

Improved guess-and-determine and distinguishing attacks on SNOW-V

Jing Yang¹, Thomas Johansson¹ and Alexander Maximov²

¹ Dept. of Electrical and Information Technology, Lund University, Lund, Sweden

{jing.yang,thomas.johansson}@eit.lth.se

² Ericsson Research, Lund, Sweden

alexander.maximov@ericsson.com

Abstract. In this paper, we investigate the security of SNOW-V, demonstrating two guess-and-determine (GnD) attacks against the full version with complexities 2^{384} and 2^{378} , respectively, and one distinguishing attack against a reduced variant with complexity 2^{303} . Our GnD attacks use enumeration with recursion to explore valid guessing paths, and try to truncate as many invalid guessing paths as possible at early stages of the recursion by carefully designing the order of guessing. In our first GnD attack, we guess three 128-bit state variables, determine the remaining four using four consecutive keystream words, and finally verify the correct guess according to the next three consecutive keystream words. The second GnD attack is similar but exploits one more keystream word as side information helping to truncate more guessing paths. Our distinguishing attack targets a reduced variant where 32-bit adders are replaced with exclusive-OR. The main advantage of our distinguishing attack is that the contribution from the linear part can be cancelled locally, while classical distinguishing attacks require to combine keystream words very far away to achieve so. Thus the samples in our distinguishing attack can be collected from short keystream sequences under different (Key, IV) pairs. These attacks do not threaten SNOW-V, but provide more in-depth details for understanding its security and give new ideas for cryptanalysis of other ciphers.

Keywords: SNOW-V · Guess-and-determine attack · Distinguishing attack

1 Introduction

SNOW-V is a new member of the SNOW family of stream ciphers, proposed in 2019 in response to the new requirements of the confidentiality and integrity algorithms in 5G and beyond from 3GPP [EJMY19]. First, the 256-bit security level is expected in 5G to resist against attackers equipped with quantum computing capability [3GP19], while the predecessor SNOW 3G being used in 4G was only specified for 128-bit key length. If the key length in SNOW 3G would be increased to 256 bits, there exist academic attacks against it much faster than exhaustive key search, see e.g. [YJM19]. Besides, the algorithms are expected to achieve high throughput in software environments, as many of the network nodes in 5G can be virtualised and the ability to use specialised hardware for cryptographic primitives will thus be reduced. The targeted speed for downlink transmission in 5G is 20 Gbps, while current performance benchmarks for SNOW 3G only give approximately 9 Gbps in a pure software environment [YJ20]. SNOW-V is designed given these motivating facts and aims to provide a 256-bit security level and perform fast enough in software environments. It has been submitted to SAGE (Security Algorithms Group of Experts) within 3GPP for consideration for possible use in 5G and is under evaluation [SAG20].

SNOW-V follows the design principles of the SNOW family, with a linear part consisting of LFSRs (Linear Feedback Shift Registers) to serve as the source of pseudo-randomness, and a non-linear part called FSM (Finite State Machine) to disrupt the linearity. Both parts are redesigned and better aligned in order to adapt to the higher performance and stronger security demands in 5G. The FSM part is now increased to a larger size and accommodates two AES encryption rounds to serve as two large S-boxes for non-linearity, thus taking full advantage of the intrinsic instruction of AES encryption round supported by most mainstream CPUs. SNOW-V can achieve rates up to 58 Gbps for encryption in a pure software environment.

Since proposed, SNOW-V has received internal and external evaluations [EJMY19, CDM20], which exhaustively visit all the promising cryptanalysis techniques of stream ciphers and ensure that none of them applies to SNOW-V faster than exhaustive key search. After that, a number of publications about cryptanalysis of SNOW-V had appeared as well. The authors in [JLH20] propose a byte-based guess-and-determine (GnD) attack against SNOW-V with complexity 2^{406} using seven keystream words. In their attack, the state registers are split into bytes with some carriers introduced, and dynamic programming is employed to help search a good guessing path that requires guessing as few bytes as possible. In [GZ21], the authors perform linear cryptanalysis of SNOW-V and propose correlation attacks against three reduced variants of it, in which either a permutation operation is omitted or 32-bit arithmetic additions are replaced with 8-bit additions. The closest variant is SNOW-V $_{\boxplus_{32}, \boxplus_8}$, in which one \boxplus_{32} (four parallel 32-bit adders) is replaced by 8-bit adders, and the complexity of the correlation attack against it is 2^{377} . The attack complexity against the full SNOW-V is unknown and one can expect that it would be even higher. The paper [HII⁺21] investigates the security of the initialisation of SNOW-V, using MILP (Mixed-integer linear programming) model to efficiently search for integral and differential characteristics. The resulting distinguishing or key recovery attacks are applicable to SNOW-V with reduced initialisation rounds of five, out of the original 16 rounds, which indicates that the SNOW-V initialisation has a very good security margin.

Though none of these cryptanalysis efforts result in a valid attack against SNOW-V faster than exhaustive key search, they are still of great importance for fully understanding the security of the cipher.

Contribution. In this paper, we perform improved cryptanalysis of SNOW-V, and propose two guess-and-determine attacks against the full version with complexities 2^{384} and 2^{378} , respectively, and one distinguishing attack against a reduced variant with complexity 2^{303} . To the best of our knowledge, our GnD attacks are the best known state recovery attacks against SNOW-V, while our linear cryptanalysis provides a new sight for distinguishing attacks against word-based stream ciphers.

In our guess-and-determine attacks, we take full advantage of the observation that some guessing values will not give valid solutions at some point in the middle of the guessing process, and one can immediately terminate this guessing branch and trace back to guess another value. Thus, some efforts of going deeper can be saved. The earlier and more often one can find such cases, the more efforts can be saved. In our first GnD attack, we guess three 128-bit state variables, and efficiently enumerate the solutions for the remaining four using four keystream words. The number of valid solutions is around 2^{384} and we use the next three keystream words to uniquely decide the correct one. We carefully design the guessing order of the variables, such that most guessing paths would be truncated at some point in the middle without going into the end. In our second GnD attack, we use one additional keystream word to impose more constraints and truncate more guessing paths, thus further reduce the complexity to 2^{378} . The improvement is not very big, but the idea of using side information to truncate more guessing paths and thus reduce the complexity is interesting in general.

Table 1: Attacks against SNOW-V and its variants.

Attack	Complexity	Data	Reference
Guess-and-Determine	2^{406}	7 keystream words	[JLH20]
	2^{384}	7 keystream words	Section 3
	2^{378}	8 keystream words	Section 4
Linear Cryptanalysis	2^{377^*}	long keystream of length 2^{254}	[GZ21]
	$2^{303^{**}}$	many short keystreams	Section 5
Integral Distinguisher	2^{48} (5 rounds)	2^{48}	[HII ⁺ 21]
Integral Key Recovery	2^{255} (3 rounds)	2^{15}	
Differential Distinguisher	2^{37} (5 rounds)	2^{37}	[HII ⁺ 21]
Differential Key Recovery	2^{154} (4 rounds)	2^{27}	

* The attack is applied on the reduced variant SNOW-V $_{\boxplus_{32}, \boxplus_8}$.

** The attack is applied on the reduced variant SNOW-V $_{\oplus}$.

In our distinguishing attack, we target a reduced variant of SNOW-V, denoted SNOW-V $_{\oplus}$, in which the 32-bit adders are replaced with exclusive-OR. We consider three consecutive 128-bit keystream words and linearly combine the bytes in these keystream words, such that the contribution from the LFSR is directly cancelled. We then explore linear masking coefficients in an efficient way to cancel out as many S-box approximations in the FSM as possible, thus to make the bias larger. We find a bias evaluated using Squared Euclidean Imbalance (SEI) around 2^{-303} and give a distinguishing attack with complexity 2^{303} . Our distinguishing attack has the advantage that the samples for building the biased keystream sequence can be collected under different K (Key) and IV (Initialisation Vector) pairs, while typical distinguishing attacks against LFSR-based stream ciphers can only collect samples from one very long keystream sequence corresponding to one specific (K, IV) pair.

Table 1 lists the existing cryptanalysis results against SNOW-V, while they all show that SNOW-V is resistant against these types of attacks.

Outline. We first give notations and expressions in Section 2, together with a brief description of SNOW-V. We then demonstrate two guess-and-determine attacks in Section 3 and Section 4, respectively. In Section 5, we perform linear cryptanalysis against SNOW-V and propose a distinguishing attack against a reduced variant. We end the paper with conclusions in Section 6.

2 Preliminaries

2.1 Notations

The exclusive-OR and addition modulo 2^m are denoted by \oplus and \boxplus_m , respectively. $\|$ denotes the concatenation operation. The m -dimensional extension field is denoted as \mathbb{F}_{2^m} . For two variables $x, y \in \mathbb{F}_{2^m}$, xy denotes the multiplication over \mathbb{F}_{2^m} . Given two vectors of length t , $\mathbf{a} = (a_{t-1}, \dots, a_1, a_0)$ and $\mathbf{b} = (b_{t-1}, \dots, b_1, b_0)$, where $a_i, b_i \in \mathbb{F}_{2^m}$ for $0 \leq i \leq t-1$, we use \mathbf{ab} to denote the point-wise multiplication defined as $\mathbf{ab} = \bigoplus_{i=0}^t a_i b_i$, where $a_i b_i$ is the multiplication over \mathbb{F}_{2^m} . We sometimes also use $(a_{t-1}, \dots, a_1, a_0) \cdot (b_{t-1}, \dots, b_1, b_0)$ to denote the same point-wise multiplication. If $m = 1$, \mathbf{ab} is the standard inner product.

The variables throughout the paper are normally 128-bit long, unless otherwise specified. For a 128-bit variable x , we can express it as a byte vector $(x_{15}, x_{14}, \dots, x_1, x_0)$, where x_i ($0 \leq i \leq 15$) is the i -th byte. We use several subscripts to indicate several bytes of a variable. For example, $x_{1,5,7}$ denotes the 1-st, 5-th, 7-th bytes of x . To express the vector of these bytes, we add a notation $[\cdot]$ outside. For example, $[x_{1,5,7}]$ denotes the byte vector

(x_1, x_5, x_7) . We add $||$ between subscript indices to denote the concatenation of these bytes, e.g., $x_1||x_5||x_7$.

2.2 Introduction to SNOW-V

In this section, we give a brief introduction to SNOW-V and predefine some notations and expressions which will be frequently used in the subsequent cryptanalysis. The overall schematic of SNOW-V is depicted in Figure 1. It follows the design principles of the SNOW-family, consisting of a linear part made of LFSRs and a non-linear part called FSM. Both parts are redesigned in order to adapt to the requirements in 5G in terms of encryption speed and security level.

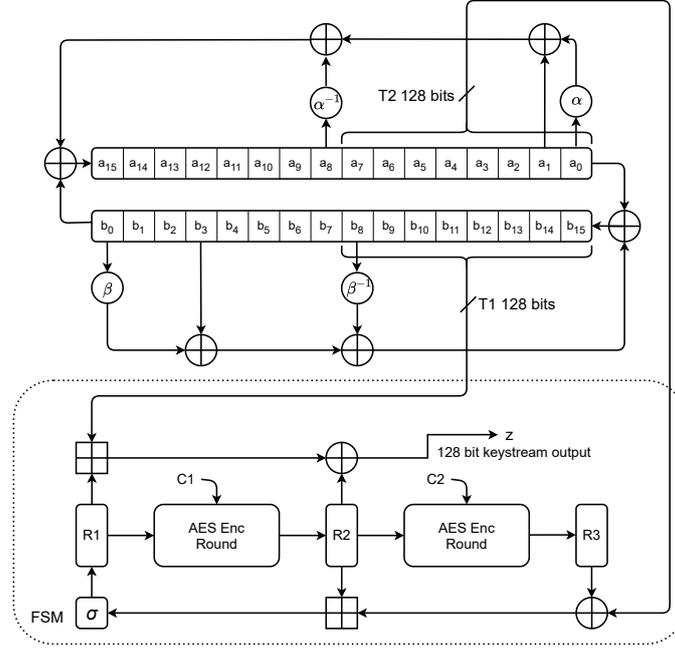


Figure 1: Overall schematic of SNOW-V [EJMY19].

The LFSR part is a new circular construction consisting of two 256-bit registers, named LFSR-A and LFSR-B, feeding to each other. Both LFSRs have 16 cells, each of which holds an element from the finite field $\mathbb{F}_{2^{16}}$. The 32 cells are denoted a_{15}, \dots, a_0 and b_{15}, \dots, b_0 , respectively. The elements in LFSR-A and LFSR-B are generated according to the generating polynomials $g^A(x)$ and $g^B(x)$, respectively, which are expressed as below:

$$g^A(x) = x^{16} + x^{15} + x^{12} + x^{11} + x^8 + x^3 + x^2 + x + 1 \in \mathbb{F}_2[x],$$

$$g^B(x) = x^{16} + x^{15} + x^{14} + x^{11} + x^8 + x^6 + x^5 + x + 1 \in \mathbb{F}_2[x].$$

Denote the state of the LFSR-A and LFSR-B at clock t as $(a_{15}^{(t)}, \dots, a_0^{(t)})$ and $(b_{15}^{(t)}, \dots, b_0^{(t)})$, respectively. Every time when clocking, the value in a cell is shifted to the next cell with a smaller index and $a_0^{(t)}, b_0^{(t)}$ exit the LFSRs. The values in cell a_{15}, b_{15} are updated as:

$$a^{(t+16)} = b^{(t)} + \alpha a^{(t)} + a^{(t+1)} + \alpha^{-1} a^{(t+8)} \pmod{g^A(\alpha)},$$

$$b^{(t+16)} = a^{(t)} + \beta b^{(t)} + b^{(t+3)} + \beta^{-1} b^{(t+8)} \pmod{g^B(\beta)},$$

where α, β are roots of the two generating polynomials $g^A(\alpha)$ and $g^B(\beta)$, respectively. Such a construction has the maximum cycle of length $2^{512} - 1$.

Every time when updating the LFSR part, LFSR-A and LFSR-B are clocked eight times, thus half of the states will be updated. After that, the two taps $T1$ and $T2$, which are formed by considering $(b_{15}, b_{14}, \dots, b_9, b_8)$, and $(a_7, a_6, \dots, a_1, a_0)$ as two 128-bit words, are fed to the FSM.

The FSM has three 128-bit registers, denoted $R1$, $R2$ and $R3$. It takes $T1, T2$ as inputs and produces a 128-bit keystream word z by the expression below,

$$z^{(t)} = (R1^{(t)} \boxplus_{32} T1^{(t)}) \oplus R2^{(t)}. \quad (1)$$

The three registers are then updated as follows:

$$R2^{(t+1)} = \text{AES}_R(R1^{(t)}), \quad (2)$$

$$R3^{(t+1)} = \text{AES}_R(R2^{(t)}), \quad (3)$$

$$R1^{(t+1)} = \sigma(R2^{(t)} \boxplus_{32} (R3^{(t)} \oplus T2^{(t)})), \quad (4)$$

where $\text{AES}_R()$ is one AES encryption round and σ is a byte-oriented permutation defined as $\sigma = [0, 4, 8, 12, 1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15]$. The AES encryption rounds and \boxplus_{32} provide the source of non-linearity.

The design document has also specified the initialisation phase and AEAD (Authenticated encryption with associated data) mode, as they are not relevant to our attacks, we skip the details but refer to the design document [EJMY19].

Notations and Expressions. We give some notations and expressions here which will be frequently used in the guess-and-determine attacks and linear cryptanalysis.

We use $(R1, R2, R3)$ and $(A0, A1, B0, B1)$ to denote the values of the registers in FSM and in LFSR, respectively, at some specific time t , where $A0$ ($B0$) and $A1$ ($B1$) are the low and high 128 bits of LFSR-A (LFSR-B), respectively. Thus, these seven variables are all 128-bit long. We can then get the following expressions:

$$\begin{aligned} B1^{(t-1)} &= B0, & B0^{(t+1)} &= B1, & B1^{(t+1)} &= A0 \oplus l_\beta(B0) \oplus h_\beta(B1), \\ A0^{(t+1)} &= A1, & R1^{(t-1)} &= \text{AES}_R^{-1}(R2), & R2^{(t-1)} &= \text{AES}_R^{-1}(R3), \end{aligned}$$

where $\text{AES}_R^{-1}()$ is the inverse of one AES encryption round. Here l_β and h_β are two linear operations relevant to the update of the LFSR, and are defined as below:

$$l_\beta(X) = (\beta(X_{15||14}) || \dots || \beta(X_{1||0})) \oplus X_{\gg 3.2}, \quad (5)$$

$$h_\beta(X) = (\beta^{-1}(X_{15||14}) || \dots || \beta^{-1}(X_{1||0})) \oplus X_{\ll 5.2}, \quad (6)$$

where X is a 128-bit variable and $X_{\ll k}, X_{\gg k}$ denote the left and right shift by k bytes, respectively. The multiplication operation with β or β^{-1} are applied to every 16-bit word independently over the field of LFSR-B. They can be expressed as the multiplication of the bit vector of the word and the binary 16×16 -bit matrices of β or β^{-1} . The binary matrix representations of β and β^{-1} are given in Appendix A. The explicit expressions of $l_\beta(X)$ and $h_\beta(X)$ in bytes are given in Appendix B.

The expressions for three consecutive keystream words at clock $t-1, t$ and $t+1$, which will be frequently used in our attacks, are derived as follows:

$$\begin{aligned} z^{(t-1)} &= (\text{AES}_R^{-1}(R2) \boxplus_{32} B0) \oplus \text{AES}_R^{-1}(R3), \\ z^{(t)} &= (R1 \boxplus_{32} B1) \oplus R2, \\ z^{(t+1)} &= (\sigma(R2 \boxplus_{32} (R3 \oplus A0)) \boxplus_{32} (A0 \oplus l_\beta(B0) \oplus h_\beta(B1))) \oplus \text{AES}_R(R1). \end{aligned} \quad (7)$$

3 The first guess-and-determine attack in $O(2^{384})$

In this section, we present our first GnD attack with complexity $O(2^{384})$, where the value inside notation $O()$ indicates the number of basic operations, which will be explained later. We first introduce some basics about guess-and-determine attacks, which apply to our second GnD attack in Section 4 as well. We then describe the attack in details and discuss its complexity.

3.1 Basics about guess-and-determine attacks

In a guess-and-determine attack, one *guesses* some variables and *determines* others according to some predefined relationships. In a GnD attack against a stream cipher, if all the variables in the whole state could be determined through guessing a number t of bits, where t is smaller than the security level, the attack is then faster than exhaustive key search. In this paper, we call every ordered tuple of the values of the guessed and further determined variables a *guessing path* or a *guess-and-determine path*, and use *end-nodes* to denote the end points of the guessing paths.

Usually, the complexity of a GnD attack is presented as 2^t , if one simply loops over all the possible values of the chosen variables for guessing. However, we notice that by guessing the variables in a careful order, one can either guess fewer variables or truncate some guessing paths in which the guessed and determined variables fail to satisfy some equation constraints in the middle. In the latter case, we can immediately trace back without going further and turn to guess another value, thus the complexity could be reduced.

For example, consider the simplest loop in the pseudo-code in Listing 1, where x, y, z are three 8-bit variables, it is straightforward to get that the complexity is $T = 2^{24}$.

```
T = 0;
for (x=0; x<256; x++)
  for (y=0; y<256; y++)
    for (z=0; z<256; z++)
      { T = T + 1;
        ...
      }
```

Listing 1: A simple GnD loop

However, for a different loop shown in Listing 2, where $L1[x]$ are lists depending on the specific values of x and $L2[x, y]$ are lists depending on the values of x, y , the size of the loop is not fixed but rather depends on the lengths of lists $L1[x]$ and $L2[x, y]$. For example, for a specific value of x , after we have gone through every value of $L1[x]$ for y (and correspondingly subsequent z), we can immediately trace back to another x value, instead of considering all the 256 values of y . In this case, the complexity is not simply 2^{24} , but instead the number of valid looping paths.

```
T = 0;
for (x=0; x<256; x++)
  for (y = L1[x].first; y!=NULL; y=y->next)
    for (z = L2[x, y].first; z!=NULL; z=z->next)
      { T = T + 1;
        ...
      }
```

Listing 2: A more complex GnD loop

Thus the complexity of a guess-and-determine attack could be expressed as $c \cdot T$, where c is some constant coefficient which we will explain later, and T is not just the size of the guessing loop, but rather dominated by the number of guessing paths that the attack algorithm will reach an end-node. If the exact value of T is infeasible to compute, the average value of it over the guessed variables is instead considered.

We will use the term **enumeration** to denote going through all the valid guess-and-determine paths, and the size/length of such an enumeration will decide the GnD complexity T . We would like to mention that the organisation of an *enumeration* may not be only plain loops, but some more sophisticated algorithms, e.g., *enumeration by recursion*, in which we adopt a recursion algorithm to explore all the solutions satisfying a certain equation or a system of conditions.

The other term c indicates some constant complexity, which solely depends on the concrete platform and the operations how other values are determined from the known ones. For example, the value of c for computing D given A, B through $D = A \oplus B$ or $A = (D \boxplus B) \oplus (D \oplus S(B))$ (S denotes S-box operation) will be different. Obviously, the complexity for the former example can be ignored as it almost consumes nothing, thus $c = 1$; however, for the latter case, it is not trivial to get the value of D directly, and *enumerations* or some other techniques are required. Thus, the cost for simple *derivations* are normally ignored, while if a *derivation* involves *enumeration*, the complexity of it should be included.

3.2 Steps of the first GnD attack

In our first GnD attack, we guess three 128-bit state variables $R1, R2, B0$ and use three consecutive keystream words to determine three more variables, $R3, B1$ and $A0$. The derivations for $R3$ and $B1$ are simple, while tricky for $A0$. We show how we derive $A0$ in a smart way in Section 3.2.2. After that, we use one more keystream word $z^{(t+2)}$ to determine the final state variable $A1$ using the same way for deriving $A0$, and finally use three additional keystream words to verify the correct guess. In total, seven 128-bit keystream words are required to determine the seven 128-bit state variables. A simplified flowchart of this GnD attack can be found in Appendix E.

3.2.1 Initial guessing set and derivations

We consider the three consecutive keystream words given in Equation 7 and introduce two intermediate 128-bit variables, C and D , which are defined as follows:

$$C = l_\beta(B0) \oplus h_\beta(B1), \quad (8)$$

$$D = \sigma(R2 \boxplus_{32} (R3 \oplus A0)) \boxplus_{32} (A0 \oplus C). \quad (9)$$

Correspondingly, the three keystream words in Equation 7 can be rewritten as:

$$\begin{aligned} z^{(t-1)} &= (\text{AES}_R^{-1}(R2) \boxplus_{32} B0) \oplus \text{AES}_R^{-1}(R3) \\ z^{(t)} &= (R1 \boxplus_{32} B1) \oplus R2 \\ z^{(t+1)} &= \text{AES}_R(R1) \oplus D \end{aligned} \quad (10)$$

There are six unknown variables in Equation 10, and to determine all of them, one has to guess not less than three. Since $R1$ and $R2$ appear twice in the expressions, we prefer to first guess them. Let us initially guess $(R1, R2, B0)$ with complexity 2^{384} . Then the variables below will be directly determined:

$$R3 \text{ from } : z^{(t-1)} = (\text{AES}_R^{-1}(R2) \boxplus_{32} B0) \oplus \text{AES}_R^{-1}(R3),$$

$$B1 \text{ from } : z^{(t)} = (R1 \boxplus_{32} B1) \oplus R2,$$

$$D \text{ from } : z^{(t+1)} = \text{AES}_R(R1) \oplus D.$$

Thus, all the variables in Equation 10 are known, either through guessing or determining. Besides, the intermediate variable C in Equation 8 is also determined, and our last step is to determine the values of the remaining two state variables, $A0$ and $A1$.

If we find an efficient way to enumerate all the solutions for A_0 (and A_1) without additional guesses, the overall GnD complexity will be exactly 2^{384} . This can be initially understood from a simple observation: there are six unknowns in the three consecutive keystream words in Equation 10, thus the space of unknowns $\mathbb{F}_2^{6 \cdot 128}$ is mapped to the space of three keystream words $\mathbb{F}_2^{3 \cdot 128}$. Therefore, the expected number of combinations of the six unknowns satisfying 384 bits of the keystream, is 2^{384} . Since we guess 384 bits of the internal state, the expected number of solutions corresponding to each guess in such an enumeration is exactly one on average, thus there are 2^{384} valid guessing paths.

We next show how we efficiently find the solutions of A_0 in Section 3.2.2 and A_1 in Section 3.2.3.

3.2.2 Deriving A_0 using a 10-step recursive enumeration

A_0 is determined using Equation 9, while we would like to note that even though other variables in Equation 9 are fixed, the value of A_0 might not be uniquely or directly determined as A_0 appears twice in the equation with non-linear operations in-between. So the task now is to efficiently find the solutions for A_0 in Equation 9, which we rewrite with highlight as below:

$$D = \sigma(R_2 \boxplus_{32} (R_3 \oplus A_0)) \boxplus_{32} (A_0 \oplus C). \quad (11)$$

We start searching for solutions of A_0 in Equation 11 in a byte-wise fashion. Each byte of D , D_i ($15 \geq i \geq 0$) is expressed as:

$$D_i = (R_{2j} \boxplus_8 (R_{3j} \oplus A_{0j}) \boxplus_8 u_j) \boxplus_8 (A_{0i} \oplus C_i) \boxplus_8 v_i, \quad j = \sigma(i), \quad (12)$$

where $u_j, v_i \in \{0, 1\}$ are carry bits that may arrive from arithmetical additions of the previous bytes. We call these byte-wise equations as D-equations. Note that some of these carry values are already known: $u_k = v_k = 0$ for $k = 0, 4, 8, 12$. For other carriers, we do not have to guess them if we derive the bytes of A_0 in a careful order in 10 steps as shown in Table 2.

Table 2: The 10 steps to derive A_0 .

Step 0:	$D_0 = (R_{2_0} \boxplus_8 (R_{3_0} \oplus A_{0_0}) \boxplus_8 u_0) \boxplus_8 (A_{0_0} \oplus C_0) \boxplus_8 v_0$ where $u_0 = v_0 = 0$ derive $\rightarrow (A_{0_0}, u_1, v_1)$
Step 1:	$D_1 = (R_{2_4} \boxplus_8 (R_{3_4} \oplus A_{0_4}) \boxplus_8 u_4) \boxplus_8 (A_{0_1} \oplus C_1) \boxplus_8 v_1$ $D_4 = (R_{2_1} \boxplus_8 (R_{3_1} \oplus A_{0_1}) \boxplus_8 u_1) \boxplus_8 (A_{0_4} \oplus C_4) \boxplus_8 v_4$ where $u_4 = v_4 = 0$ and u_1, v_1 are known from Step 0 derive $\rightarrow (A_{0_1}, A_{0_4}, u_2, v_2, u_5, v_5)$
Step 2:	$D_5 = (R_{2_5} \boxplus_8 (R_{3_5} \oplus A_{0_5}) \boxplus_8 u_5) \boxplus_8 (A_{0_5} \oplus C_5) \boxplus_8 v_5$ where u_5, v_5 are known from Step 1 derive $\rightarrow (A_{0_5}, u_6, v_6)$
Step 3:	$D_2 = (R_{2_8} \boxplus_8 (R_{3_8} \oplus A_{0_8}) \boxplus_8 u_8) \boxplus_8 (A_{0_2} \oplus C_2) \boxplus_8 v_2$ $D_8 = (R_{2_2} \boxplus_8 (R_{3_2} \oplus A_{0_2}) \boxplus_8 u_2) \boxplus_8 (A_{0_8} \oplus C_8) \boxplus_8 v_8$ where $u_8 = v_8 = 0$ and u_2, v_2 are known from Step 1 derive $\rightarrow (A_{0_2}, A_{0_8}, u_3, v_3, u_9, v_9)$
Step 4:	$D_3 = (R_{2_{12}} \boxplus_8 (R_{3_{12}} \oplus A_{0_{12}}) \boxplus_8 u_{12}) \boxplus_8 (A_{0_3} \oplus C_3) \boxplus_8 v_3$ $D_{12} = (R_{2_3} \boxplus_8 (R_{3_3} \oplus A_{0_3}) \boxplus_8 u_3) \boxplus_8 (A_{0_{12}} \oplus C_{12}) \boxplus_8 v_{12}$ where $u_{12} = v_{12} = 0$ and u_3, v_3 are known from Step 3 derive $\rightarrow (A_{0_3}, A_{0_{12}}, u_{13}, v_{13})$
Step 5:	$D_6 = (R_{2_9} \boxplus_8 (R_{3_9} \oplus A_{0_9}) \boxplus_8 u_9) \boxplus_8 (A_{0_6} \oplus C_6) \boxplus_8 v_6$ $D_9 = (R_{2_6} \boxplus_8 (R_{3_6} \oplus A_{0_6}) \boxplus_8 u_6) \boxplus_8 (A_{0_9} \oplus C_9) \boxplus_8 v_9$

	where	u_6, v_6, u_9, v_9 are known from Steps 2 and 3
	derive	$\rightarrow (A0_6, A0_9, u_7, v_7, u_{10}, v_{10})$
Step 6:	D_{10}	$= (R2_{10} \boxplus_8 (R3_{10} \oplus A0_{10}) \boxplus_8 u_{10}) \boxplus_8 (A0_{10} \oplus C_{10}) \boxplus_8 v_{10}$
	where	u_{10}, v_{10} are known from Step 5
	derive	$\rightarrow (A0_{10}, u_{11}, v_{11})$
Step 7:	D_7	$= (R2_{13} \boxplus_8 (R3_{13} \oplus A0_{13}) \boxplus_8 u_{13}) \boxplus_8 (A0_7 \oplus C_7) \boxplus_8 v_7$
	D_{13}	$= (R2_7 \boxplus_8 (R3_7 \oplus A0_7) \boxplus_8 u_7) \boxplus_8 (A0_{13} \oplus C_{13}) \boxplus_8 v_{13}$
	where	u_7, v_7, u_{13}, v_{13} are known from Steps 4 and 5
	derive	$\rightarrow (A0_7, A0_{13}, u_{14}, v_{14})$
Step 8:	D_{11}	$= (R2_{14} \boxplus_8 (R3_{14} \oplus A0_{14}) \boxplus_8 u_{14}) \boxplus_8 (A0_{11} \oplus C_{11}) \boxplus_8 v_{11}$
	D_{14}	$= (R2_{11} \boxplus_8 (R3_{11} \oplus A0_{11}) \boxplus_8 u_{11}) \boxplus_8 (A0_{14} \oplus C_{14}) \boxplus_8 v_{14}$
	where	$u_{11}, v_{11}, u_{14}, v_{14}$ are known from Steps 6 and 7
	derive	$\rightarrow (A0_{11}, A0_{14}, u_{15}, v_{15})$
Step 9:	D_{15}	$= (R2_{15} \boxplus_8 (R3_{15} \oplus A0_{15}) \boxplus_8 u_{15}) \boxplus_8 (A0_{15} \oplus C_{15}) \boxplus_8 v_{15}$
	where	u_{15}, v_{15} are known from Step 8
	derive	$\rightarrow (A0_{15})$

For each of the 2^{384} values of the initial guessing set $(R1, R2, B0)$, there could be different numbers, either zero or nonzero, of solutions for $A0$. Most of the guessing values will even not pass the first step in Table 2 as no valid solutions exist for the first equation, and we can immediately trace back to guess another value of $(R1, R2, B0)$; while other guessing values could have more than one solutions. However, we will show in Section 3.3.1 that the average number of solutions over $(R1, R2, B0)$ is exactly one. The simplest way to enumerate all solutions is to use a recursion procedure. For example, we can loop for all solutions in the first step, and for each valid solution we recursively call the second step, and so on.

If we only use simple loops for enumerating all the solutions in each step in Table 2, the constant c in the complexity will be quite big ($c \approx 2^8$), but later in Section 3.3.3 we will show how to reduce c to much smaller in efficient ways.

3.2.3 Deriving $A1$ and final verification

In the above initial guess and enumeration, we know six out of seven 128-bit variables of the state. There will be 2^{384} guessing paths that arrive to this final stage of the attack. In order to derive the final 128-bit state variable $A1$, we simply use the fourth keystream word $z^{(t+2)}$:

$$z^{(t+2)} = (R1^{(t+2)} \boxplus_{32} B1^{(t+2)}) \oplus R2^{(t+2)},$$

where

$$\begin{aligned} R1^{(t+2)} &= \sigma(R2^{(t+1)} \boxplus_{32} (R3^{(t+1)} \oplus A1)) = \sigma(\text{AES}_R(R1) \boxplus_{32} (\text{AES}_R(R2) \oplus A1)), \\ R2^{(t+2)} &= \text{AES}_R(R1^{(t+1)}) = \text{AES}_R(\sigma(R2 \boxplus_{32} (R3 \oplus A0))), \\ B1^{(t+2)} &= A0^{(t+1)} \oplus l_\beta(B0^{(t+1)}) \oplus h_\beta(B1^{(t+1)}) \\ &= A1 \oplus l_\beta(B1) \oplus h_\beta(A0 \oplus l_\beta(B0) \oplus h_\beta(B1)). \end{aligned}$$

Denote $C' = l_\beta(B1) \oplus h_\beta(A0 \oplus l_\beta(B0) \oplus h_\beta(B1))$, then we can get the equation for $A1$:

$$z^{(t+2)} \oplus R2^{(t+2)} = \sigma(R2^{(t+1)} \boxplus_{32} (R3^{(t+1)} \oplus A1)) \boxplus_{32} (A1 \oplus C').$$

One can see that the equation above has exactly the same form as the expression for $A0$ in Equation 11, and therefore, we could use the ten steps in Table 2 to enumerate all the

solutions of $A1$. The distribution of number of solutions will be the same and there will be one solution in average for each tuple of values of the other variables.

So far, we have guessed three state variables and determined the four others, such that the values of the seven 128-bit words satisfy the four consecutive 128-bit keystream words. The number of valid combinations of values is 2^{384} and in order to decide which one is correct, we use three additional subsequent keystream words to verify. The verification only involves simple derivations thus the cost can be ignored.

3.3 Discussion on the complexity

3.3.1 Study of the two types of D-equations in the 10 steps

In this section, we compute the distribution of the number of solutions for the D-equations in Table 2 and show that the average value is one. This helps us to derive the total complexity.

In the equation $D_i = (R2_j \boxplus_8 (R3_j \oplus A0_j) \boxplus_8 u_j) \boxplus_8 (A0_i \oplus C_i) \boxplus_8 v_i$, the input carry bits u_j, v_i can be removed by setting $R2'_j = R2_j \boxplus u_j$ and $D'_i = D_i \boxplus v_i$, which will not influence the distribution or average value of the number of solutions. The ten groups of D-equations in Table 2 can be divided into two equivalent types, which we denote as *Type-1* and *Type-2* equations.

Type-1 equations have the form below:

$$A = (B \boxplus_n (C \oplus X)) \boxplus_n (X \oplus D),$$

where (A, B, C, D) are n -bit variables and X is the unknown that we need to enumerate to find the solutions. Such *Type-1* equations appear in Steps $\{0, 2, 6, 9\}$.

Type-2 equations have the form below:

$$\begin{aligned} A_1 &= (B_1 \boxplus_n (C_1 \oplus X_1)) \boxplus_n (X_2 \oplus D_1), \\ A_2 &= (B_2 \boxplus_n (C_2 \oplus X_2)) \boxplus_n (X_1 \oplus D_2), \end{aligned}$$

where X_1, X_2 are the unknown variables that we want to enumerate for the solutions, while others are n -bit variables. Such *Type-2* equations appear in Steps $\{1, 3, 4, 5, 7, 8\}$.

For both types of equations, we have computed the distribution tables of the numbers of solutions for the unknown X -bytes, given that the other known variables are uniformly distributed. We exhaustively (with some optimisations and cut-offs) try all the values of the known variables, and count all the solutions for the unknowns.

Table 3 presents the probabilities of X having different number of solutions for *Type-1* equations corresponding to a random tuple of (A, B, C, D) over binary fields of different dimensions n . The probabilities are derived through $p = x/f$, where x 's are the corresponding integers in the table and f is some normalization factor. The probability of having at least one solution when $n = 8$ can be computed easily, which is $2^{-3.91}$. This means that in Equation 11, only $2^{-3.91}$ of the combinations of $(R2, R3, C, D)$ will result into valid solutions and continue with Step 1, and so on; while for the remaining majority combination values, we just stop and trace back to the last step.

We can further compute the average number of solutions, Avr , as below:

$$Avr = \sum_{i=0}^{2^n-1} i \cdot Pr\{\#Solutions = i\}.$$

The computed average value is exactly one.

We can derive the distribution and average value of number of solutions for *Type-2* equations as well using a similar way. The distribution table under different n values is

Table 3: Distribution table of the number of solutions of X for *type-1* equations.

#Solutions normalization factor f	n=1 2^1	n=2 2^3	n=3 2^5	n=4 2^7	n=5 2^9	n=6 2^{11}	n=7 2^{13}	n=8 2^{15}
0	1	5	23	101	431	1805	7463	30581
2	1	2	4	8	16	32	64	128
4		1	4	12	32	80	192	448
8			1	6	24	80	240	672
16				1	8	40	160	560
32					1	10	60	280
64						1	12	84
128							1	14
256								1

given in Appendix D. The probability of having at least one solution is $2^{-3.53}$ and the average number of solutions is one as well.

Since the ten tuples of equations are independent to each other (except the carriers, but the carriers do not influence the probability of having solutions), the probability of $A0$ having at least one solution is computed as $2^{-3.53 \times 6 - 3.91 \times 4} = 2^{-36.84}$. This means that only a small fraction, i.e., $2^{-36.84}$, of the 2^{384} initial guesses of $(R1, R2, B0)$ will actually have solutions for $A0$, while for other guessing values, the guessing process can be just terminated here. However, when $A0$ has valid solutions, the number of solutions will be around $2^{36.84}$ in average but, again, the overall average number of solutions is exactly one.

3.3.2 The total attack complexity

The large fraction of the guesses, i.e., $2^{384} \cdot (1 - 2^{-3.91})$, will fail in Step 0 in Table 2, as Step 0 involves solving a *Type-1* equation and the probability of having solutions is $2^{-3.91}$. The remaining small fraction of the guesses, i.e., $2^{384} \cdot 2^{-3.91} \approx 2^{380.09}$, will advance to Step 1. The number of solutions in Step 0 will be $2^{3.91}$ in average, thus the total number of guessing paths that will arrive Step 1 is again $2^{380.09} \cdot 2^{3.91} \approx 2^{384}$. The same observation applies to every step in Table 2. Thus for 2^{384} input combinations to the recursive enumeration algorithm for deriving $A0$, we will get 2^{384} possible solutions, exactly one per guess in average.

For the final step to determine $A1$, the situation will be the same, i.e., the majority of the derived six-word tuple will fail the first step, and only a small fraction will advance to the next step, and so on. The average number of solutions is again one and there are 2^{384} valid guessing paths. Thus the total complexity of the GnD attack is $O(2^{384})$.

3.3.3 Further reducing the complexity constant c

The complexity is written as $c \cdot 2^{384}$ where c is the complexity of operations involved in each guessing path, mainly lies in solving the D-equations of either type. The simplest way to solve the equations is to loop over all the values of the unknown bytes and check if the values satisfy the equations, which, however, can be improved in a number of ways listed below.

Loop for one-byte solution in $2 \cdot 8$ instead of 2^8 . Recall that to enumerate all solutions for $A0$ we make a byte-wise recursion of depth ten, and in each step we loop over the byte sub-solutions. However, we can make an even deeper recursion and search for solutions of each *bit* of $A0_i$. This will shrink the constant complexity c from 256 down to $2 \cdot 8$, since now we only need to get the solutions of the eight bits of a byte using a recursive way. For each bit we try either 0 or 1, then go into the next bit with considering the carriers, and so on.

So we can enumerate the 128-bit unknown $A0$ by deriving each of the 128 bits in a recursive step. We have actually implemented such a bit-oriented recursive enumeration algorithm, which is given in [Appendix C](#).

Precomputed lookup tables. We can also precompute lookup tables helping to instantly give the list of sub-solutions for each tuple of D-equations given known variables. The tables record all the possible values of the known variables and the corresponding solutions for the unknowns, thus one can immediately get the solutions by looking up the tables.

The smallest table will be of size $2^{32} \rightarrow 256 \times 10$ bits for Step 0, where each entry corresponds to one value of the known variables, 256 is the maximum number of possible solutions corresponding to one entry, and 10 bits correspond to the value of one unknown (one byte) and two carriers (two bits). There will be exactly 2^{32} valid records of size 10 bits in the table. An example of the smallest table is as below:

$$T_0[R2_0, R3_0, C_0, D_0] \rightarrow \{A0_0, u_1, v_1\}.$$

The largest table is of size $2^{68} \rightarrow 256 \times 20$ bits in Step 5:

$$T_5[R2_6, R3_6, C_6, D_6, u_6, v_6, R2_9, R3_9, C_9, D_9, u_9, v_9] \rightarrow \{A0_6, A0_9, u_7, v_7, u_{10}, v_{10}\}.$$

Truncating guessing paths reaching the 10-step stage for deriving $A0$. The number of guessing paths that reach the 10-step stage for deriving $A0$ can be further reduced by guessing the variables in the initial set in bytes, instead of 128 bits, in a careful order. We give a simple example here, and there exist some more tricky ones.

We first guess the following 25 bytes and 2 bits in complexity 2^{202} :

$$R1_{0,1,3,4,5,9,10,14,15}, R2_{0,1,4,5}, R3_{0,1,4,5}, B0_{0,1,4,5,6,7,10,11}, w_{0,4},$$

where $w_{0,4}$ are two carry bits for 32-bit additions. Then the following variables can be derived:

$$\begin{aligned} D_0 &= z_0^{(t+1)} \oplus (2 \cdot S(R1_0) \oplus 3 \cdot S(R1_5) \oplus 1 \cdot S(R1_{10}) \oplus 1 \cdot S(R1_{15})), \\ D_1 &= z_1^{(t+1)} \oplus (1 \cdot S(R1_0) \oplus 2 \cdot S(R1_5) \oplus 3 \cdot S(R1_{10}) \oplus 1 \cdot S(R1_{15})), \\ D_4 &= z_4^{(t+1)} \oplus (1 \cdot S(R1_3) \oplus 2 \cdot S(R1_4) \oplus 3 \cdot S(R1_9) \oplus 1 \cdot S(R1_{14})), \\ D_5 &= z_5^{(t+1)} \oplus (1 \cdot S(R1_3) \oplus 1 \cdot S(R1_4) \oplus 2 \cdot S(R1_9) \oplus 3 \cdot S(R1_{14})), \\ B1_{0||1} &= (z_{0||1}^{(t)} \oplus R2_{0||1}) \boxminus_{32} R1_{0||1} \boxminus_{32} w_0, \\ B1_{4||5} &= (z_{4||5}^{(t)} \oplus R2_{4||5}) \boxminus_{32} R1_{4||5} \boxminus_{32} w_4, \\ C_{0||1} &= \beta B0_{0||1} \oplus B0_{6||7} \oplus \beta^{-1} B1_{0||1}, \\ C_{4||5} &= \beta B0_{4||5} \oplus B0_{10||11} \oplus \beta^{-1} B1_{4||5}. \end{aligned}$$

With the set of the guessed and determined values, we can now check whether a solution for bytes $A0_0, A0_1, A0_4, A0_5$ exists, in the first three steps in [Table 2](#). The probability of valid solutions, denoted p_{0-2} , can be computed as $p_{0-2} = 2^{-3.91 \times 2 - 3.52} = 2^{-11.34}$. If no solutions exist, we just roll back and make another guess; otherwise we guess the remaining 23 ($= 48 - 25$) bytes of the initial guessing set and run the 10-step algorithm to enumerate all values of $A0$. The total number of nodes that will arrive to the 10-step stage will be:

$$T = 2^{200+2} + (p_{0-2} \cdot 2^{200+2}) \cdot 2^{184-2} = 2^{202} + p_{0-2} \cdot 2^{384}.$$

This means that only $2^{372.66}$ guessing paths (out of 2^{384}) will reach the 10-step stage for enumerating $A0$. However, the total complexity will still be 2^{384} , as the fact that there are 2^{384} solutions satisfying the three consecutive keystream words remains unchanged. [Figure 2](#) gives an illustration of the first GnD attack and the ‘‘effect’’ of the idea to do a pre-test after guessing only 202 bits.

4 The second guess-and-determine attack in $O(2^{378.16})$

In this section, we provide a second guess-and-determine attack which can further reduce the complexity by using one additional keystream block, $z^{(t-2)}$, as side information to truncate more guessing paths. The improvement over the first GnD attack is not very big, but the idea of exploiting more equation constraints to truncate guessing paths itself is interesting, and our second GnD attack serves as a direct illustration of this idea.

4.1 Use $z^{(t-2)}$ to truncate more guessing paths

In the first GnD attack we have four equations in seven 128-bit unknowns, we guess three of them and determine the others. For each guessing value, one solution in average will be derived and thus there are 2^{384} valid solutions in total. Even if we apply the idea of first guessing 202 bits to truncate some guessing paths at a even earlier stage, we still have to run through all 2^{384} solutions in average. Using the next three keystream words we can verify which guess is correct, thus we can recover all the seven state variables by guessing three 128-bit variables and the complexity is 2^{384} .

If we want to further reduce the complexity, we could try to see if we can truncate more guessing paths of the initial 384-bit guessing set. Besides the guessing paths that have no solutions in the D-equations, we can also truncate those having solutions for D-equations but not satisfying some other side equation constraints, such that we will have a smaller than 2^{384} number of solutions for $A0$ overall.

Specifically, we use one additional keystream word at clock $t - 2$, i.e., $z^{(t-2)}$, to impose more constraints and truncate more guessing paths. The expression of $z^{(t-2)}$ is shown below:

$$z^{(t-2)} = (R1^{(t-2)} \boxplus_{32} B1^{(t-2)}) \oplus R2^{(t-2)},$$

where $R1^{(t-2)}, B1^{(t-2)}, R2^{(t-2)}$ are derived as follows:

$$\begin{aligned} R1^{(t-2)} &= \text{AES}_R^{-1}(R2^{(t-1)}) = \text{AES}_R^{-1}(\text{AES}_R^{-1}(R3)), \\ B1^{(t-2)} &= B0^{(t-1)}, \\ R2^{(t-2)} &= \text{AES}_R^{-1}(R3^{(t-1)}) = \text{AES}_R^{-1}((\sigma(R1) \boxplus_{32} R2^{(t-1)}) \oplus A0^{(t-1)}) \\ &= \text{AES}_R^{-1}((\sigma(R1) \boxplus_{32} \text{AES}_R^{-1}(R3)) \oplus A0^{(t-1)}). \end{aligned}$$

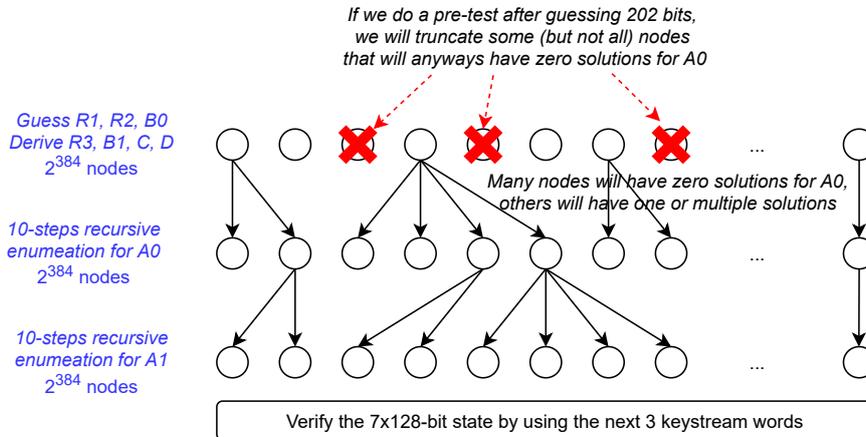


Figure 2: Illustration of the first GnD attack.

According to the LFSR update functions, we can derive:

$$A0^{(t-1)} = B1 \oplus l_\beta(B0^{(t-1)}) \oplus h_\beta(B1^{(t-1)}) = B1 \oplus l_\beta(B0^{(t-1)}) \oplus h_\beta(B0).$$

Thus $z^{(t-2)}$ can be written as an equation in one unknown variable $B0^{(t-1)}$ (given the other values are known or guessed):

$$z^{(t-2)} = \underbrace{(\text{AES}_R^{-1}(\text{AES}_R^{-1}(R3)))}_{X} \boxplus_{32} B0^{(t-1)} \\ \oplus \text{AES}_R^{-1}(\underbrace{(\sigma(R1) \boxplus_{32} \text{AES}_R^{-1}(R3)) \oplus h_\beta(B0) \oplus B1}_{Y} \oplus l_\beta(B0^{(t-1)})).$$

Using X, Y to denote the expressions in the brackets, we could simplify the above equation as $z^{(t-2)} = (X \boxplus_{32} B0^{(t-1)}) \oplus \text{AES}_R^{-1}(Y \oplus l_\beta(B0^{(t-1)}))$. Similar to the situation for $A0$ in our first GnD attack, $B0^{(t-1)}$ appears twice with non-linear operations in-between, thus it can have different numbers of solutions given specific values of X, Y . If we change to initially guess the two 128-bit variables X and Y , the expression of $z^{(t-2)}$ can help to truncate more guessing paths that have no valid solutions for $B0^{(t-1)}$. Specifically, for each guessing value of the (X, Y) pair, if we can immediately give a binary answer, i.e., Yes or No, about whether there is at least one solution for $B0^{(t-1)}$, we can discard those (X, Y) values with no solutions, and only continue guessing the third 128-bit variable for the others. Note that we do not enumerate solutions for $B0^{(t-1)}$ in $z^{(t-2)}$, since then we would get the same complexity 2^{384} as the first GnD attack, and we will later show how we efficiently get the binary answer of whether valid solutions exist without enumeration in Section 4.2.1. Actually, we will guess $(X, B0^{(t-1)})$ instead of (X, Y) there, but we still first describe the idea by guessing (X, Y) since it is easier to illustrate how $z^{(t-2)}$ is exploited to truncate more guessing paths.

Let p_z denote the probability that $B0^{(t-1)}$ has solutions in the equation of $z^{(t-2)}$, then the total complexity of the second GnD attack would be computed as:

$$T = \underbrace{2^{256}}_{\text{guess } X, Y} \cdot \left(\underbrace{(1 - p_z)}_{\text{"No"}} + \underbrace{p_z}_{\text{"Yes"}} \cdot \underbrace{2^{128}}_{\text{3rd guess}} \right) \approx p_z \cdot 2^{384}.$$

We have derived the specific value of p_z in Appendix G, which is $2^{-5.84}$, thus the total complexity of the second GnD attack is around $O(2^{384-5.84}) \approx O(2^{378.16})$.

4.2 Scenario of the second GnD attack

The flowchart of the second GnD attack is given in Appendix E, which follows the steps below:

- (1) Guess X and Y with complexity 2^{256} ;
- (2) For each (X, Y) value, check if $B0^{(t-1)}$ in $z^{(t-2)}$ has solutions: if yes, continue with guessing the third variable in the next step; otherwise roll back to the last step;
- (3) Guess $B0$ in complexity 2^{128} and further derive $R2, R3$ as below:

$$R3 \text{ from: } X = \text{AES}_R^{-1}(\text{AES}_R^{-1}(R3))$$

$$R2 \text{ from: } z^{(t-1)} = (\text{AES}_R^{-1}(R2) \boxplus_{32} B0) \oplus \text{AES}_R^{-1}(R3)$$

This step will be entered $p_z \cdot 2^{256}$ times in average.

- (4) For each valid combination of $(X, Y, B0)$, we get the following two equations in two unknowns $R1$ and $B1$:

$$\begin{aligned} z^{(t)} \oplus R2 &= R1 \boxplus_{32} B1, \\ Y \oplus h_\beta(B0) &= (\sigma(R1) \boxminus_{32} \text{AES}_R^{-1}(R3)) \oplus B1. \end{aligned} \quad (13)$$

We check if $B1, R1$ have valid solutions given other variables, and roll back if the answer is negative, otherwise we **enumerate** all solutions recursively. We have computed the distribution and average value of the number of solutions using the similar way for the D-equations in the first GnD attack and the details are given in [Appendix F](#). There is again one solution in average for each combination of the known variables. Similarly, lookup tables can be precomputed to help enumerate solutions efficiently.

- (5) Derive the D-equations in $A0$ and **enumerate** all the solutions recursively as done in the first GnD attack;
- (6) Derive the D-equations in $A1$ and **enumerate** all the solutions recursively as done in the first GnD attack;
- (7) Use the next three keystream words to verify the correct guess.

4.2.1 Guess $(X, B0^{(t-1)})$ instead of (X, Y)

In the first step, we need to give a binary answer about whether solutions exist for $B0^{(t-1)}$ in the equation:

$$z^{(t-2)} = (B0^{(t-1)} \boxplus_{32} X) \oplus \text{AES}_R^{-1}(l_\beta(B0^{(t-1)}) \oplus Y).$$

One simple way to achieve this is to run an enumeration algorithm on $B0^{(t-1)}$, and whenever a solution is found, we stop and move to the next step. This is similar to the process of computing p_z in [Appendix G](#). However, the process is actually an enumeration algorithm on $B0^{(t-1)}$ with complexity $2^{48-5.84}$, resulting in the total complexity even higher than $O(2^{384})$.

Instead, we can actually guess $(X, B0^{(t-1)})$ instead of (X, Y) , and Y can be uniquely determined given $(X, B0^{(t-1)})$. But it could happen that for different $(X, B0^{(t-1)})$ pairs, the values of Y are the same. So for every new X we must ensure that the value of Y is new, and skip the cases when the same (X, Y) pair has already been considered. Thus, for each new value of X we make a binary vector of length 2^{128} in which we flag (i.e., set to 1) those Y 's that have already been considered for this specific X . Thus, in step (1) in [Subsection 4.2](#), we guess $(X, B0^{(t-1)})$ and determine Y , and in step (2), we check if (X, Y) pair has already been flagged as 1: if so, we roll back to guess another value; otherwise, continue with guessing $B0$ in step (3). Other steps are just the same as before.

```

T = 0;
N = pow(2, 128); // 2 to the power of 128
char flag[N];
for(X = 0; X < N; ++X)
{
    for(i = 0; i < N; ++i)
        flag[i] = 0;
    for(B0 = 0; B0 < N; ++B0) // B0 at clock t-1
    {
        derive Y;
        if(flag[Y] == 0)
        {
            // we enter this branch with probability p_z in average
            flag[Y] = 1;
            for(B0t = 0; B0t < N; ++B0t)
                // guess the third unknown B0t: B0 at clock t
                {
                    T = T + 1; // complexity to enumerate all guess basis
                    (*)... further derivation and enumerations, Steps 3-7
                }
        }
    }
}

```

```

    }
  }
}

```

Listing 3: Outline of the second GnD attack.

Listing 3 gives the pseudo-code of the second GnD attack. It is easy to see that the number of times that the GnD attack arrives to the point (*) is $T \approx 2^{256} \cdot p_z \cdot 2^{128}$ where $p_z = 2^{-5.84}$, thus the complexity is about $O(2^{378})$.

5 New ideas in linear cryptanalysis of SNOW-V

The basic idea of linear cryptanalysis is to approximate the non-linear operations of a cipher as linear ones, and further to explore linear relationships either between keystream words, or between keystream words and initial states, which could result into a distinguishing attack or a correlation attack, respectively. Usually, such a linear approximation will introduce a noise, and the quality of the linear approximation is measured by the bias of this noise, which will directly influence the attack complexity. There are many ways to define the bias and derive the complexity, and in our attack, we use SEI as defined in [BJV04]. For a variable with distribution D , the SEI of it is computed as:

$$\epsilon(D) = |D| \cdot \sum_{i=0}^{|D|-1} \left(D[i] - \frac{1}{|D|} \right)^2,$$

where $D[i]$ is the occurrence probability of the value in the i -th entry. For a distribution with SEI $\epsilon(D)$, the number of samples required to distinguish the distribution from the uniform random distribution is in the order of $1/\epsilon(D)$ [BJV04].

In a distinguishing attack against LFSR-based stream ciphers, one has to cancel out the contribution from the linear part, i.e., the LFSR, thus introducing a biased property for the keystream. Typically, this is achieved by finding a low-weight (usually weights 3, 4 or 5) multiple of the generating polynomial and combining the keystream words at time instances corresponding to this multiple. The linear contribution will be cancelled and thus one biased keystream sample is constituted. By collecting enough many such samples, it is possible to distinguish this keystream sample sequence from random. The time instances corresponding to the multiple are very far away from each other, thus an extremely large length of keystream sequence corresponding to one (K, IV) pair is required.

In this section, we perform linear cryptanalysis of SNOW-V and propose a distinguishing attack with complexity $O(2^{303})$ against a reduced version in which the 32-bit adders are replaced with exclusive-OR. Our attack has the advantage that it does not need to combine keystream words at multiple time instances far away to build samples. Thus, the samples can be collected from short keystream sequences under different (K, IV) pairs.

5.1 Linear Approximation in SNOW-V

We first express the operations in the AES encryption round as $L \cdot S$, where S denotes S-box operation and L is the combination of the `ShiftRow` and `Mixcolumn` operations. Similarly, the inverse AES encryption round can be expressed as $S^{-1} \cdot L^{-1}$, where S^{-1} denotes the inverse S-box operation and L^{-1} is the combination of inverse `Mixcolumn` and inverse `ShiftRow` operations. L and L^{-1} can be expressed as two 16×16 -byte matrices, in which each entry is an element from \mathbb{F}_{2^8} . The expressions of L and L^{-1} are given in Appendix A. Besides, we replace \boxplus_{32} with \oplus , and make a substitution of the variables $R2, R3$ as $L \cdot R2, L \cdot R3$, respectively. Hence, $R2, R3$ are not the original variables, but for ease of reading, we still use the original notations.

Then the expressions of the three consecutive keystream words in Equation 7 can be rewritten as follows:

$$\begin{aligned} z^{(t-1)} &= S^{-1}(R2) \oplus B0 \oplus S^{-1}(R3), \\ z^{(t)} &= R1 \oplus B1 \oplus L \cdot R2, \\ z^{(t+1)} &= \sigma L \cdot R2 \oplus \sigma L \cdot R3 \oplus (\sigma A0 \oplus A0) \oplus l_\beta(B0) \oplus h_\beta(B1) \oplus L \cdot S(R1). \end{aligned}$$

The variables $B0, B1, A0$ are contributions from the LFSR, and we would like to cancel them out. To achieve so, we first apply two linear masks l_β, h_β , which can be expressed as two 128×128 binary matrices, to $z^{(t)}$ and $z^{(t-1)}$, respectively, and introduce a new 128-bit variable Y defined as below:

$$Y = l_\beta(z^{(t-1)}) \oplus h_\beta(z^{(t)}) \oplus z^{(t+1)}. \quad (14)$$

The contribution from the variables $B0$ and $B1$ is cancelled in Y , and what remains from the LFSR is only $(\sigma A0 \oplus A0)$. Now let us introduce ten byte-based variables from Y , shown below:

$$\begin{aligned} E_0 &= Y_0, & E_1 &= Y_1 \oplus Y_4, & E_2 &= Y_5, & E_3 &= Y_2 \oplus Y_8, & E_4 &= Y_6 \oplus Y_9, \\ E_5 &= Y_{10}, & E_6 &= Y_3 \oplus Y_{12}, & E_7 &= Y_7 \oplus Y_{13}, & E_8 &= Y_{11} \oplus Y_{14}, & E_9 &= Y_{15}, \end{aligned}$$

where Y_i is the i -th byte of Y . Each byte-wise expression E_k ($0 \leq k \leq 9$) cancels out the contribution from $A0$, and only the byte variables from registers $R1, R2, R3$ remain. Each of the above E_k terms can be expressed in a form as below:

$$\begin{aligned} E_k &= \bigoplus_{i=0}^{15} [l_{k,i}^{(1)} \cdot R1_i \oplus n_{k,i}^{(1)} \cdot S(R1_i)] \\ &\quad \oplus [l_{k,i}^{(2)} \cdot R2_i \oplus n_{k,i}^{(2)} \cdot S^{-1}(R2_i)] \oplus [l_{k,i}^{(3)} \cdot R3_i \oplus n_{k,i}^{(3)} \cdot S^{-1}(R3_i)], \end{aligned} \quad (15)$$

where $l_{k,i}^{(j)}, n_{k,i}^{(j)}$ ($j \in \{1, 2, 3\}, 0 \leq k \leq 9, 0 \leq i \leq 15$) are 8×8 binary matrices that can be derived following the expressions of Y and E terms. This means that each E_k can contain up to 48 independent noise terms of the form $ax \oplus bS(x)$, i.e., up to 48 approximations of the S-boxes or the inverse S-boxes. We can derive the expression for the total noise N as a linear combination of these ten E -bytes as follows:

$$N = c_0 \cdot E_0 \oplus c_1 \cdot E_1 \oplus \dots \oplus c_9 \cdot E_9,$$

where c_i 's are linear masking coefficients or binary matrices that an attacker can freely choose. It is computationally infeasible to exhaust all the values of these matrices, and below we show how we efficiently search them to achieve a decent bias.

Since we have ten byte expressions each of which can have up to 48 S-box approximations, it is possible to find some linear combinations of these ten bytes such that some S-box approximations could be removed in N , i.e., the coefficients of the linear part and the S-box part of some bytes both become zero. Now we are interested in the maximum number of S-box approximations that can be removed, as it can give a higher bias.

We first use MILP (Mixed-Integer Linear Programming) to help find a lower bound on the number of active S-boxes, as done in [ENP19]. By solving the MILP problem, we get a first insight that there will be not less than 37 active S-boxes. We next show how we explore linear masking coefficients to remove as many S-box approximations as possible.

5.2 Exploring maskings to remove S-box approximations

We can construct a t -bit noise N_t using the ten 8-bit E -expressions, which is expressed in a matrix form as below:

$$N_t = \begin{pmatrix} c_0 & c_1 & \dots & c_9 \end{pmatrix}_{t \times 10 \cdot 8} \cdot \begin{pmatrix} E_0 \\ E_1 \\ \vdots \\ E_9 \end{pmatrix}_{10 \cdot 8} = \mathbf{c} \cdot \mathbf{E},$$

where c_i 's, $0 \leq i \leq 9$, are $t \times 8$ binary matrices that the attacker can choose freely, but with the constraint that the rank of \mathbf{c} is t , i.e., all rows are nonzero and linearly independent. For simplicity purposes, let us introduce 96 8-bit variables as follows:

$$\begin{aligned} \text{for } i = 0, \dots, 15 : \quad X_i &= R1_i, & Y_i &= S(R1_i), \\ X_{16+i} &= R2_i, & Y_{16+i} &= S^{-1}(R2_i), \\ X_{32+i} &= R3_i, & Y_{32+i} &= S^{-1}(R3_i). \end{aligned}$$

Note that every X_j ($0 \leq j \leq 47$) can be regarded as a uniformly distributed random variable, and Y_j is the corresponding value after the application of the S-box or inverse S-box. Thus, an expression of the form $a \cdot X_j \oplus b \cdot Y_j$, where a, b are two linear maskings, can be possibly biased only when $a \neq 0, b \neq 0$. When $a = 0, b \neq 0$ or $a \neq 0, b = 0$, the expression will be uniform; and when $a = 0, b = 0$, this approximation can be removed. Since every E_i is a linear expression of the X, Y variables, the expression of the noise N_t can be rewritten as:

$$\begin{aligned} N_t &= \begin{pmatrix} c_0 & c_1 & \dots & c_9 \end{pmatrix}_{t \times 10 \cdot 8} \cdot \left[\mathbf{A}_{10 \cdot 8 \times 48 \cdot 8} \cdot \begin{pmatrix} X_0 \\ \vdots \\ X_{47} \end{pmatrix}_{48 \cdot 8} \oplus \mathbf{B}_{10 \cdot 8 \times 48 \cdot 8} \cdot \begin{pmatrix} Y_0 \\ \vdots \\ Y_{47} \end{pmatrix}_{48 \cdot 8} \right] \\ &= \mathbf{c} \cdot [\mathbf{A} \cdot \mathbf{X} \oplus \mathbf{B} \cdot \mathbf{Y}], \end{aligned}$$

where \mathbf{A} and \mathbf{B} are two $10 \cdot 8 \times 48 \cdot 8$ binary matrices derived from the ten E -expressions in Equation 15. It is therefore clear that the total t -bit noise N_t consists of at most 48 sub-noise parts:

$$N_t = \bigoplus_{i=0}^{47} \underbrace{(\mathbf{c} \cdot \mathbf{A})_{[0:t-1; 8i:8i+7]}}_{a_i} \cdot X_i \oplus \underbrace{(\mathbf{c} \cdot \mathbf{B})_{[0:t-1; 8i:8i+7]}}_{b_i} \cdot Y_i,$$

where a_i and b_i are $t \times 8$ binary sub-matrices, constituted from the t rows and the eight columns from $8i$ to $8i + 7$ of the matrices $\mathbf{c} \cdot \mathbf{A}$ and $\mathbf{c} \cdot \mathbf{B}$, respectively. There are in total 96 such matrices.

Obviously, if $a_i = b_i = 0$, the i -th sub-noise part vanishes to zero, and thus the total noise will have a larger bias. If, on the other hand, only one of the two matrices is zero, the contribution of that i -th sub-noise will make some or all bits of N_t pure random, thus these bits will have no contribution to the bias. If all bits are affected and become random, the total bias will be 0. Therefore, we are interested in selecting the masking matrix \mathbf{c} such that we can cancel as many S-box approximations out of 48 as possible, meanwhile guaranteeing that the xor-sum of the remaining sub-noises is biased. Next we show how we achieve this.

Algorithm to derive the linear masking matrix \mathbf{c} . Let us select k distinct indices $\{i_1, i_2, \dots, i_k\} \in \{0, 1, \dots, 47\}$, and we want to cancel the sub-noise parts corresponding

to these k indices, i.e., to make $a_{i_j} = b_{i_j} = 0$ for $j = 1, 2, \dots, k$, by carefully choosing the linear masking c_i 's. We can construct a matrix \mathbf{K} that consists of the corresponding 8-bit columns taken from the matrices \mathbf{A} and \mathbf{B} :

$$\mathbf{K}_{10 \cdot 8 \times 2k \cdot 8} = \begin{pmatrix} \mathbf{A}_{[0:7; 8i_1:8i_1+7]} & \mathbf{B}_{[0:7; 8i_1:8i_1+7]} & \cdots & \mathbf{A}_{[0:7; 8i_k:8i_k+7]} & \mathbf{B}_{[0:7; 8i_k:8i_k+7]} \\ \mathbf{A}_{[8:15; 8i_1:8i_1+7]} & \mathbf{B}_{[8:15; 8i_1:8i_1+7]} & \cdots & \mathbf{A}_{[8:15; 8i_k:8i_k+7]} & \mathbf{B}_{[8:15; 8i_k:8i_k+7]} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \mathbf{A}_{[72:79; 8i_1:8i_1+7]} & \mathbf{B}_{[72:79; 8i_1:8i_1+7]} & \cdots & \mathbf{A}_{[72:79; 8i_k:8i_k+7]} & \mathbf{B}_{[72:79; 8i_k:8i_k+7]} \end{pmatrix},$$

and we want to find a nonzero matrix \mathbf{c} such that:

$$\mathbf{c}_{t \times 80} \cdot \mathbf{K}_{80 \times 2k \cdot 8} = \mathbf{0}_{t \times 2k \cdot 8}.$$

First of all, if the rank r of the matrix \mathbf{K} is 80, there are no valid solutions of \mathbf{c} satisfying $\mathbf{c} \cdot \mathbf{K} = \mathbf{0}$. While if $r < 80$, there exist $t = 80 - r$ nonzero linear combinations that will map through \mathbf{K} to zero. This also explains how the size t for the total noise N_t is derived in our attack.

In order to search for the kernel linear combinations, we initially set \mathbf{c} as a square identity matrix $\mathbf{c}_{80 \times 80} = \mathbf{I}_{80 \times 80}$, then perform the standard Gaussian elimination on the binary matrix \mathbf{K} to transform it to the row echelon form \mathbf{K}' , and apply the same operations to the matrix $\mathbf{c}_{80 \times 80}$. This is quite similar to the steps of deriving an inverse matrix of \mathbf{K} , if \mathbf{K} would be a square matrix.

In the end, we get the row echelon form $\mathbf{K}' = \mathbf{c} \cdot \mathbf{K}$, where the last $t = 80 - r$ rows of \mathbf{K}' are zeroes, while the matrix \mathbf{c} will be of the full-rank 80. Then we keep the last t rows of \mathbf{c} and discard all other r rows, thus deriving the desired $\mathbf{c}_{t \times 80}$ satisfying $\mathbf{c} \cdot \mathbf{K} = \mathbf{0}$.

Search strategy for a good linear approximation. It is now clear that a larger bias of the total noise can be achieved by removing as many S-box approximations (out of 48) as possible. We can do it by exhaustively selecting k indices in $\binom{48}{k}$ ways, then applying the algorithm above to check if a solution for the matrix \mathbf{c} exists for the selected sub-noises, and if so, derive t and the corresponding linear masking matrix \mathbf{c} . Then given the derived $\mathbf{c}_{t \times 80}$, we construct the distribution of the total t -bit noise N_t and compute the bias. We pick the solution for which the total bias is the largest.

Correction approach. For many k -tuples of indices we would get a full-rank \mathbf{K} , and thus we do not have to continue further computations. However, another step of cutting out k -tuples is to do a *correction* approach for the matrix \mathbf{c} . If t is shrunk down to 0 during such a correction, there is no need to continue further computations and we jump to the next k -tuple. The *correction* idea is as follows.

Given a derived masking matrix $\mathbf{c}_{t \times 80}$, we can meet the situation when some of the 48 sub-noises will have $a_i = 0$ and $b_i \neq 0$ (or vice versa), which means that some bits of the t -bit total noise become uniformly distributed. In such a case, we can try to *correct* the masking matrix $\mathbf{c}_{t \times 80}$ by removing those rows where the rows of b_i are nonzero. In this way we shrink t down but get $a_i = b_i = 0$. If t becomes 0 at the end of this procedure, we proceed to the next k -tuple.

If for all 48 sub-noises we get either $a_i = 0, b_i = 0$ or $a_i \neq 0, b_i \neq 0$, the resulting linear masking matrix \mathbf{c} may lead to a biased total noise. We then construct the distribution of the total noise N_t and compute the corresponding bias. When constructing the distribution, we can utilise the Walsh-Hadamard Transforms to speed up the convolution of the 48 t -bit sub-noises [MJ05, YJM19].

Results. In our simulations we managed to find a 16-bit approximation N_{16} , i.e., $t = 16$,

and the masking matrix $\mathbf{c}_{16 \times 80}$ can effectively eliminate nine S-box approximations. The received bias (SEI) is

$$\epsilon(N_{16}) \approx 2^{-303}.$$

The linear masking $\mathbf{c}_{16 \times 80}$ is given in Listing 4, where the bits are encoded as 64-bit unsigned integers in C/C++, and are mapped to the bits of \mathbf{c} as follows:

$$\mathbf{c}_{16 \times 80}[i, j] = (C[i][j/64] \gg (j\%64)) \& 1.$$

```
uint64_t C[16][2] = {
{ 0x0000020200020000ULL, 0x0000ULL}, { 0x94730000005e0000ULL, 0x0000ULL},
{ 0x0000080800080000ULL, 0x0000ULL}, { 0x48c4159600fa0120ULL, 0x0002ULL},
{ 0x48c421a200ce0120ULL, 0x0002ULL}, { 0x0000444400440000ULL, 0x0000ULL},
{ 0x3c15810000220080ULL, 0x0001ULL}, { 0x0000000000000022ULL, 0x0000ULL},
{ 0x40c1000006000000ULL, 0x0100ULL}, { 0x0000000000000008ULL, 0x0000ULL},
{ 0x0000000000000060ULL, 0x0000ULL}, { 0x0000000000000021ULL, 0x0000ULL},
{ 0x0000000000000040ULL, 0x0000ULL}, { 0x0000000000000010ULL, 0x0000ULL},
{ 0x4b39000000ee0000ULL, 0x0000ULL}, { 0x54cc000000fe0000ULL, 0x8000ULL}};
```

Listing 4: The linear masking $\mathbf{c}_{16 \times 80}$.

We also tested if there exists a linear masking that can eliminate ten or more S-box approximations. We ran our exhaustive search program with $k = 10$ for all the $\binom{48}{10} \approx 2^{32.6}$ 10-tuples, but with no valid results returned. By this we confirm that at most nine S-box approximations can be removed from the total noise expression.

5.3 Distinguishing attack against SNOW-V

If all arithmetical additions are substituted with exclusive-OR, we could have a distinguishing attack against this variant with data complexity 2^{303} . Specifically, one should collect around 2^{303} triples of consecutive keystream words, and construct Y by applying l_β, h_β operations on $z^{(t-1)}, z^{(t)}$, respectively, for each t ; then build the E -bytes from Y and finally apply the linear masking $\mathbf{c}_{16 \times 80}$ in Listing 4 to the vector of E -bytes, thus one keystream sample of the form N_t is derived. One needs to collect 2^{303} such samples and the derived sequence can be distinguished from random.

We emphasize that the bias derived in our attack does not depend on the key or IV. Moreover, unlike the typical linear attacks against LFSR-based stream ciphers in which the keystream samples can only be collected through one very long keystream sequence corresponding to one (K, IV) pair, the data in our attack can be collected from many short keystream sequences under different (K, IV) pairs. Though the data complexity is still out of reach in practice, the attacking scenario is more relevant to the practical situation. For example, in the mobile communication system, one (K, IV) pair only allows to maximally generate 2^{32} keystream words (SNOW-V design limits it to 2^{64}), and many encryption sessions are short signalling messages. Classical linear attacks subject to these constraints, while our attack can still collect samples from these short keystream sequences. The attack can also be used to recover some unknown bits of a plaintext encrypted a large number of times with different IVs and potentially different keys, e.g., in a broadcast setting [SSS⁺19].

Our attack is the first linear attack that can exploit short keystream sequences among the SNOW family of stream ciphers. However, the attack complexity is still beyond the exhaustive key search.

Discussion on the full version. If we take the 32-bit adders into consideration, the bias would change. However, how the bias would vary is not clear, as the \boxplus_{32} operations can be seen as part of multiple S-boxes and their approximations. On the other hand, it is computationally difficult to compute the bias by exhaustive looping. We do not have a

good idea about how to compute that bias in practice, and leave it as an open question for further research.

6 Conclusions

In this paper, we investigate the security of SNOW-V and propose two guess-and-determine attacks with complexities 2^{384} and 2^{378} , respectively, and one distinguishing attack against a reduced version of SNOW-V, in which the 32-bit adders are replaced with exclusive-OR, with complexity 2^{303} . These attacks do not threaten the full SNOW-V, but provide a deeper understanding into its security. Besides, our attacks provide new ideas for cryptanalysis against other ciphers. Specifically, we recommend that in a guess-and-determine attack, instead of simple looping, one should carefully design the order of the guessing and always truncate those paths invalidating some equation constraints. In this way, one can save the cost for going through the invalid guessing paths and thus the complexity is reduced. For linear cryptanalysis against LFSR-based stream ciphers, it is valuable to check if the LFSR contribution can be cancelled locally without need to find a multiple of the generating polynomial and combine keystream words very far away. In this way, the required keystream samples can be collected under different key and IV pairs, exempt from the restrictions on length limits of keystream sequences in practice.

References

- [3GP19] 3GPP. TS 33.841 (V16.1.0): 3rd generation partnership project; technical specification group services and systems aspects; security aspects; study on the support of 256-bit algorithms for 5G (release 16). March 2019. <https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=3422>.
- [BJV04] Thomas Baigneres, Pascal Junod, and Serge Vaudenay. How far can we go beyond linear cryptanalysis? In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 432–450. Springer, 2004.
- [CDM20] Carlos Cid, Matthew Dodd, and Sean Murphy. A Security Evaluation of the SNOW-V Stream Cipher. 4 June 2020. Quaternion Security Ltd. https://www.3gpp.org/ftp/tsg_sa/WG3_Security/TSGS3_101e/Docs/S3-202852.zip.
- [EJMY19] Patrik Ekdhahl, Thomas Johansson, Alexander Maximov, and Jing Yang. A new SNOW stream cipher called SNOW-V. *IACR Transactions on Symmetric Cryptology*, 2019(3):1–42, Sep. 2019. <https://tosc.iacr.org/index.php/ToSC/article/view/8356>.
- [ENP19] Maria Eichlseder, Marcel Nageler, and Robert Primas. Analyzing the linear keystream biases in AEGIS. *IACR Transactions on Symmetric Cryptology*, pages 348–368, 2019.
- [GZ21] Xinxin Gong and Bin Zhang. Resistance of SNOW-V against fast correlation attacks. *IACR Transactions on Symmetric Cryptology*, (1):378–410, March 2021. <https://tosc.iacr.org/index.php/ToSC/article/view/8843>.
- [HII⁺21] Jin Hoki, Takanori Isobe, Ryoma Ito, Fukang Liu, and Kosei Sakamoto. Distinguishing and key recovery attacks on the reduced-round SNOW-V and SNOW-Vi. *Cryptology ePrint Archive*, Report 2021/546, 2021. <https://eprint.iacr.org/2021/546>.

- [JLH20] Lin Jiao, Yongqiang Li, and Yonglin Hao. A guess-and-determine attack on SNOW-V stream cipher. *The Computer Journal*, 63(12):1789–1812, 03 2020. <https://doi.org/10.1093/comjnl/bxaa003>.
- [MJ05] Alexander Maximov and Thomas Johansson. Fast computation of large distributions and its cryptographic applications. In Bimal Roy, editor, *Advances in Cryptology - ASIACRYPT 2005*, pages 313–332, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
- [SAG20] ETSI SAGE. 256-bit algorithms based on SNOW 3G or SNOW V. 4 November 2020. LS S3-203338. https://www.3gpp.org/ftp/tsg_sa/WG3_Security/TSGS3_101e/Docs/S3-203338.zip.
- [SSS⁺19] Danping Shi, Siwei Sun, Yu Sasaki, Chaoyun Li, and Lei Hu. Correlation of quadratic boolean functions: Cryptanalysis of all versions of full MORUS. In *Annual International Cryptology Conference*, pages 180–209. Springer, 2019.
- [YJ20] Jing Yang and Thomas Johansson. An overview of cryptographic primitives for possible use in 5G and beyond. *Science China Information Sciences*, 63(12):1–22, 2020.
- [YJM19] Jing Yang, Thomas Johansson, and Alexander Maximov. Vectorized linear approximations for attacks on SNOW 3G. *IACR Transactions on Symmetric Cryptology*, pages 249–271, 2019.

A The matrices

0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 1	1 1 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 1	0 0 1 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0	0 0 0 1 0 0 0 0 0	0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0	0 0 0 0 1 0 0 0 0	0 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0	0 0 0 0 0 0 0 0 0	1 0 0 0 0 1 0 0 0	0 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0	0 0 0 0 0 0 0 0 1	1 0 0 0 0 0 1 0 0	0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0	0 0 0 0 0 0 0 0 1	0 0 0 0 0 0 0 1 0	0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0	0 0 0 0 0 0 0 0 1	0 0 0 0 0 0 0 0 1	0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0	0 0 0 0 0 0 0 0 0	1 0 0 0 0 0 0 0 0	1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1	0 0 0 0 0 0 0 0 1	0 0 0 0 0 0 0 0 0	0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0	1 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0	0 0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0	0 1 0 0 0 0 0 0 0	1 0 0 0 0 0 0 0 0	0 0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 0 0	0 0 1 0 0 0 0 0 1	0 0 0 0 0 0 0 0 0	0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0 0	0 0 0 1 0 0 0 0 0	0 0 0 0 0 0 0 0 0	0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 0	0 0 0 0 1 0 0 0 0	1 0 0 0 0 0 0 0 0	0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0	0 0 0 0 0 1 0 0 1	1 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 1 0 1	1 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 1 1	1 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0

Listing 5: The 16×16 binary matrices for β (left) and β^{-1} (right).

e b d 9	0 0 0 0	0 0 0 0	0 0 0 0	2 0 0 0	0 3 0 0	0 0 1 0	0 0 0 1
0 0 0 0	0 0 0 0	0 0 0 0	9 e b d	1 0 0 0	0 2 0 0	0 0 3 0	0 0 0 1
0 0 0 0	0 0 0 0	d 9 e b	0 0 0 0	1 0 0 0	0 1 0 0	0 0 2 0	0 0 0 3
0 0 0 0	b d 9 e	0 0 0 0	0 0 0 0	3 0 0 0	0 1 0 0	0 0 1 0	0 0 0 2
0 0 0 0	e b d 9	0 0 0 0	0 0 0 0	0 0 0 1	2 0 0 0	0 3 0 0	0 0 1 0
9 e b d	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 1	1 0 0 0	0 2 0 0	0 0 3 0
0 0 0 0	0 0 0 0	0 0 0 0	d 9 e b	0 0 0 3	1 0 0 0	0 1 0 0	0 0 2 0
0 0 0 0	0 0 0 0	b d 9 e	0 0 0 0	0 0 0 2	3 0 0 0	0 1 0 0	0 0 1 0
0 0 0 0	0 0 0 0	e b d 9	0 0 0 0	0 0 1 0	0 0 0 1	2 0 0 0	0 3 0 0
0 0 0 0	9 e b d	0 0 0 0	0 0 0 0	0 0 3 0	0 0 0 1	1 0 0 0	0 2 0 0
d 9 e b	0 0 0 0	0 0 0 0	0 0 0 0	0 0 2 0	0 0 0 3	1 0 0 0	0 1 0 0
0 0 0 0	0 0 0 0	0 0 0 0	b d 9 e	0 0 1 0	0 0 0 2	3 0 0 0	0 1 0 0
0 0 0 0	0 0 0 0	0 0 0 0	e b d 9	0 3 0 0	0 0 1 0	0 0 0 1	2 0 0 0
0 0 0 0	0 0 0 0	9 e b d	0 0 0 0	0 2 0 0	0 0 3 0	0 0 0 1	1 0 0 0
0 0 0 0	d 9 e b	0 0 0 0	0 0 0 0	0 1 0 0	0 0 2 0	0 0 0 3	1 0 0 0

```
b d 9 e | 0 0 0 0 | 0 0 0 0 | 0 0 0 0 | 0 1 0 0 | 0 0 1 0 | 0 0 0 2 | 3 0 0 0
```

Listing 6: The L^{-1} (left) and L (right) matrices.

B The operations of l_β and h_β in bytes

$l_\beta(B0)$ can be expressed in bytes as below:

$$\begin{aligned} l_\beta(B0)_{0||1} &= \beta B0_{0||1} \oplus B0_{6||7}, & l_\beta(B0)_{2||3} &= \beta B0_{2||3} \oplus B0_{8||9}, \\ l_\beta(B0)_{4||5} &= \beta B0_{4||5} \oplus B0_{10||11}, & l_\beta(B0)_{6||7} &= \beta B0_{6||7} \oplus B0_{12||13}, \\ l_\beta(B0)_{8||9} &= \beta B0_{8||9} \oplus B0_{14||15}, & l_\beta(B0)_{10||11} &= \beta B0_{10||11}, \\ l_\beta(B0)_{12||13} &= \beta B0_{12||13}, & l_\beta(B0)_{14||15} &= \beta B0_{14||15}. \end{aligned}$$

$h_\beta(B1)$ can be expressed in bytes as below:

$$\begin{aligned} h_\beta(B1)_{0||1} &= \beta^{-1} B1_{0||1}, & h_\beta(B1)_{2||3} &= \beta^{-1} B1_{2||3}, \\ h_\beta(B1)_{4||5} &= \beta^{-1} B1_{4||5}, & h_\beta(B1)_{6||7} &= \beta^{-1} B1_{6||7}, \\ h_\beta(B1)_{8||9} &= \beta^{-1} B1_{8||9}, & h_\beta(B1)_{10||11} &= \beta^{-1} B1_{10||11} \oplus B1_{0||1}, \\ h_\beta(B1)_{12||13} &= \beta^{-1} B1_{12||13} \oplus B1_{2||3}, & h_\beta(B1)_{14||15} &= \beta^{-1} B1_{14||15} \oplus B1_{4||5}. \end{aligned}$$

C Recursion implementation for the 10-steps algorithm

Note that for a random choice of inputs $C, D, R2, R3$, the probability that there is at least one solution of $A0$ is $2^{-36.84}$. However, if solutions exist, the average number of solutions will be $2^{36.84}$. Therefore, in the code below we also include the flag `solvable=0/1` as the argument to the method `Dequation::random()` that generates either a fully random input where $A0$ may possibly have a solution, or a random input where $A0$ is guaranteed to have a solution – that is for testing and simulation purposes.

```
struct Dequation
{
    u8 R2[16], R3[16], C[16], D[16]; // input
    u8 u[16], v[16]; // internal
    u8 A0[16]; // result

    void computeD(u8 * Dr)
    {
        u8 T1[16];
        for (int i = 0; i < 4; i++)
            ((u32*)T1)[i] = ((u32*)R2)[i] + (((u32*)R3)[i] ^ ((u32*)A0)[i]);
        for (int i = 0; i < 16; i++)
            Dr[i] = T1[((i >> 2) | (i << 2)) & 0xf];
        for (int i = 0; i < 4; i++)
            ((u32*)Dr)[i] += ((u32*)A0)[i] ^ ((u32*)C)[i];
    }

    void random(int solvable=0)
    {
        memset(this, 0xff, sizeof(*this));
        for (int i = 0; i < 16; i++)
        {
            R2[i] = rand();
            R3[i] = rand();
            C[i] = rand();
            A0[i] = rand();
            D[i] = rand();
        }
        if(solvable) computeD(D);
    }
}
```

```

int expr(int i, int j, int Xi, int Xj)
{ return D[i] ^ ((R2[j] + (R3[j] ^ Xj) + u[j]) + (Xi ^ C[i]) + v[i]);
}

void solve1(int step, int i, int X=0, int bit=-1)
{
    if (bit >= 0 && (expr(i, i, X, X) & (1 << bit))) return;
    if (bit == 7)
    {
        AO[i] = X;
        next_carries(i, i);
        solve(step + 1);
        return;
    }
    solve1(step, i, X, ++bit);
    solve1(step, i, X ^ (1 << bit), bit);
}

void solve2(int step, int i, int j, int Xi = 0, int Xj=0, int bit=-1)
{
    if (bit >= 0 && ((expr(i, j, Xi, Xj) | expr(j, i, Xj, Xi)) & (1 << bit)))
        return;

    if (bit == 7)
    {
        AO[i] = Xi;
        AO[j] = Xj;
        next_carries(i, j);
        next_carries(j, i);
        solve(step + 1);
        return;
    }
    int t = (1 << ++bit);
    solve2(step, i, j, Xi, Xj, bit);
    solve2(step, i, j, Xi ^ t, Xj, bit);
    solve2(step, i, j, Xi, Xj ^ t, bit);
    solve2(step, i, j, Xi ^ t, Xj ^ t, bit);
}

void next_carries(int i, int j)
{
    int nu = ((int)R2[j] + (int)(R3[j] ^ AO[j]) + (int)u[j]);
    int nv = (nu & 0xff) + (int)(AO[i] ^ C[i]) + (int)v[i];
    ++i, ++j;
    if (j & 3) u[j] = nu >> 8;
    if (i & 3) v[i] = nv >> 8;
}

void solve(int step = 0)
{
    static int S[10] = { 0, 1, 2, 5, 3, 6, 10, 7, 11, 15 };
    if (step == 0)
        u[0] = u[4] = u[8] = u[12] = v[0] = v[4] = v[8] = v[12] = 0;

    if (step == 10)
    {
        // A solution for A0 is found! do something with it...
        u8 ver[16]; // we just verify that the solution is correct
        computed(ver);
        if (memcmp(D, ver, 16))
            printf("ERROR: Verification of the derived A0 failed!\n");
        return;
    }
}

```


120						5	273
124						1	41
128						56	4688
132							5
136							5
140							5
144							82
148							1
152							5
156							5
160							56
164							1
168							33
172							1
176							18
180							5
184							5
188							1
192							160
196							3
200							5
204							1
208							18
212							1
216							5
220							1
224							56
228							1
232							5
236							1
240							18
244							1
248							5
252							1
256							128

Table 4: Distribution table for *Type-2* equations.

E The flowcharts of the guess-and-determine attacks

F The probability of valid solutions of Equation 13

In this section, we compute the probability of valid solutions in Equation 13. We recall that the equations are:

$$z^{(t)} \oplus R2 = R1 \boxplus_{32} B1,$$

$$Y \oplus h_{\beta}(B0) = (\sigma(R1) \boxminus_{32} \text{AES}_R^{-1}(R3)) \oplus B1,$$

where $R1$ and $B1$ are the two unknowns. First we note that $z^{(t)}$ and $R2$ are independent from the rest variables, looping over the xor-sum of $z^{(t)}$ and $R2$ is equivalent to looping over one random variable. Thus, we use a new variable U to denote $z^{(t)} \oplus R2$. Similarly, Y and $B0$ are independent from the rest variables, and we can regard $Y \oplus h_{\beta}(B0)$ as a

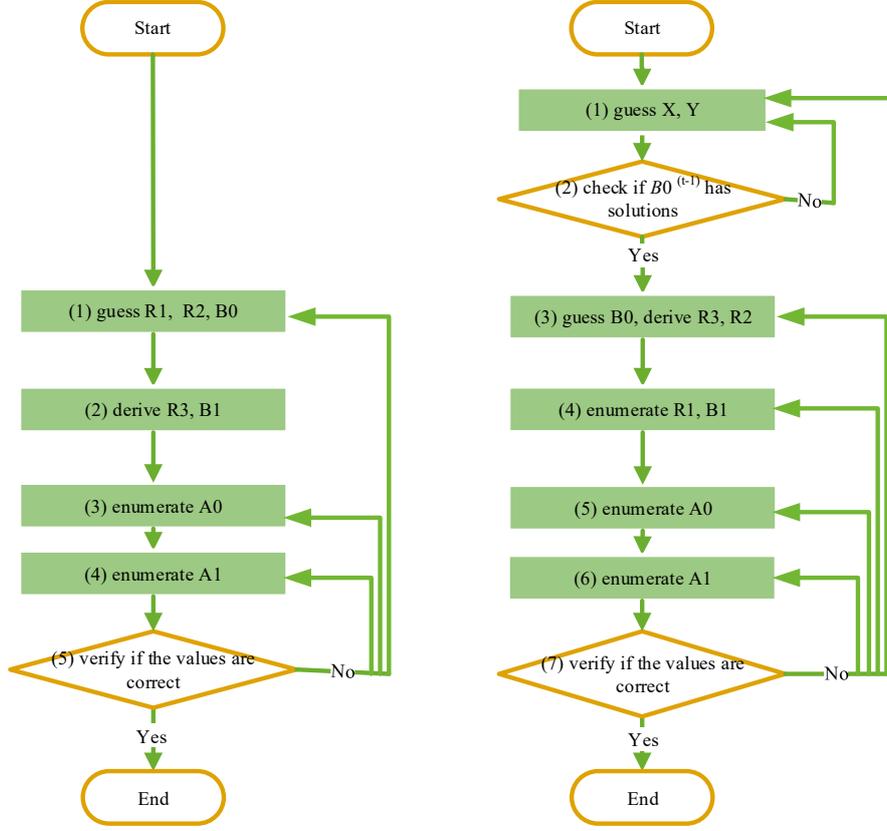


Figure 3: Illustration of the GnP attacks (left: first; right: second).

new variable V . Here we should be careful about $h_\beta(B0)$: since h_β is a full-rank matrix, when $B0$ takes all the values, $h_\beta(B0)$ will also take all the values. $\text{AES}_R^{-1}(R3)$ can also be regarded as a random variable W as it is a bijective mapping.

Thus we have a simplified system of equations:

$$\begin{aligned} U &= R1 \boxplus_{32} B1, \\ V &= (\sigma(R1) \boxminus_{32} W) \oplus B1. \end{aligned} \quad (16)$$

According to Equation 16, we have $B1 = (\sigma(R1) \boxminus_{32} W) \oplus V$, and further get:

$$U = R1 \boxplus_{32} ((\sigma(R1) \boxminus_{32} W) \oplus V). \quad (17)$$

The distributions of number of solutions of Equation 16 and Equation 17 are the same, since $B1$ is uniquely determined given V, W and $R1$. We have experimentally verified this observation over smaller dimensions. Thus we can use Equation 17 to get the distribution of number of solutions of $R1$ and $B1$.

Similarly, we would have two types of equations, the first type with the form below,

$$U_0 = R1_0 \boxplus_8 ((R1_0 \boxminus_8 W_0 \boxminus_8 v_0) \oplus V_0) \boxplus_8 u_0,$$

and the second type with the form below:

$$\begin{aligned} U_1 &= R1_1 \boxplus_8 ((R1_4 \boxminus_8 W_1 \boxminus_8 v_1) \oplus V_1) \boxplus_8 u_1 \\ U_4 &= R1_4 \boxplus_8 ((R1_1 \boxminus_8 W_4 \boxminus_8 v_4) \oplus V_4) \boxplus_8 u_4. \end{aligned}$$

We have experimentally computed the distributions of solutions for these two types of equations, and the probabilities of having solutions are $2^{-3.91}$ and $2^{-3.53}$, respectively. The average number of solutions is exactly one for each combination of other variables. The results are just the same to the ones of the D-equations for $A0$ in the first GnD attack.

G The probability p_z

In this section, we derive the probability p_z of $B0^{(t-1)}$ having solutions in the equation of $z^{(t-2)}$.

Recall that the equation of $z^{(t-2)}$ is expressed as below:

$$z^{(t-2)} = (B0^{(t-1)} \boxplus_{32} X) \oplus \text{AES}_R^{-1}(l_\beta(B0^{(t-1)}) \oplus Y),$$

where $\text{AES}_R^{-1}(X)$ can be expressed as $S^{-1}(L^{-1} \cdot X)$, and l_β operation is defined in Equation 5. We temporarily replace \boxplus_{32} with \boxplus_8 . For simplicity, we denote $Y' = L^{-1}Y$ and ignore the time notations, then we can simplify the equation as:

$$z = (B0 \boxplus_8 X) \oplus S^{-1}(L^{-1}l_\beta(B0) \oplus Y').$$

Now our task is to compute the probability of $B0$ having solutions given z, X, Y' . We use an enumeration algorithm to achieve this by considering four groups of equations recursively, which are given below.

Step 1. Before giving the first group of equations, we first use z_{12} as an example to illustrate how to derive each byte of z in details. z_{12} can be expressed as:

$$\begin{aligned} z_{12} &= (B0_{12} \boxplus_8 X_{12}) \\ &\oplus S^{-1}((e, b, d, 9) \cdot (\beta(B0_{12||13})_0, \beta(B0_{12||13})_1, \beta(B0_{14||15})_0, \beta(B0_{14||15})_1) \oplus Y'_{12}), \end{aligned}$$

where $\beta(B0_{i||i+1})_j, i \in \{12, 14\}, j \in \{0, 1\}$ is the j -th byte of $\beta(B0_i||B0_{i+1})$.

For simplicity of expressions, we use $[B0_{i,i+1,i+2,i+3}]$ to denote the vector of the four bytes $(B0_i, B0_{i+1}, B0_{i+2}, B0_{i+3})$ and $[\psi B0_{i,i+1,i+2,i+3}]$ to denote the vector of the four bytes after multiplying with β , i.e., $(\beta(B0_{i||i+1})_0, \beta(B0_{i||i+1})_1, \beta(B0_{i+2||i+3})_0, \beta(B0_{i+2||i+3})_1)$, for $i = 0, 4, 8, 12$.

Now consider the first group of equations:

$$\begin{aligned} z_{12} &= (B0_{12} \boxplus_8 X_{12}) \oplus S^{-1}((e, b, d, 9) \cdot [\psi B0_{12,13,14,15}] \oplus Y'_{12}), \\ z_{11} &= (B0_{11} \boxplus_8 X_{11}) \oplus S^{-1}((b, d, 9, e) \cdot [\psi B0_{12,13,14,15}] \oplus Y'_{11}), \\ z_6 &= (B0_6 \boxplus_8 X_6) \oplus S^{-1}((d, 9, e, b) \cdot [\psi B0_{12,13,14,15}] \oplus Y'_6), \\ z_1 &= (B0_1 \boxplus_8 X_1) \oplus S^{-1}((9, e, b, d) \cdot [\psi B0_{12,13,14,15}] \oplus Y'_1). \end{aligned}$$

Given the bytes of z, X, Y' , we can freely choose the values of $B0_{13,14,15}$, then in z_{12} only $B0_{12}$ remains unknown. Once $B0_{12}$ is further determined, $B0_{1,6,11}$ will be derived uniquely from $z_{1,6,11}$, thus there is always a solution for these bytes if $B0_{12}$ in z_{12} has solutions. So the main task now is to compute the probability of $B0_{12}$ having solutions in z_{12} . According to the expression of β matrix given in Appendix A, z_{12} can be further derived as:

$$z_{12} = (B0_{12} \boxplus_8 X_{12}) \oplus S^{-1}(e \cdot (B0_{12} \ll 1) \oplus b \cdot (B0_{12} \gg 7) \oplus Y''_{12}),$$

where Y''_{12} is a new variable, which is the linear combination of $Y'_{12}, B0_{13}, B0_{14}, B0_{15}$. We can compute the probability of $B0_{12}$ having at least one solution, denoted $p_z(B0_{12})$, which is:

$$p_z(B0_{12}) \approx 0.363230705.$$

Thus, in Step 1 we can loop over $B_{0_{13,14,15}}$, solve $B_{0_{12}}$ with valid solutions of probability $p_z(B_{0_{12}})$, and further derive $B_{0_{1,6,11}}$ correspondingly.

Step 2. Consider the second group of equations:

$$\begin{aligned} z_{13} &= (B_{0_{13}} \boxplus_8 X_{13}) \oplus S^{-1}((9, e, b, d) \cdot ([\psi B_{0_{8,9,10,11}}] \oplus (B_{0_{14}}, B_{0_{15}}, 0, 0)) \oplus Y'_{13}), \\ z_8 &= (B_{0_8} \boxplus_8 X_8) \oplus S^{-1}((e, b, d, 9) \cdot ([\psi B_{0_{8,9,10,11}}] \oplus (B_{0_{14}}, B_{0_{15}}, 0, 0)) \oplus Y'_8), \\ z_7 &= (B_{0_7} \boxplus_8 X_7) \oplus S^{-1}((b, d, 9, e) \cdot ([\psi B_{0_{8,9,10,11}}] \oplus (B_{0_{14}}, B_{0_{15}}, 0, 0)) \oplus Y'_7), \\ z_2 &= (B_{0_2} \boxplus_8 X_2) \oplus S^{-1}((d, 9, e, b) \cdot ([\psi B_{0_{8,9,10,11}}] \oplus (B_{0_{14}}, B_{0_{15}}, 0, 0)) \oplus Y'_2). \end{aligned}$$

Here we can only freely choose $B_{0_{9,10}}$, as the values of $B_{0_{11,14,15}}$ have already been considered in Step 1. We add the linear combinations of these known variables to the Y' -terms, resulting in new Y'' variables, and use a new variable X'_{13} to denote $B_{0_{13}} \boxplus_8 X_{13}$, which is also known. Thus we need to find solutions of B_{0_8} that satisfies the two equations below:

$$\begin{aligned} z_{13} &= X'_{13} \oplus S^{-1}(9 \cdot (B_{0_8} \ll 1) \oplus e \cdot (B_{0_8} \gg 7) \oplus Y''_{13}), \\ z_8 &= (B_{0_8} \boxplus_8 X_8) \oplus S^{-1}(e \cdot (B_{0_8} \ll 1) \oplus b \cdot (B_{0_8} \gg 7) \oplus Y''_8). \end{aligned}$$

We have computed that the probability of valid solutions for B_{0_8} is:

$$p_z(B_{0_8}) \approx 0.363230705 \cdot 2^{-8}.$$

This can be understood in another way: the probability of B_{0_8} having solutions in z_8 is 0.363230705, and the solutions will satisfy the equation of z_{13} with probability around 2^{-8} . After we have solved B_{0_8} , we can further derive B_{0_7} and B_{0_2} uniquely. Thus in Step 2 we can loop over $B_{0_{9,10}}$, solve B_{0_8} with valid solutions of probability $p_z(B_{0_8})$, and further derive $B_{0_{2,7}}$.

Step 3. We further consider the next group of equations:

$$\begin{aligned} z_{14} &= (B_{0_{14}} \boxplus_8 X_{14}) \oplus S^{-1}((d, 9, e, b) \cdot ([\psi B_{0_{4,5,6,7}}] \oplus [B_{0_{10,11,12,13}}]) \oplus Y'_{14}), \\ z_9 &= (B_{0_9} \boxplus_8 X_9) \oplus S^{-1}((9, e, b, d) \cdot ([\psi B_{0_{4,5,6,7}}] \oplus [B_{0_{10,11,12,13}}]) \oplus Y'_9), \\ z_4 &= (B_{0_4} \boxplus_8 X_4) \oplus S^{-1}((e, b, d, 9) \cdot ([\psi B_{0_{4,5,6,7}}] \oplus [B_{0_{10,11,12,13}}]) \oplus Y'_4), \\ z_3 &= (B_{0_3} \boxplus_8 X_3) \oplus S^{-1}((b, d, 9, e) \cdot ([\psi B_{0_{4,5,6,7}}] \oplus [B_{0_{10,11,12,13}}]) \oplus Y'_3). \end{aligned}$$

The known bytes $B_{0_{6,7,10,11,12,13}}$ are added to the Y' -terms, while the bytes $B_{0_{9,14}}$ are added to the X -terms. We can freely loop over B_{0_5} and solve the following equation in B_{0_4} :

$$z_4 = (B_{0_4} \boxplus_8 X_4) \oplus S^{-1}(e \cdot (B_{0_4} \ll 1) \oplus b \cdot (B_{0_4} \gg 7) \oplus Y''_4).$$

The probability of valid B_{0_4} solutions in z_4 is again computed as 0.363230705, and such solutions will satisfy z_{14}, z_9 with probability around 2^{-16} . Thus the total probability of valid B_{0_4} solutions, denoted $p_z(B_{0_4})$, is computed as:

$$p_z(B_{0_4}) \approx 0.363230705 \cdot 2^{-16}.$$

After B_{0_4} having been solved, B_{0_3} can be uniquely determined according to z_3 . Thus in Step 3 we can loop over B_{0_5} , solve B_{0_4} with valid solutions of probability $p_z(B_{0_4})$, and further derive B_{0_3} .

Step 4. The last group of equations contain the remaining four byte expressions $z_{0,5,10,15}$ in only one unknown variable B_{0_0} , while other variables are already known:

$$\begin{aligned} z_0 &= (B_{0_0} \boxplus_8 X_0) \oplus S^{-1}((e, b, d, 9) \cdot ([\psi B_{0_0,1,2,3}] \oplus [B_{0_6,7,8,9}]) \oplus Y'_0), \\ z_5 &= (B_{0_5} \boxplus_8 X_5) \oplus S^{-1}((9, e, b, d) \cdot ([\psi B_{0_0,1,2,3}] \oplus [B_{0_6,7,8,9}]) \oplus Y'_5), \\ z_{10} &= (B_{0_{10}} \boxplus_8 X_{10}) \oplus S^{-1}((d, 9, e, b) \cdot ([\psi B_{0_0,1,2,3}] \oplus [B_{0_6,7,8,9}]) \oplus Y'_{10}), \\ z_{15} &= (B_{0_{15}} \boxplus_8 X_{15}) \oplus S^{-1}((b, d, 9, e) \cdot ([\psi B_{0_0,1,2,3}] \oplus [B_{0_6,7,8,9}]) \oplus Y'_{15}). \end{aligned}$$

Similarly, B_{0_0} will have valid solutions with probability 0.363230705 in z_0 , and these solutions will satisfy z_5, z_{10}, z_{15} with probability 2^{-24} . Thus the probability of valid B_{0_0} solutions, denoted $p_z(B_{0_0})$, is:

$$p_z(B_{0_0}) \approx 0.363230705 \cdot 2^{-24}.$$

Summary. We can freely choose six bytes of B_0 , i.e., $B_{0_{5,9,10,13,14,15}}$, of total size 2^{48} , which will result into valid solutions for bytes $B_{0_{0,4,8,12}}$ with probability $p_z(B_{0_0}) \cdot p_z(B_{0_4}) \cdot p_z(B_{0_8}) \cdot p_z(B_{12})$. Other bytes will be further uniquely determined. Thus the total probability p_z is computed as:

$$p_z = 2^{48} \cdot p_z(B_{0_0}) \cdot p_z(B_{0_4}) \cdot p_z(B_{0_8}) \cdot p_z(B_{12}) \approx 2^{-5.84}.$$

We cannot really compute an exact success probability for 32-bit adders \boxplus_{32} , but one can expect that it would be very similar to the derived probability, as only several carrier bits need to be further considered.