V_0 (due to the carry bit from B) leads to $C_J = C_0$ (the carry bit is already presented and cannot change the higher bits). The number of variants when the input value results in the carry bit for C_0 is equal to number of variants when the carry bit appears only for C_J. Thus, the carry bits from B have no influence on the probability of the event $C_J \neq C_0$.

Therefore, the probability of the event that for $B_J = B_0$ the bits of C_J and C_0 are the same for all I_J equals $1 - \frac{2^s-1}{2^{se}}$. □