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

Jing Yang[1], Thomas Johansson[1] and Alexander Maximov[2]

[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 according to four consecutive keystream words. We finally use the next three keystream words to verify the correct guess. 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 operations. The samples 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 [EJMY19] 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 [3GP19]. First, the 256-bit security level is expected in 5G to resist against attackers equipped with quantum computing capability, 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]. 3GPP has asked ETSI SAGE (Security Algorithms Group of Experts) to select and evaluate efficient confidentiality and integrity algorithms for 5G use [3GP19]. 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 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 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 providing 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 [EJMY19] and more than 1 Tbps in hardware [CBB20].

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, more in-depth and focused studies followed, e.g., [JLH20, GZ21, HII$^+$21]. For example, the paper [HII$^+$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 and key recovery attacks are applicable to SNOW-V with reduced initialisation rounds of five, out of the original 16, which indicates that the initialisation has a good security margin. Below we give a more detailed introduction to the guess-and-determine attacks [JLH20, CDM20] and linear cryptanalysis [GZ21] against SNOW-V, and present our contribution.

**Guess-and-determine (GnD) attacks.** A basic GnD attack of complexity $2^{512}$ is proposed in the evaluation report [CDM20]. In this attack, one has to guess three out of the seven internal 128-bit state variables and derive another three using three consecutive keystream words. Although not all derivation details were given, it was assumed that the derivation is possible with a negligible time, leading to recovering six state variables in time complexity $2^{384}$. The last seventh state variable is recovered by guessing, thus the total complexity is $2^{512}$. The next four keystream words are thereafter used to verify the correct guess. The authors in [JLH20] propose a byte-based GnD attack against SNOW-V with complexity $2^{406}$ using seven keystream words. In their attack, the state variables are split into bytes with some carriers introduced, and dynamic programming tool is used to help search a good guessing path that requires guessing as few bytes as possible. Both GnD attacks require seven consecutive 128-bit keystream words which looks reasonable, as the internal state has seven 128-bit unknown variables that needs to be either guessed or determined.

**Our contribution.** Our GnD attacks follow the research line of the GnD attack in [CDM20] and fill the gaps of it. In our first GnD attack, we find an efficient *recursive enumeration* technique in a byte-wise manner to determine three more state variables given three guesses and three consecutive keystream words, such that the complexity of deriving six state variables is still $2^{384}$. We then use the same enumeration way to derive the last seventh state variable with negligible overhead, thus the total attack complexity is $2^{384}$. This improves the GnD attacks both in [CDM20] ($2^{512}$) and [JLH20] ($2^{406}$).

In our recursive enumeration technique, 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. We carefully design the guessing order of the guesses, such that most guessing paths would be truncated at some point without going into the end, resulting in the total GnD attack complexity $2^{384}$.

In our second GnD attack, we use one additional "backward" keystream word as side information to impose more constraints and truncate more "forward" guessing paths, thus further reduce the complexity to $2^{378}$. In order to retrieve the side information efficiently (e.g., instantly) we need a volatile table of size $2^{128}$ bits, which might be implemented in RAM or HDD. The improvement factor over the first GnD attack is not so significant, but the idea of using side information to refute more guessing paths and thus reduce the overall time complexity is interesting in general.

**Linear cryptanalysis.** The SNOW family of stream ciphers is constructed from two components – the LFSR, serving as the source of pseudo-randomness, and the FSM, providing nonlinearity. In typical linear cryptanalysis of such a construction, the nonlinear part FSM is approximated by a linear expression between some keystream words and LFSR variables plus a biased noise variable $N^{(t)}$, while the variables in the FSM are cancelled. In a distinguishing attack, such linear expressions at $k$ time instances corresponding to either the feedback polynomial or a low-weight (usually 3 or 4) multiple of it will be combined (typically through exclusive-OR operation), such that the contribution terms from the LFSR are cancelled. Hence, the linear expressions involve only the keystream symbols and noises, making a reorganised keystream sample sequence biased and distinguishable from random, given enough number of samples. As the FSM approximation expression is repeated $k$ times, the total noise would then involve $k$ sub-noises.

For example, in SNOW 2.0 [EJ02], the LFSR has a feedback polynomial of weight four in the time frame of width 17 over $\mathbb{F}_{2^{32}}$.The authors of [WBDC03, NW06] found a very strong approximation of the FSM such that the bias is large, and combining four such approximations to cancel out the LFSR contribution led to a distinguishing attack of overall complexity $2^{225}$ in [WBDC03] and further improved to $2^{174}$ in [NW06]. Note that in both papers the feedback polynomial is used to find the four time instances such that the LFSR contribution can be cancelled, and the required samples can be collected from many short keystreams under different key and IV (Initialisation Vector) pairs.

A straightforward prevention of above situation is to increase the number of taps in the LFSR update function. In this case the direct usage of the feedback polynomial would involve many more sub-noises, and the bias of the total noise would be very small. However, there is a possibility to find a theoretical low-weigh multiple of any feedback polynomial, due to the birthday paradox, such that one can still construct a biased noise sample from several keystream words but far apart in time instances, i.e., the attacker needs a long keystream sequence to collect one single sample. This strategy was used in, e.g., the recent cryptanalysis of ZUC-256 [YJM20] and SNOW 3G [YJM19], in which weight-4 multiples are used. In SNOW-V, an *equivalent* $32 \times 16$-bit LFSR has a feedback polynomial of weight 12.

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 ones. The closest variant is SNOW-V$_{\boxplus_{32},\boxplus_8}$, in which one $\boxplus_{32}$ (four parallel 32-bit adders) is replaced by $\boxplus_8$ (16 parallel 8-bit adders), and the complexity of the correlation attack against it is $2^{377}$. Correlation attacks are focused on recovering the internal state and thus require a single long keystream under a fixed (key, IV) pair.

**Our contribution.** 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. Unlike the classical approach, e.g., the above mentioned, where one first approximates the FSM and thereafter cancels the LFSR contribution by combining expressions at several time instances, we do it vice-versa and cancel the LFSR contribution directly without combining several approximations. We explore the fact that three LFSR registers appear *twice* in three consecutive keystream words – the minimum needed for the FSM approximation in SNOW-V – and moreover, they happen to contribute linearly in SNOW-V$_{\oplus}$ thus can be directly cancelled.

Therefore, 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 $2^{303}$. A single noise sample is collected from just *three 128-bit consecutive keystream words*, and the samples can be collected from many short keystream sequences under different (key, IV) pairs.

Though none of existing and our 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. Table 1 lists the main existing cryptanalysis results against SNOW-V, and comparison with new results in this paper.

**Outline.** We first provide some 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 the reduced variant SNOW-V$_{\oplus}$. We end the paper with conclusions in Section 6.

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

| Attack | Complexity | Data | Reference |
|---|---|---|---|
| Guess-and-Determine | $2^{512}$ | 7 keystream words | [CDM20] |
| | $2^{406}$ | 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] |
| Differential Distinguisher | $2^{97}$ (4 rounds) | $2^{97}$ | [HII+21] |
| Differential Key Recovery | $2^{154}$ (4 rounds) | $2^{27}$ | |

$^{*}$ The attack has memory complexity $2^{128}$ as it needs a volatile table of size $2^{128}$ bits.
$^{**}$ The attack is applied on the reduced variant SNOW-V$_{\boxplus_{32},\boxplus_{8}}$.
$^{***}$ The attack is applied on the reduced variant SNOW-V$_{\oplus}$.

# 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 binary 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 $\mathbf{a}, \mathbf{b}$ of length $t$, $\mathbf{a} = (a_{t-1}, \ldots, a_1, a_0)$ and $\mathbf{b} = (b_{t-1}, \ldots, b_1, b_0)$, 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 computed as $\mathbf{ab} = \oplus_{i=0}^{t} a_i b_i$, where $a_i b_i$ is the multiplication over $\mathbb{F}_{2^m}$. We sometimes also use $(a_{t-1}, \ldots, a_1, a_0) \cdot (b_{t-1}, \ldots, b_1, b_0)$ 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 $(x_{15}, x_{14}, \ldots, x_1, x_0)$, 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., $x_{1||5||7}$ denotes $x_1||x_5||x_7$.

## 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 the LFSR part and the FSM.

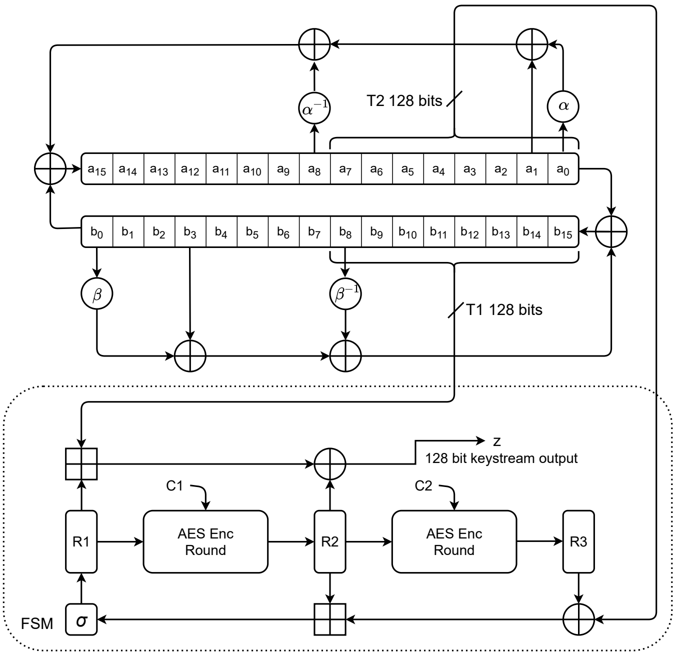
$\alpha^{-1}$
T2 128 bits
$\alpha$
$a_{15}$ $a_{14}$ $a_{13}$ $a_{12}$ $a_{11}$ $a_{10}$ $a_9$ $a_8$ $a_7$ $a_6$ $a_5$ $a_4$ $a_3$ $a_2$ $a_1$ $a_0$
$b_0$ $b_1$ $b_2$ $b_3$ $b_4$ $b_5$ $b_6$ $b_7$ $b_8$ $b_9$ $b_{10}$ $b_{11}$ $b_{12}$ $b_{13}$ $b_{14}$ $b_{15}$
$\beta$
$\beta^{-1}$
T1 128 bits
z
128 bit keystream output
C1
C2
R1
AES Enc Round
R2
AES Enc Round
R3
FSM
$\sigma$

**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 $\mathbb{F}_{2^{16}}$. These elements in LFSR-A, denoted $a_{15}, \dots, a_0$, and LFSR-B, denoted $b_{15}, \dots, b_0$, 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 LFSR-A and LFSR-B at clock $t$ by $(a_{15}^{(t)}, \dots, a_0^{(t)})$ 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:

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

where $\alpha, \beta$ 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,

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

The three registers are then updated as follows:

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

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

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

where $\text{AES}_R()$ is one AES encryption round and $\sigma$ 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 and represents the whole state of the cipher. We can then get the following expressions:

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

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

$$l_\beta(X) = (\beta(X_{15||14}) \; || \cdots || \; \beta(X_{1||0})) \oplus X_{\gg 3 \cdot 2}, \tag{5}$$

$$h_\beta(X) = (\beta^{-1}(X_{15||14}) \; || \cdots || \; \beta^{-1}(X_{1||0})) \oplus X_{\ll 5 \cdot 2}, \tag{6}$$

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 $\beta$ or $\beta^{-1}$ 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 $16 \times 16$-bit matrices of $\beta$ or $\beta^{-1}$. The binary matrix representations of $\beta$ and $\beta^{-1}$ 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:

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

# 3 The first guess-and-determine attack ($T = 2^{384}$)

In this section, we fill the gaps of the GnD attack in [CDM20] and improve the complexity from $2^{512}$ there down to $2^{384}$. 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. Knowing the whole state of a stream cipher at a certain time allows to trivially recover the whole keystream corresponding to the specific secret key and IV. If the initialisation phase has no special protection, one can even recover the secret key.

In this paper, we call every ordered tuple of 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 computed 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 already known (either guessed or 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, $R3, B1$ and $A0$. The guessing path is quite similar to the one in [CDM20], which guesses $R1, R2, R3$ instead, and then similarly deriving $B0, B1, A0$. The derivations for $R3$ and $B1$ are simple, while tricky for $A0$. It is assumed in [CDM20] that $A0$ can be derived efficiently with negligible complexity but no details are provided. We will fill this gap in Section 3.2.2 by breaking down $A0$ into bytes in a similar manner as in [JLH20], but handling the order of derivations and carries in a better way. After that, we use one more keystream word $z^{(t+2)}$ to determine the final state variable $A1$ using the same way for deriving $A0$ with negligible time, instead of purely guessing it with complexity $2^{128}$ as done in [CDM20], which helps to reduce the total complexity $2^{512}$ there to $2^{384}$. Finally we 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 F.

### 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:

$$C = l_\beta(B0) \oplus h_\beta(B1), \tag{8}$$

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

Correspondingly, the three keystream words can be rewritten as:

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

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 $R3, B1$ and $D$ will be directly determined from Equation 10, respectively. 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 $A0$ (and $A1$) without additional guesses, the overall GnD complexity will be exactly $2^{384}$. We next show how we efficiently find the solutions of $A0$ in Section 3.2.2, and use the same method to derive the last state variable $A1$ with negligible complexity in Section 3.2.3.

### 3.2.2 Deriving $A0$ using a 10-step recursive enumeration

$A0$ is determined using Equation 9, while we mention that even when all other variables in Equation 9 are fixed, the value of $A0$ might not be uniquely or directly determined as $A0$ appears twice in the equation with non-linear operations in between. So the task now is to efficiently find the solutions for $A0$ in Equation 9, and we next show how we achieve it in a byte-wise fashion. Each byte of $D$, $D_i$ $(15 \geq i \geq 0)$ is expressed as:

$$D_i = (R2_j \boxplus_8 (R3_j \oplus A0_j) \boxplus_8 u_j) \boxplus_8 (A0_i \oplus C_i) \boxplus_8 v_i, \quad j = \sigma(i), \tag{11}$$

where $u_j, v_i \in \{0, 1\}$ are carry bits that arrive from arithmetic additions of the previous bytes. We call these byte-wise equations as *D-equations*. Note that some 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 $A0$ in a careful order in 10 steps as given in Table 2.

**Table 2:** The 10 steps to derive $A0$.

| | | |
|---|---|---|
| **Step 0:** | $D_0$ | $= (R2_0 \boxplus_8 (R3_0 \oplus A0_0) \boxplus_8 u_0) \boxplus_8 (A0_0 \oplus C_0) \boxplus_8 v_0$ |
| | where | $u_0 = v_0 = 0$ |
| | derive | $\rightarrow (A0_0, u_1, v_1)$ |
| **Step 1:** | $D_1$ | $= (R2_4 \boxplus_8 (R3_4 \oplus A0_4) \boxplus_8 u_4) \boxplus_8 (A0_1 \oplus C_1) \boxplus_8 v_1$ |
| | $D_4$ | $= (R2_1 \boxplus_8 (R3_1 \oplus A0_1) \boxplus_8 u_1) \boxplus_8 (A0_4 \oplus C_4) \boxplus_8 v_4$ |
| | where | $u_4 = v_4 = 0$ and $u_1, v_1$ are known from Step 0 |
| | derive | $\rightarrow (A0_1, A0_4, u_2, v_2, u_5, v_5)$ |
| **Step 2:** | $D_5$ | $= (R2_5 \boxplus_8 (R3_5 \oplus A0_5) \boxplus_8 u_5) \boxplus_8 (A0_5 \oplus C_5) \boxplus_8 v_5$ |
| | where | $u_5, v_5$ are known from Step 1 |
| | derive | $\rightarrow (A0_5, u_6, v_6)$ |
| **Step 3:** | $D_2$ | $= (R2_8 \boxplus_8 (R3_8 \oplus A0_8) \boxplus_8 u_8) \boxplus_8 (A0_2 \oplus C_2) \boxplus_8 v_2$ |
| | $D_8$ | $= (R2_2 \boxplus_8 (R3_2 \oplus A0_2) \boxplus_8 u_2) \boxplus_8 (A0_8 \oplus C_8) \boxplus_8 v_8$ |
| | where | $u_8 = v_8 = 0$ and $u_2, v_2$ are known from Step 1 |
| | derive | $\rightarrow (A0_2, A0_8, u_3, v_3, u_9, v_9)$ |
| **Step 4:** | $D_3$ | $= (R2_{12} \boxplus_8 (R3_{12} \oplus A0_{12}) \boxplus_8 u_{12}) \boxplus_8 (A0_3 \oplus C_3) \boxplus_8 v_3$ |
| | $D_{12}$ | $= (R2_3 \boxplus_8 (R3_3 \oplus A0_3) \boxplus_8 u_3) \boxplus_8 (A0_{12} \oplus C_{12}) \boxplus_8 v_{12}$ |
| | where | $u_{12} = v_{12} = 0$ and $u_3, v_3$ are known from Step 3 |
| | derive | $\rightarrow (A0_3, A0_{12}, u_{13}, v_{13})$ |
| **Step 5:** | $D_6$ | $= (R2_9 \boxplus_8 (R3_9 \oplus A0_9) \boxplus_8 u_9) \boxplus_8 (A0_6 \oplus C_6) \boxplus_8 v_6$ |
| | $D_9$ | $= (R2_6 \boxplus_8 (R3_6 \oplus A0_6) \boxplus_8 u_6) \boxplus_8 (A0_9 \oplus C_9) \boxplus_8 v_9$ |
| | 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 $(R1, R2, B0)$, we could have different numbers, either zero or nonzero, of solutions for $A0$. Most of the guessing values will not even pass the first step in Table 2 as no valid solutions exist for the first $D$-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 values of $A0_0$ 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 a number of efficient ways.

### 3.2.3 Deriving $A1$ and final verification

After the above initial guessing and enumeration, we now 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 use the fourth keystream word $z^{(t+2)}$:

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

where

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

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 9, and therefore, we could use the ten steps in Table 2 to enumerate all solutions for $A1$. The distribution of the 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 remaining four, 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 the subsequent three keystream words for verification. 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 exactly one.

In Equation 11, the input carry bits $u_j, v_i$ can be removed by setting $R2'_j = R2_j \boxplus u_j$ and $D'_i = D_i \boxminus v_i$, respectively, which will not influence the distribution or the average value of the number of solutions. The ten steps in Table 2 can be divided into two equivalent types, which we denote by *Type-1* and *Type-2* $D$-equations.

*Type-1* equations have the form:

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

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

*Type-2* equations have the form:

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

where $X_1, X_2$ are two unknown variables that we want to enumerate, while others are $n$-bit known 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 number of solutions for the unknown $X$-bytes, given that other known variables are uniformly distributed. We exhaustively (with some optimisations and cut-offs) try all the values of the known variables, and count the number of solutions for the unknowns.

Table 3 presents the probabilities of $X$ having different numbers of solutions for *Type-1* equations corresponding to a random tuple $(A, B, C, D)$ over $\mathbb{F}_{2^n}$. The probabilities are derived through $p = x/f$, where $x$'s are the integers in the table and $f$ is the corresponding normalisation factor. The probability of having at least one solution when $n = 8$ can be computed easily as $2^{-3.91}$. This means that in Equation 9, 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 of the combinations we just stop and trace back to the last step of the recursion. 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.

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

| #Solutions<br>normalisation factor $f \rightarrow$ | n=1<br>$2^1$ | n=2<br>$2^3$ | n=3<br>$2^5$ | n=4<br>$2^7$ | n=5<br>$2^9$ | n=6<br>$2^{11}$ | n=7<br>$2^{13}$ | n=8<br>$2^{15}$ |
|---|---|---|---|---|---|---|---|---|
| 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 |

We also derive the distribution and average value of the number of solutions for *Type-2* equations using a similar technique. The distribution table under different $n$ values is 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 $2^{-3.53 \times 6 - 3.91 \times 4} = 2^{-36.82}$. This means that only a small fraction, i.e., $2^{-36.82}$, 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 $2^{36.82}$ in average, and the overall average number of solutions is still one.

### 3.3.2 The total attack complexity

As it was mentioned earlier, the large fraction of the guesses $2^{384} \cdot (1 - 2^{-3.91})$ will fail in Step 0 in Table 2, as it involves solving a *Type-1* equation and the probability of having at least one solution there is $2^{-3.91}$. The remaining small fraction, $2^{384} \cdot 2^{-3.91} \approx 2^{380.09}$, of the guesses 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 end-nodes, exactly one per guessing tuple $(R1, R2, B0)$ *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 $2^{384}$.

### 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.

**Bit-wise enumeration recursion instead of byte-wise.** Recall that the simplest way to enumerate all solutions for $A0$ is to make a byte-wise recursion of depth ten, and in each step we loop over the unknown *byte(s)* of $A0$, thus the overall enumeration recursion will have a constant factor $c = 256$ steps. However, we can change the recursion to be deeper with depth $10 \cdot 8$ and search for solutions of each *bit(s)* of $A0$. This will shrink the constant $c$ from 256 down to 2 [1], since now we only need to test the binary bit-value(s) before going to the next recursion depth while considering the resulting carriers from the current step. So we can enumerate the 128-bit unknown $A0$ by deriving one or two bits in each recursive step. We have actually implemented such a bit-wise recursive enumeration algorithm, see Appendix C. Note that the proposed recursion is linear with a fixed depth, and may as well be organised as a number of (many) nested loops.

**Precomputed lookup tables.** Another approach is to precompute lookup tables helping to instantly give the list of sub-solutions for each tuple of $D$-equations. The tables record all the possible values of the known variables and the corresponding solutions for the unknowns.

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

[1] For a Type-2 $D$-equation we can loop over the first unknown bit-value (0 or 1, thus $c = 2$), then derive the second unknown bit-value using the first equation, and then test the pair of the bits using the second equation.

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 $p_{0-2} = 2^{-3.91\times2-3.52} = 2^{-11.34}$. 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 $T'$ 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.

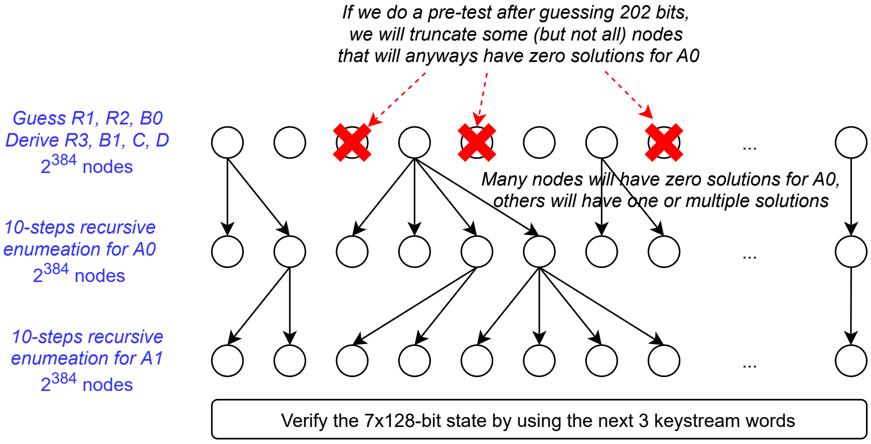
If we do a pre-test after guessing 202 bits,
we will truncate some (but not all) nodes
that will anyways have zero solutions for A0
Guess R1, R2, B0
Derive R3, B1, C, D
$2^{384}$ nodes
Many nodes will have zero solutions for A0,
others will have one or multiple solutions
10-steps recursive
enumeation for A0
$2^{384}$ nodes
10-steps recursive
enumeation for A1
$2^{384}$ nodes
...
Verify the 7x128-bit state by using the next 3 keystream words

**Figure 2:** Illustration of the first GnD attack.

# 4 The second guess-and-determine attack ($T = 2^{378.16}$)

In this section, we provide a second guess-and-determine attack which can further reduce the complexity by using one additional "backward" keystream block $z^{(t-2)}$ as *side information* to truncate more "forward" guessing paths. Thus, this approach needs eight keystream words. The improvement over the first GnD attack is not so significant, 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 it.

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

If we want to further reduce the complexity of the first GnD attack, we could try to see if we can use some additional information, *besides* those seven keystream words that are already involved in the first attack. With such additional information, we can truncate some portion of the guessing paths that have solutions for the $D$-equations while not for the additional information, *proportionally*. Thus, the average number of end-points $2^{384}$ will be reduced proportionally as well. 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:

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

According to the LFSR update function, 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 that other variables are either known, guessed, or determined):

$$\begin{aligned}
z^{(t-2)} = &\,(\underbrace{\mathrm{AES}_R^{-1}(\mathrm{AES}_R^{-1}(R3))}_{X} \boxplus_{32} B0^{(t-1)})\\
&\oplus \mathrm{AES}_R^{-1}(\underbrace{(\sigma(R1) \boxminus_{32} \mathrm{AES}_R^{-1}(R3)) \oplus h_\beta(B0) \oplus B1}_{Y} \oplus l_\beta(B0^{(t-1)})).
\end{aligned}$$

Using $X, Y$ to denote the expressions in the brackets, we could simplify the above equation as $z^{(t-2)} = (X \boxplus_{32} B0^{(t-1)}) \oplus \mathrm{AES}_R^{-1}(Y \oplus l_\beta(B0^{(t-1)}))$. 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 $(X, Y)$, 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 $z^{(t-2)}$, otherwise 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 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 $(X, Y)$ as it is easier to illustrate how $z^{(t-2)}$ is exploited.

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:

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

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 $2^{384-5.84} \approx 2^{378.16}$.

## 4.2 Scenario of the second GnD attack

The flowchart of the second GnD attack is given in Appendix F, 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:

$$\begin{aligned} &R3 \text{ from: } X = \mathrm{AES}_R^{-1}(\mathrm{AES}_R^{-1}(R3)), \\ &R2 \text{ from: } z^{(t-1)} = (\mathrm{AES}_R^{-1}(R2) \boxplus_{32} B0) \oplus \mathrm{AES}_R^{-1}(R3)). \end{aligned}$$

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 $B1$:

$$\begin{aligned} z^{(t)} \oplus R2 &= R1 \boxplus_{32} B1, \\ Y \oplus h_\beta(B0) &= (\sigma(R1) \boxminus_{32} \mathrm{AES}_R^{-1}(R3)) \oplus B1. \end{aligned} \tag{12}$$

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 E. 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) **Enumerate** all solutions for $A0$ as done in Section 3.2.2 in the first GnD attack;

(6) **Enumerate** all solutions for $A1$ as done in Section 3.2.3 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:

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

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 return “Yes”. This is similar to the process of computing $p_z$ in Appendix G. 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 $2^{384}$.

However, 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 $(X, Y)$ are the same. So for every new $X$ we must ensure that the value of $Y$ is new, and skip the cases when the pair $(X, Y)$ 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 that specific value of $X$. 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: 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 $2^{378}$. However, in order to gain the advantage in time complexity over the first GnD attack we have to use memory of size $2^{128}$ bits.

## 5 Linear cryptanalysis of SNOW-V$_\oplus$

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 it from the uniform random distribution is in the order of $1/\epsilon(D)$ [BJV04].

In this section, we perform linear cryptanalysis of SNOW-V and propose a distinguishing attack with complexity $2^{303}$ against a reduced version SNOW-V$_\oplus$ in which the 32-bit adders are replaced with exclusive-OR. In the attack we explore the feature that three consecutive keystream words contain the contribution from the LFSR linearly and redundantly, due to the chosen tap positions of $T1$ and $T2$ in the design. Thus, unlike the linear attacks against the predecessors SNOW 2.0 (e.g., [WBDC03, NW06]), and SNOW 3G [YJM19], where one first approximates the FSM and then cancels out the contribution from the LFSR either according to the feedback polynomial or a multiple of it, here we do it vice-versa. We will first cancel the LFSR variables locally within these three keystream words without combining several time instances, and thereafter construct a noise expression based on the remaining expressions over the FSM variables.

## 5.1 Linear approximation

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 $16 \times 16$-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 first. To achieve so, we apply two linear operations $l_\beta, h_\beta$, which can be expressed as two $128 \times 128$ binary matrices, to $z^{(t)}$ and $z^{(t-1)}$, respectively, and introduce a new 128-bit variable $W$ defined as below:

$$W = l_\beta(z^{(t-1)}) \oplus h_\beta(z^{(t)}) \oplus z^{(t+1)}. \tag{13}$$

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

$$\begin{aligned} &E_0 = W_0, \quad E_1 = W_1 \oplus W_4, \quad E_2 = W_5, \quad E_3 = W_2 \oplus W_8, \quad E_4 = W_6 \oplus W_9, \\ &E_5 = W_{10}, \quad E_6 = W_3 \oplus W_{12}, \quad E_7 = W_7 \oplus W_{13}, \quad E_8 = W_{11} \oplus W_{14}, \quad E_9 = W_{15}, \end{aligned}$$

where $W_i$ is the $i$-th byte of $W$. Each byte-wise expression $E_k$ $(0 \le k \le 9)$ 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:

$$\begin{aligned} E_k = &\bigoplus_{i=0}^{15} [l_{k,i}^{(1)} \cdot R1_i \oplus n_{k,i}^{(1)} \cdot S(R1_i)] \\ &\oplus [l_{k,i}^{(2)} \cdot R2_i \oplus n_{k,i}^{(2)} \cdot S^{-1}(R2_i)] \oplus [l_{k,i}^{(3)} \cdot R3_i \oplus n_{k,i}^{(3)} \cdot S^{-1}(R3_i)], \end{aligned} \tag{14}$$

where $l_{k,i}^{(j)}, n_{k,i}^{(j)}$ $(j \in \{1, 2, 3\}, 0 \le k \le 9, 0 \le i \le 15)$ are $8 \times 8$ binary matrices that can be derived following the expressions of $W$ 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:

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

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 $w$-bit noise $N_w$ using the ten 8-bit $E$-expressions, which is expressed in a matrix form as below:

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

where $c_i$'s, $0 \le i \le 9$, are $w \times 8$ binary matrices that the attacker can choose freely, but with the constraint that the rank of $\mathbf{c}$ is $w$, i.e., all $w$ rows are nonzero and linearly independent. For simplicity, 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 $X_j$ $(0 \le j \le 47)$ can be regarded as a uniformly distributed random variable, and $Y_j$ is the corresponding value after applying 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 $N_w$ can be rewritten as:

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

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 14. It is therefore clear that the total $w$-bit noise $N_w$ consists of at most 48 sub-noise parts:

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

where $a_i$ and $b_i$ are $w \times 8$ binary sub-matrices, constituted from the $w$ 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 $N_w$ 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 c.** Let us select $k$ distinct indices $\{i_1, i_2, \ldots, i_k\} \in \{0, 1, \ldots, 47\}$, 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 $j = 1, 2, \ldots, k$, 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 $\mathbf{B}$:

$$\mathbf{K}_{10\cdot 8\times 2k\cdot 8} =$$

$$\begin{pmatrix} \mathbf{A}_{[0:7;\ 8i_1:8i_1+7]} & \mathbf{B}_{[0:7;\ 8i_1:8i_1+7]} & \cdots & \mathbf{A}_{[0:7;\ 8i_k:8i_k+7]} & \mathbf{B}_{[0:7;\ 8i_k:8i_k+7]} \\ \mathbf{A}_{[8:15;\ 8i_1:8i_1+7]} & \mathbf{B}_{[8:15;\ 8i_1:8i_1+7]} & \cdots & \mathbf{A}_{[8:15;\ 8i_k:8i_k+7]} & \mathbf{B}_{[8:15;\ 8i_k:8i_k+7]} \\ & & \vdots & & \\ \mathbf{A}_{[72:79;\ 8i_1:8i_1+7]} & \mathbf{B}_{[72:79;\ 8i_1:8i_1+7]} & \cdots & \mathbf{A}_{[72:79;\ 8i_k:8i_k+7]} & \mathbf{B}_{[72:79;\ 8i_k:8i_k+7]} \end{pmatrix},$$

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

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

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 $w = 80 - r$ nonzero linear combinations that will map through $\mathbf{K}$ to zero. This also explains how the size $w$ for the total noise $N_w$ was 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 $w = 80 - r$ rows of $\mathbf{K}'$ are zeroes, while the matrix $\mathbf{c}$ will be of the full-rank 80. Then we keep the last $w$ rows of $\mathbf{c}$ and discard all other $r$ rows, thus deriving the desired $\mathbf{c}_{w\times 80}$ 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 $w$ and the corresponding linear masking matrix $\mathbf{c}$. Then given the derived $\mathbf{c}_{w\times 80}$, we construct the distribution of the total $w$-bit noise $N_w$ 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 $w$ 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 $\mathbf{c}_{w\times 80}$, 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 $w$-bit total noise become uniformly distributed. In such a case, we can try to *correct* the masking matrix $\mathbf{c}_{w\times 80}$ by removing those rows where the rows of $b_i$ are nonzero. In this way we shrink $w$ down but get $a_i = b_i = 0$. If $w$ 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 $N_w$ and compute the corresponding bias. When constructing the distribution, we can utilise the Walsh-Hadamard Transforms to speed up the convolution of the 48 $w$-bit sub-noises [MJ05, YJM19].

**Results.** In our simulations we managed to find a 16-bit approximation $N_{16}$, i.e., $w = 16$, 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},
{ 0x40c1000000600000ULL, 0x0100ULL}, { 0x0000000000000008ULL, 0x0000ULL},
{ 0x0000000000000060ULL, 0x0000ULL}, { 0x0000000000000021ULL, 0x0000ULL},
{ 0x0000000000000004ULL, 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

If all arithmetic 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 $2^{303}$ different triples of consecutive keystream words and construct the sequence of 16-byte words $\{W\}$ of length $2^{303}$ by applying Equation 13 for each triple; then build the sequence of 10-byte words $\{E\}$ from $\{W\}$; and, finally, apply the linear masking $\mathbf{c}_{\mathbf{16}\times\mathbf{80}}$ given in Listing 4 to each word in $\{E\}$, thus receiving the sequence of length $2^{303}$ of biased 16-bit noise samples $\{N_{16}\}$, which can be distinguished from random.

The bias derived in our attack does not depend on the key or IV, and the time width to build a single sample is just three keystream words, which means that the data in our attack can be collected from many short keystream sequences under different (key, IV) pairs. Though the data complexity is still out of reach in practice, the attacking scenario is more relevant to the practical situation. 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].

**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, 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 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 can be reduced. A very interesting open problem would be to investigate whether there are possible speed-ups for these kind of GnD attacks using quantum computers. For linear cryptanalysis against LFSR-based stream ciphers, it might be interesting to check if the LFSR contribution can be cancelled locally first, then the remaining equations on FSM variables may be used to construct a biased noise.

## Acknowledgements

We thank the reviewers for valuable comments and questions that made it possible to improve this paper at a great extent. This work was in part financially supported by the Swedish Foundation for Strategic Research, grant RIT17-0005 and the ELLIIT program. Jing Yang is also supported by the scholarship from the National Digital Switching System Engineering and Technological Research Center, China.

## 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.

[CBB20] Andrea Caforio, Fatih Balli, and Subhadeep Banik. Melting SNOW-V: improved lightweight architectures. *Journal of Cryptographic Engineering*, pages 1–21, 2020.

[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.

[EJ02] Patrik Ekdahl and Thomas Johansson. A new version of the stream cipher SNOW. In *International Workshop on Selected Areas in Cryptography*, pages 47–61. Springer, 2002.

[EJMY19] Patrik Ekdahl, Thomas Johansson, Alexander Maximov, and Jing Yang. A new SNOW stream cipher called SNOW-V. *IACR Transactions on Symmetric Cryptology*, pages 1–42, 2019.

[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*, pages 378–410, 2021.

[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, 2020.

[MJ05] Alexander Maximov and Thomas Johansson. Fast computation of large distributions and its cryptographic applications. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 313–332. Springer, 2005.

[NW06] Kaisa Nyberg and Johan Wallén. Improved linear distinguishers for SNOW 2.0. In *International Workshop on Fast Software Encryption*, pages 144–162. Springer, 2006.

[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.

[WBDC03] Dai Watanabe, Alex Biryukov, and Christophe De Canniere. A distinguishing attack of SNOW 2.0 with linear masking method. In *International Workshop on Selected Areas in Cryptography*, pages 222–233. Springer, 2003.

[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.

[YJM20] Jing Yang, Thomas Johansson, and Alexander Maximov. Spectral analysis of ZUC-256. *IACR transactions on symmetric cryptology*, pages 266–288, 2020.

# A The matrices

```
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
1 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 1 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 1 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 1 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 1          1 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 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 1 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 1 | 0 0 0 0 0 0 0 1          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 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 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 1 0 0 0 0 1          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          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          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 1 0 1          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 1 1          1 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 0
```

**Listing 5:** The 16 × 16 binary matrices for $\beta$ (left) and $\beta^{-1}$ (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 $l_\beta$ and $h_\beta$ 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 of having at least one solution of $A0$ is $2^{-36.82}$. However, if solutions exist, the average number of solutions will be $2^{36.82}$. 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)
        {
            A0[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)
        {
            A0[i] = Xi;
            A0[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] ^ A0[j]) + (int)u[j]);
        int nv = (nu & 0xff) + (int)(A0[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;
        }

        int i = S[step], j = ((i >> 2) | (i << 2)) & 0xf; // j = sigma(i)
        if (i == j) solve1(step, i);
        else solve2(step, i, j);
    }

};
```

**Listing 7:** A possible recursion organisation for 10-steps.

## D The distribution table of solutions for *Type-2* equations

Consider $n$-bit variables $A_{1,2}, B_{1,2}, C_{1,2}, D_{1,2}, X_{1,2}$ and two $n$-bit equations:

$$A_1 = (B_1 \boxplus_n (C_1 \oplus X_1)) \boxplus_n (X_2 \oplus D_1),$$
$$A_2 = (B_2 \boxplus_n (C_2 \oplus X_2)) \boxplus_n (X_1 \oplus D_2).$$

Table 4 contains the probabilities of the pair $(X_1, X_2)$ having $k$ solutions for a random tuple $(A_{1,2}, B_{1,2}, C_{1,2}, D_{1,2})$, which are derived through $p = x/f$, where $x$'s are the integers in the table and $f$ is the corresponding normalisation factor. For the GnD attack against SNOW-V we are interested in the distribution where $n = 8$.

| #Solutions | n=1 | n=2 | n=3 | n=4 | n=5 | n=6 | n=7 | n=8 |
|---|---|---|---|---|---|---|---|---|
| factor $f$ → | $2^2$ | $2^3$ | $2^7$ | $2^{10}$ | $2^{14}$ | $2^{18}$ | $2^{22}$ | $2^{26}$ |
| 0 | 1 | 5 | 91 | 793 | 13484 | 225652 | 3734648 | 61316512 |
| 2 | 1 | 2 | 16 | 64 | 512 | 4096 | 32768 | 262144 |
| 4 |  | 1 | 18 | 119 | 1377 | 14759 | 150417 | 1478903 |
| 8 |  |  | 3 | 43 | 803 | 12265 | 166035 | 2071185 |
| 12 |  |  |  | 1 | 29 | 529 | 7761 | 100077 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 16 | | | | 4 | 162 | 3978 | 76314 | 1256786 |
| 20 | | | | | 1 | 33 | 661 | 10405 |
| 24 | | | | | 5 | 205 | 5001 | 94273 |
| 28 | | | | | 1 | 33 | 661 | 10405 |
| 32 | | | | | 10 | 536 | 16552 | 385832 |
| 36 | | | | | | 3 | 117 | 2691 |
| 40 | | | | | | 5 | 225 | 5901 |
| 44 | | | | | | 1 | 37 | 809 |
| 48 | | | | | | 18 | 978 | 30258 |
| 52 | | | | | | 1 | 37 | 809 |
| 56 | | | | | | 5 | 225 | 5901 |
| 60 | | | | | | 1 | 41 | 985 |
| 64 | | | | | | 24 | 1632 | 61440 |
| 68 | | | | | | | 1 | 41 |
| 72 | | | | | | | 19 | 981 |
| 76 | | | | | | | 1 | 41 |
| 80 | | | | | | | 18 | 1050 |
| 84 | | | | | | | 5 | 217 |
| 88 | | | | | | | 5 | 245 |
| 92 | | | | | | | 1 | 41 |
| 96 | | | | | | | 56 | 3864 |
| 100 | | | | | | | 1 | 43 |
| 104 | | | | | | | 5 | 245 |
| 108 | | | | | | | 1 | 49 |
| 112 | | | | | | | 18 | 1050 |
| 116 | | | | | | | 1 | 41 |
| 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 probability of valid solutions in Equation 12

In this section, we compute the probability of valid solutions in Equation 12. We recall that the equations are:

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

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

Thus we have a simplified system of equations:

$$\begin{aligned} U &= R1 \boxplus_{32} B1, \\ V &= (\sigma(R1) \boxminus_{32} W) \oplus B1. \end{aligned} \tag{15}$$

According to Equation 15, we have $B1 = (\sigma(R1) \boxminus_{32} W) \oplus V$, and further get:

$$U = R1 \boxplus_{32} ((\sigma(R1) \boxminus_{32} W) \oplus V). \tag{16}$$

The distributions of number of solutions of Equation 15 and Equation 16 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 16 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:

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

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.

## F The flowcharts of the guess-and-determine attacks

Figure 3 presents simple illustrations of the proposed GnD attacks.

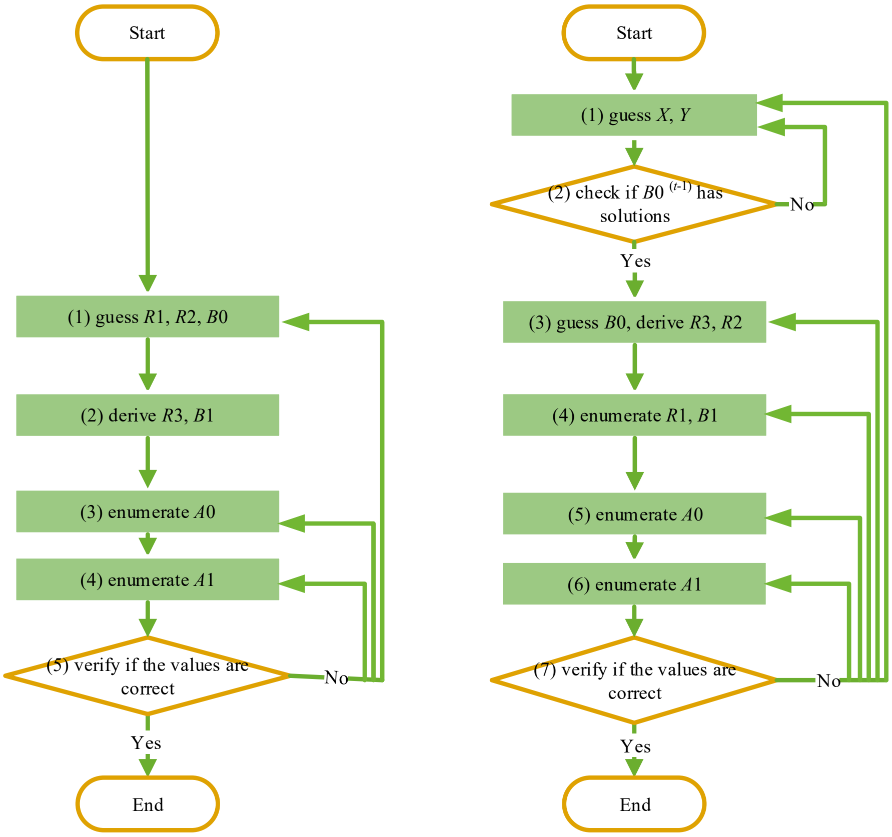

**Figure 3:** Illustration of the GnD attacks (left: first; right: second).

## 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:

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

where $\mathrm{AES}_R^{-1}(X)$ can be expressed as $S^{-1}(L^{-1} \cdot X)$, and $l_\beta$ 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 $\beta$, i.e.,

$$[\psi B0_{i,i+1,i+2,i+3}] = (\beta(B0_{i||i+1})_0, \beta(B0_{i||i+1})_1, \beta(B0_{i+2||i+3})_0, \beta(B0_{i+2||i+3})_1),$$

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 $\beta$ 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 $B0_{13,14,15}$, solve $B0_{12}$ with valid solutions of probability $p_z(B0_{12})$, and further derive $B0_{1,6,11}$ correspondingly.

**Step 2.** Consider the second group of equations:

$$\begin{aligned} z_{13} &= (B0_{13} \boxplus_8 X_{13}) \oplus S^{-1}((9,e,b,d) \cdot ([\psi B0_{8,9,10,11}] \oplus (B0_{14}, B0_{15}, 0, 0)) \oplus Y'_{13}), \\ z_8 &= (B0_8 \boxplus_8 X_8) \oplus S^{-1}((e,b,d,9) \cdot ([\psi B0_{8,9,10,11}] \oplus (B0_{14}, B0_{15}, 0, 0)) \oplus Y'_8), \\ z_7 &= (B0_7 \boxplus_8 X_7) \oplus S^{-1}((b,d,9,e) \cdot ([\psi B0_{8,9,10,11}] \oplus (B0_{14}, B0_{15}, 0, 0)) \oplus Y'_7), \\ z_2 &= (B0_2 \boxplus_8 X_2) \oplus S^{-1}((d,9,e,b) \cdot ([\psi B0_{8,9,10,11}] \oplus (B0_{14}, B0_{15}, 0, 0)) \oplus Y'_2). \end{aligned}$$

Here we can only freely choose $B0_{9,10}$, as the values of $B0_{11,14,15}$ 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 $B0_{13} \boxplus X_{13}$, which is also known. Thus we need to find solutions of $B0_8$ that satisfies the two equations below:

$$\begin{aligned} z_{13} &= X'_{13} \oplus S^{-1}(9 \cdot (B0_8 \ll 1) \oplus e \cdot (B0_8 \gg 7) \oplus Y''_{13}), \\ z_8 &= (B0_8 \boxplus_8 X_8) \oplus S^{-1}(e \cdot (B0_8 \ll 1) \oplus b \cdot (B0_8 \gg 7) \oplus Y''_8). \end{aligned}$$

We have computed that the probability of valid solutions for $B0_8$ is:

$$p_z(B0_8) \approx 0.363230705 \cdot 2^{-8}.$$

This can be understood in another way: the probability of $B0_8$ 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 $B0_8$, we can further derive $B0_7$ and $B0_2$ uniquely. Thus in Step 2 we can loop over $B0_{9,10}$, solve $B0_8$ with valid solutions of probability $p_z(B0_8)$, and further derive $B0_{2,7}$.

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

$$\begin{aligned}
z_{14} &= (B0_{14} \boxplus_8 X_{14}) \oplus S^{-1}((d,9,e,b) \cdot ([\psi B0_{4,5,6,7}] \oplus [B0_{10,11,12,13}]) \oplus Y'_{14}),\\
z_9 &= (B0_9 \boxplus_8 X_9) \oplus S^{-1}((9,e,b,d) \cdot ([\psi B0_{4,5,6,7}] \oplus [B0_{10,11,12,13}]) \oplus Y'_9),\\
z_4 &= (B0_4 \boxplus_8 X_4) \oplus S^{-1}((e,b,d,9) \cdot ([\psi B0_{4,5,6,7}] \oplus [B0_{10,11,12,13}]) \oplus Y'_4),\\
z_3 &= (B0_3 \boxplus_8 X_3) \oplus S^{-1}((b,d,9,e) \cdot ([\psi B0_{4,5,6,7}] \oplus [B0_{10,11,12,13}]) \oplus Y'_3).
\end{aligned}$$

The known bytes $B0_{6,7,10,11,12,13}$ are added to the $Y'$-terms, while the bytes $B0_{9,14}$ are added to the $X$-terms. We can freely loop over $B0_5$ and solve the following equation in $B0_4$:

$$z_4 = (B0_4 \boxplus_8 X_4) \oplus S^{-1}(e \cdot (B0_4 \ll 1) \oplus b \cdot (B0_4 \gg 7) \oplus Y''_4).$$

The probability of valid $B0_4$ 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 $B0_4$ solutions, denoted $p_z(B0_4)$, is computed as:

$$p_z(B0_4) \approx 0.363230705 \cdot 2^{-16}.$$

After $B0_4$ having been solved, $B0_3$ can be uniquely determined according to $z_3$. Thus in Step 3 we can loop over $B0_5$, solve $B0_4$ with valid solutions of probability $p_z(B0_4)$, and further derive $B0_3$.

**Step 4.** The last group of equations contain the remaining four byte expressions $z_{0,5,10,15}$ in only one unknown variable $B0_0$, while other variables are already known:

$$\begin{aligned}
z_0 &= (B0_0 \boxplus_8 X_0) \oplus S^{-1}((e,b,d,9) \cdot ([\psi B0_{0,1,2,3}] \oplus [B0_{6,7,8,9}]) \oplus Y'_0),\\
z_5 &= (B0_5 \boxplus_8 X_5) \oplus S^{-1}((9,e,b,d) \cdot ([\psi B0_{0,1,2,3}] \oplus [B0_{6,7,8,9}]) \oplus Y'_5),\\
z_{10} &= (B0_{10} \boxplus_8 X_{10}) \oplus S^{-1}((d,9,e,b) \cdot ([\psi B0_{0,1,2,3}] \oplus [B0_{6,7,8,9}]) \oplus Y'_{10}),\\
z_{15} &= (B0_{15} \boxplus_8 X_{15}) \oplus S^{-1}((b,d,9,e) \cdot ([\psi B0_{0,1,2,3}] \oplus [B0_{6,7,8,9}]) \oplus Y'_{15}).
\end{aligned}$$

Similarly, $B0_0$ 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 $B0_0$ solutions, denoted $p_z(B0_0)$, is:

$$p_z(B0_0) \approx 0.363230705 \cdot 2^{-24}.$$

**Summary.** We can freely choose six bytes of $B0$, i.e., $B0_{5,9,10,13,14,15}$, of total size $2^{48}$, which will result into valid solutions for bytes $B0_{0,4,8,12}$ with probability $p_z(B0_0) \cdot p_z(B0_4) \cdot p_z(B0_8) \cdot p_z(B_{12})$. Other bytes will be further uniquely determined. Thus the total probability $p_z$ is computed as:

$$p_z = 2^{48} \cdot p_z(B0_0) \cdot p_z(B0_4) \cdot p_z(B0_8) \cdot p_z(B_{12}) \approx 2^{-5.84}.$$

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.