SNOW-Vi: an extreme performance variant of SNOW-V for low-end CPUs

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

[1] Ericsson Research, Lund, Sweden
{patrik.ekdahl,alexander.maximov}@ericsson.com
[2] Dept. of Electrical and Information Technology, Lund University, Lund, Sweden
{thomas.johansson,jing.yang}@eit.lth.se

Abstract. In this paper we propose a faster variant of SNOW-V, called SNOW-Vi, that can perform fast enough not only in cloud settings but also on low grade CPUs. The increase in software performance is around 50% in average, up to 92 Gbps. This makes the applicability of the cipher much wider and it covers more use cases. SNOW-Vi differs in the way how the LFSR is updated and also introduces a new location of the tap $T2$ for a stronger security, while everything else is kept the same as in SNOW-V. The security analyses previously done for SNOW-V are not affected in most aspects, and SNOW-Vi provides the same 256-bit security level as SNOW-V.

Keywords: SNOW · Stream Cipher · 5G Mobile System Security.

1 Introduction and motivation

SNOW-V [EJMY19] is the most recent member of the SNOW family of stream ciphers with the design goal to be fast in virtualised environments. However, on low grade CPUs with limited instruction sets, SNOW-V may not perform as good. For example, there might be the use case where the 5G encryption layer is not virtualised, but processed in software on the base station, where typically there is a mixture of dedicated hardware and general CPU resources. These CPUs are normally not server grade but lower grade CPUs that are more suitable for embedding in a base station. By running the encryption layer in software we are then forced to perform fast air encryption on low grade CPUs as well. This possible use case was only partially covered by the SNOW-V design goals, and in this work we found a way how to speed up SNOW-V and thus to extend its usage.

Here we propose SNOW-Vi[1] – an extreme performance variant of SNOW-V, that reaches much higher speeds on a wider variety of platforms, including lower grade CPUs. The basis for SNOW-Vi is not only cloud hosting CPUs with SIMD registers of 256 bit or wider, but covering platforms with only 128 bit registers much better. With this new variant we can tackle the speed requirements also in lower grade CPUs.

The encryption speed of SNOW-Vi is around 50% faster than that of SNOW-V, in average, while the security stands on the same level. And the minimum requirement for the CPU is that it supports the AES round function as an instruction, and at least 128 bit SIMD registers.

This paper is organised as follows. Firstly we present the new design of SNOW-Vi. Secondly, we evaluate the security of SNOW-Vi by revisiting all known analyses for SNOW-V and applying it to this new design, making sure it still fulfils the security goals. Finally, we perform an extensive software evaluation.

[1] "Vi" stands for "Virtualisation, improved".

2 The design

The design of SNOW-Vi, in the parts of keystream generation and initialisation procedure, is exactly the same as in SNOW-V, with the only difference in the LFSR update function and the tap $T2$ moved to the higher half of LFSR-A – these changes dramatically improve the speed in software implementations, and strengthen the security of the cipher. We refer to the original paper of SNOW-V [EJMY19] for all other details of the design. The new LFSR is depicted in Figure 1 and updates as follows:

$$
\begin{aligned}
a^{(t+16)} &= b^{(t)} + \alpha a^{(t)} + a^{(t+7)} \mod g^A(\alpha),\\
b^{(t+16)} &= a^{(t)} + \beta b^{(t)} + b^{(t+8)} \mod g^B(\beta),
\end{aligned}
$$

where the two fields $\mathbb{F}_2^A$ and $\mathbb{F}_2^B$ have the generating polynomials:

$$
\begin{aligned}
g^A(x) &= x^{16} + x^{14} + x^{11} + x^9 + x^6 + x^5 + x^3 + x^2 + 1 \in \mathbb{F}_2[x] \quad (\texttt{0x4a6d}),\\
g^B(x) &= x^{16} + x^{15} + x^{14} + x^{11} + x^{10} + x^7 + x^2 + x + 1 \in \mathbb{F}_2[x] \quad (\texttt{0xcc87}).
\end{aligned}
$$

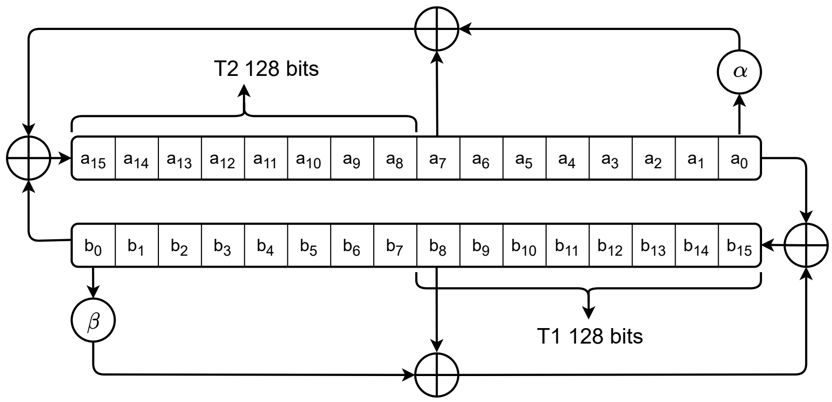

Figure 1: LFSR construction in SNOW-Vi.

3 Security analysis

In this section we perform a step-by-step security re-evaluation of SNOW-Vi based on previously known analyses of SNOW-V, given in [EJMY19, CDM20, JLH20].

3.1 The new tap position of $T2$

While we propose a simplified update functions in the LFSR for a better performance, we also have to ensure the security of the new proposal. By moving the tap $T2$ to the higher half of the LFSR-A we believe the security of SNOW-Vi is strengthened. Below we give more details on motivations for this particular design choice.

From linear analysis perspectives. Let us assume the content of the LFSR is $(A1|A0)$ and $(B1|B0)$, four 128-bit words. Recall the expressions on three consecutive keystream words $z^{(t-1)}$, $z^{(t)}$, and $z^{(t+1)}$:

$$
\begin{aligned}
z^{(t-1)} &= (S^{-1}(L^{-1} \cdot \hat{R}2) \boxplus_{32} T1^{(t-1)}) \oplus S^{-1}(L^{-1} \cdot \hat{R}3),\\
z^{(t)} &= (\hat{R}1 \boxplus_{32} T1^{(t)}) \oplus \hat{R}2,\\
z^{(t+1)} &= (\sigma(\hat{R}2 \boxplus_{32} (\hat{R}3 \oplus T2^{(t)})) \boxplus_{32} T1^{(t+1)}) \oplus L \cdot S(\hat{R}1).
\end{aligned}
$$

Any choice of the LFSR update function, for that particular circular-LFSR construction, would result in the following linear relations:

$$\begin{aligned}
B1^{(t+1)} &= A0^{(t)} \oplus f_\beta(B0^{(t)}, B1^{(t)}),\\
A1^{(t+1)} &= B0^{(t)} \oplus f_\alpha(A0^{(t)}, A1^{(t)}),\\
B0^{(t+1)} &= B1^{(t)},\\
A0^{(t+1)} &= A1^{(t)},
\end{aligned}$$

where f_α and f_β are two linear functions that correspond to the LFSR update procedure. These expressions are generic for both SNOW-V and SNOW-Vi.

In SNOW-V, the taps are $T1 = B1$ and $T2 = A0$, which implies that in 3 consecutive keystream expressions the contribution from the LFSR involves 3 out of 4 128-bit words:

$$\begin{aligned}
T1^{(t)} &= B1^{(t)},\\
T1^{(t-1)} &= B1^{(t-1)} = B0^{(t)},\\
T2^{(t)} &= A0^{(t)},\\
T1^{(t+1)} &= B1^{(t+1)} = A0^{(t)} \oplus f_\beta(B0^{(t)}, B1^{(t)}).
\end{aligned}$$

Note that those 3 LFSR words appear in the 3 keystream expressions twice, thus there is a chance for a biased noise expression by considering three consecutive 128-bit keystream words.

We, however, believe that there is no immediate security threat for SNOW-V as it is most likely that up to 48 SBoxes and arithmetical additions will be involved in a hypothetical noise expression. The bias there is expected to be very small (e.g., only SBoxes would already give the bias $\epsilon(48 \times [x \oplus S(x)]) \approx 2^{-286.4}$), and not enough for mounting a linear attack on SNOW-V.

On the other hand, we have noticed that if we take the pair of taps $(T1, T2)$ from either $(A0, B0)$ or $(A1, B1)$, then 3 consecutive keystream expressions would involve all four 128-bit words of the LFSR, and, moreover, at least 256 bits of them ($A1^{(t)}$ and $A0^{(t)}$) will appear in the keystream expressions only once. E.g., the taps in SNOW-Vi are $T1 = B1$ and $T2 = A1$, which implies: $T1^{(t)} = B1^{(t)}$, $T1^{(t-1)} = B0^{(t)}$, $T2^{(t)} = A1^{(t)}$, $T1^{(t+1)} = A0^{(t)} \oplus f_\beta(B0^{(t)}, B1^{(t)})$.

This means that in SNOW-Vi one has to collect at least 512 bits of the keystream in order to have there *some* nonzero bias. That bias is expected to be even smaller than that in SNOW-V since it would involve even more SBoxes and arithmetical additions.

From initialisation analysis perspectives. Now when we discovered that a new tap position would suggest a strengthened security from the linear analysis arguments, we then started to look on what would be the most promising combination, by trying all possible variants and performing a brief MDM test for each of them.

Table 1: Number of nonrandom initialisation rounds (out of 16). MDM test with cube size=3. The mixing effect is better for smaller values.

Taps		#non-random
T1	T2	rounds
A1	B1	5.69-6.21
B1	A1	5.43-5.75
A0	B0	6.25-7.24
B0	A0	9.16-10.8

In Table 1, for each variant of tap positions, we have received the range of non-random initialisation rounds (the range also depends on the key/iv-loading scheme). The smaller the values the better the mixing effect. A good mixing effect also contributes to a better mixing during the keystream generation phase. The obvious choice was to pick the variant $(B1, A1)$ for SNOW-Vi, while keeping key/iv-loading scheme unchanged.

From implementation perspectives. In addition to other implementation tricks, the new tap position $T2 = A1$ makes it possible to first update the LFSR once, then update the FSM twice, since then the two consecutive values of $T1$ and $T2$ become directly available in the content of the LFSR.

3.2 Properties of the LFSR

The new LFSR has a maximum cycle length of $2^{512} - 1$, that can be verified by the same methods as in [EJMY19, CDM20]. The characteristic polynomial is primitive and has 209 terms: $m(x) = \sum_{i=1}^{|T|} x^{T_i}$, where:

$$\begin{aligned} T = [&512, 496, 488, 480, 472, 462, 455, 448, 444, 439, 438, 437, 430, 426, 422, 421, 420, 419, 414,\\ &412, 408, 404, 403, 402, 401, 399, 398, 394, 392, 390, 387, 386, 385, 384, 382, 381, 380, 373,\\ &371, 369, 367, 366, 359, 358, 353, 351, 350, 349, 347, 346, 341, 340, 339, 336, 335, 334, 333,\\ &329, 319, 318, 317, 316, 314, 313, 312, 311, 310, 309, 305, 304, 303, 302, 301, 300, 298, 297,\\ &296, 295, 291, 290, 289, 287, 281, 280, 278, 277, 276, 275, 273, 272, 270, 267, 266, 263, 262,\\ &261, 258, 257, 254, 252, 246, 245, 243, 235, 233, 231, 229, 228, 226, 225, 224, 223, 222, 221,\\ &220, 219, 218, 216, 215, 214, 212, 211, 210, 207, 201, 199, 198, 197, 196, 195, 194, 192, 191,\\ &190, 189, 185, 184, 181, 179, 175, 173, 170, 169, 168, 166, 160, 158, 156, 155, 152, 147, 146,\\ &145, 143, 140, 137, 136, 134, 133, 132, 131, 128, 127, 125, 116, 111, 109, 108, 105, 103, 101,\\ &100, 99, 98, 95, 94, 90, 86, 84, 82, 80, 79, 75, 74, 73, 70, 69, 68, 67, 57, 55, 52, 51, 50, 49, 48,\\ &45, 43, 42, 36, 32, 23, 22, 21, 16, 14, 8, 7, 0]. \end{aligned}$$

Field polynomials both have weight 8 (excluding x^{16}), so that if a reduction happens then exactly half of the 16 bits will be flipped. Additionally, the base fields were selected such that they have exactly 4 coinciding bits, 4 bits where flip not happening, and two 4-bit sets where only one of the two fields flip the bits.

3.3 Linear attacks

Assume that α and β are 16×16 binary matrices that represent multiplication in corresponding fields, then:

$$\begin{aligned} \beta a^{(t+16)} &= \beta b^{(t)} + \beta\alpha a^{(t)} + \beta a^{(t+7)},\\ a^{(t+24)} &= b^{(t+8)} + \alpha a^{(t+8)} + a^{(t+15)},\\ a^{(t+32)} &= \underbrace{b^{(t+16)}}_{=a^{(t)}} + \alpha a^{(t+16)} + a^{(t+23)}, \end{aligned}$$

from where we derive the recurrence for a-terms in SNOW-Vi as:

$$\begin{aligned} 0 &= (x^{16} + x^8 + \beta)(x^{16} + x^7 + \alpha) + 1\\ &= x^{32} + x^{24} + x^{23} + (\alpha + \beta)x^{16} + x^{15} + \alpha x^8 + \beta x^7 + (1 + \beta\alpha), \end{aligned}$$

to be compared with the feedback recurrence in SNOW-V:

$$0 = (x^{16} + \alpha^{-1}x^8 + x^1 + \alpha)(x^{16} + \beta^{-1}x^8 + x^3 + \beta) + 1.$$

I.e., in SNOW-Vi we have 8-weight recurrence and in SNOW-V it is 12-weight recurrence.

For standard linear distinguishing and correlation attacks one has to find a multiple of the above recurrence of weight 3 or 4. Thus, we believe that 8 is also good enough to be resistant against linear cryptanalysis. Since the FSM is not changed, the complexity of a linear distinguishing attack remains around $O(2^{645})$ for a 3-weight multiple, see [EJMY19] for details.

3.4 Attacks on the initialisation

As done for SNOW-V, we use maximum degree monomial (MDM) test and cube attack based on division property to check if the Key and IV bits are fully mixed after the initialisation.

3.4.1 MDM tests

In a MDM test, each output keystream bit is regarded as a random Boolean function of the Key and IV bits, and the MDM coefficient in the algebraic normal form (ANF) of the Boolean function should follow a random uniform distribution between $\{0, 1\}$. However, in the initial few rounds of the initialisation, the mixing effect is not enough and the MDMs of the corresponding Boolean functions are much more likely to be zero than one, thus resulting into a zero sequence before they become random-like. The MDM test checks how long this zero sequence persists throughout the full initialisation rounds. As done for SNOW-V, we start with a relatively small size of cube set under which the randomness result deviates the most from the expected value (i.e., the longest zero sequence) and gradually increase to a 24-bit set. For SNOW-V, we start with the worst 3-bit cube set, while for SNOW-Vi, we try to be stricter and start with the worse 4-bit cube.

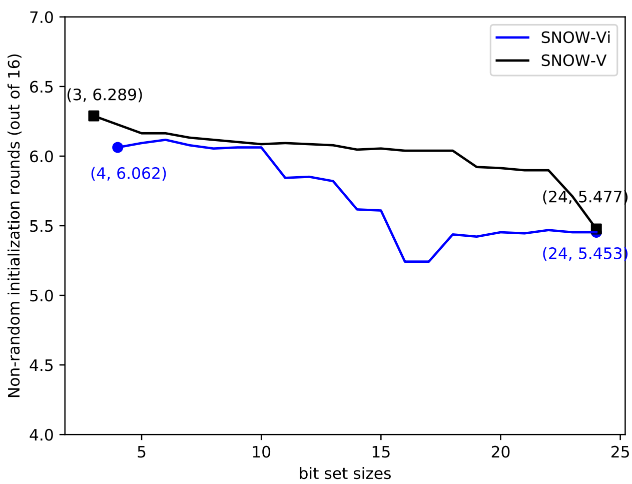

Figure 2: The number of rounds failing the MDM test.

Figure 2 shows the numbers of rounds failing the MDM test under different cube sizes compared to SNOW-V. From the result, one can directly see that the randomness of the initialisation output of SNOW-Vi is better than SNOW-V. Specifically, the randomness under the worst cube of size four in SNOW-Vi is even better than that for SNOW-V of the worst cube of size three. The difference might be larger if the worst cube set of a certain larger size is explored instead of greedily adding the worst one bit to the existing subset. However, this is computationally demanding. Next, we use a more fine-grained way using division property to check the initialisation.

3.4.2 Cube attacks based on division property

Cube attacks based on division property evaluate the set of involved key bits J in the superpoly given a certain cube I, and recover the superpoly if feasible. The propagation rules of division property for different operations in a cipher can be modelled by some (in)equalities of a MILP(Mixed Integer Linear Programming) problem. By solving the MILP problem using some optimisation tools, one can get the involved key bits and the upper bound of the algebraic degree d of the superpoly; the larger $|J|$ and d are, the better the mixing effect is. The time complexity for recovering the superpoly is given by $2^{|I|} \times \binom{|J|}{\leq d}$.

Table 2: Cube attacks on reduced-rounds of SNOW-Vi ($|I|, d, |J|$, and C denote the cube size, the degree, the number of involved key bits, and attacking complexity, respectively).

Rounds	3		4		≥ 5	
Ciphers	SNOW-Vi	SNOW-V	SNOW-Vi	SNOW-V	SNOW-Vi	SNOW-V
$\|I\|$	4	15	128	40	128	128
d	28	17	242	145	256	256
$\|J\|$	100	131	256	256	256	256
C	$2^{86.7}$	$2^{84.9}$	$> 2^{256}$	$> 2^{256}$	$> 2^{256}$	$> 2^{256}$

The MILP model of SNOW-Vi is generally similar with that for SNOW-V, given in Algorithm 5 in [EJMY19]; while only the modelling for the update of the LFSR should be modified. We tried different cubes and tested the involved key bits and the maximum degrees of the corresponding superpolies under different rounds. The results are presented in Table 2 and one can see that the mixing effect of SNOW-Vi is better than SNOW-V. Specifically, after four rounds, for a cube size 40 in SNOW-V, all key bits are involved and the maximum degree is 145. When the cube size goes larger, the number of involved key bits and degree would both reduce. However, in SNOW-Vi, for the cube of all IV bits, all key bits are involved, and the maximum degree is 242. This can be expected since when $T2$ is moved to the higher part of LFSR-A, the new update results of IV and key bits are immediately fed to the FSM, making the mixing faster. After five rounds, all key bits and IV bits are fully mixed just like SNOW-V. The results match well with the results from the MDM test.

3.5 Algebraic attacks

In algebraic attacks one expresses the cipher output as algebraic equations over the unknown key (or state) bits, and try to solve the resulting system of nonlinear equations. The only source of non-linearity during a normal cipher update iteration of SNOW-Vi is the FSM, and that is unchanged from SNOW-V. In the algebraic attack analysis of SNOW-V in [CDM20], the authors make use of the fact that the tap values $T1^{(t)}$ and $T2^{(t)}$ are linear combinations of the first values $T1^{(-1)}, T1^{(0)}, T2^{(-1)}, T2^{(0)}$ and each iteration of the cipher can be written as

$$\begin{aligned}
T1^{(t+1)} &= Lin_\beta(T1^{(t-1)}, T1^{(t)}, T2^{(t-1)}, T2^{(t)}),\\
T2^{(t+1)} &= Lin_\alpha(T1^{(t-1)}, T1^{(t)}, T2^{(t-1)}, T2^{(t)}),\\
R1^{(t+1)} &= \sigma(R2^{(t)} \boxplus_{32} (R3^{(t)} \oplus T2^{(t)}),\\
R2^{(t+1)} &= AESr(R1^{(t)}),\\
R3^{(t+1)} &= AESr(R2^{(t)}),\\
z^{(t+1)} &= (R1^{(t)} \boxplus_{32} T1^{(t)}) \oplus R2^{(t)}.
\end{aligned}$$

We can see that these equations are still valid in SNOW-Vi. Following the arguments in [CDM20] we note that the linear parts of the cipher can be "*effectively disregarded when determining the number of nonlinear equations and the number of associated variables*". Hence the proposed change in linear update functions for $T1$ and $T2$ does not affect the complexity of mounting an algebraic attack using quadratic (or higher degree) equations. The conclusion is that both linearisation methods and Gröbner basis algorithms remain unfeasible for algebraic attacks on SNOW-Vi.

3.6 Guess-and-determine attacks

In guess-and-determine attacks one guesses part of the state and from the keystream equations one determines other parts of the state. One guesses as few bits as possible and then determines as many as possible through given equations. For the case of SNOW-Vi the situation is very similar to SNOW-V. The equation $z(t) = (R1(t) \boxplus_{32} T_1(t)) \oplus R2(t)$ involves three unknowns, each of size 128 bits. In order to determine state bits, one then has to guess 256 bits. Looking at the equation for the next output block, it requires guessing another 128 bits. This illustrates that a guess-and-determine attack on SNOW-Vi is still of large complexity.

A straightforward guess-and-determine attack is given in [CDM20], which requires guessing 512 bits within three consecutive keystream output words to recover the full 896 state bits. The attack there applies to SNOW-Vi exactly the same. Thus we could first get an upper bound on the complexity of the guess-and-determine attack against SNOW-Vi, which is 2^{512}.

In January 2020, Jiao *et al* in [JLH20] gave a byte-based guess-and-detemine attack against SNOW-V with complexity 2^{406} within seven keystream blocks. In their attack, the registers in LFSR and FSM are split into bytes and the update operations are correspondingly transformed with some carriers introduced. The attack first presets an initial guessing set and run some algorithm to explore guessing paths and thus driving a guessing basis. This process is repeated several times to remove possible redundant bytes. Though the details of the guess-and-determine attacks against SNOW-V and SNOW-Vi under their attack would be different, the general guessing route could be the same.

The final initial guessing set used in [JLH20] has 24 byte variables, and these variables are all from the FSM registers or the higher halves of the LFSR registers, while the variables which are tapped are not used. Thus we could use the same initial guessing set and have similar guessing path. During the guessing process, 12 more bytes from the FSM registers and 13 more bytes from LFSR are guessed. Since there are three taps for LFSR-A and LFSR-B in SNOW-V while two in SNOW-Vi, we make the worst assumption that when 13 bytes are required for guessing in SNOW-V, only around eight bytes are needed in SNOW-Vi. In this case, there are still $24 + 12 + 8 = 44$ bytes, which are 352 bits. Besides this, some carriers must be guessed. Thus the complexity of the guess-and-determine attack against SNOW-Vi is larger than 2^{352}. We can make an even worse assumption that the guessed variables in LFSR can be freely derived, resulting in guessing $24 + 12 = 36$ bytes, i.e., 288 bits, for which the complexity is still larger than 2^{288}.

3.7 Other analyses

From studying [CDM20], we note that most of the results received for SNOW-V are not affected by the new LFSR: the transfer of key entropy (Section 2.1), the injectiveness of initialisation (Section 2.4), time-memory-data trade-off attacks (Section 6), related Key-IV attacks (Section 7), side-channel attacks (Section 8), AEAD mode (Section 9). In fact, even derivations in Section 3.1 on correlation attacks remain true for SNOW-Vi, while in this paper we reconsider some other aspects of linear analysis.

Hardware evaluations. We expect minor changes in hardware compared to SNOW-V (see, e.g., [CBB20]). Our assessment is that the performance results should not be affected at all, since the critical path is actually in the FSM that is unchanged. The area size and the energy consumption in SNOW-Vi should be slightly better (i.e., lower) than in SNOW-V, since the new LFSR has a reduced number of gates for its feedback update function, and therefore also consumes less power.

4 Software evaluation

Performance of SNOW-Vi heavily depends on the ability to reduce the number of instructions, as well as careful consideration of hardware peculiarities, such as CPU interleaving capabilities, use of registers, instructions latency and throughput characteristics. In this section we analyse SNOW-Vi from the software point of view, consider different implementation techniques and various target platforms.

4.1 Implementations and notations

Algorithms. We have done a number of different implementations in C/C++ of the two ciphers, SNOW-V and SNOW-Vi, that we can use for relative comparison on various platforms. We also used OpenSSL tools on test targets to measure the performance of AES-256-CTR, for comparison purposes.

Registers. In both SNOW-V and SNOW-Vi we have implementations that utilise: only 128-bits registers (e.g., XMM on Intel platforms), and up to 256-bit registers (e.g., YMM). ARM NEON only supports 128-bits registers.

Instruction sets. We have implementation version with different restrictions in instruction sets. For Intel platforms, we start with the most restricted SSE4.1 set and then add more capabilities as we try implementations utilising AVX2 and also AVX-512. For ARM platforms, we only have the NEON instruction set. All implementations and platforms uses an AES round function instruction. We present C/C++ versions using Intel intrinsics below, but it's relatively straight forward to convert to NEON, and our test code will be shared upon request.

Code generation. In SSE-type of code generation the CPU can only handle instructions of the form $x = x + y$, thus changing the value of one of the input registers to hold the result. In AVX-type of code generation CPU instructions can have 3 arguments, i.e. $x = y + z$, thus preserving the values of the input registers.

Unrolled versions. By design, both SNOW algorithms that we use in the performance tests would simply have bulk encryption in a loop that process 16 bytes in each step (if we ignore unaligned bytes). That is the same situation as with AES-256-CTR. These implementations we call for 1-unrolled versions. However, there might be a performance gain if each step of such an encryption loop would process 4×16 bytes, instead, and where the Key/IV initialisation is also partly or fully unrolled. These versions we call 4-unrolled implementations.

Notation. We adopt the following notation to indicate a specific case that we were testing: [**Alg/Unroll/Regs-Inst**], where: **Alg** is the algorithm name – {SNOW-V, SNOW-Vi, AES-256-CTR}; **Unroll** determines if the implementation is a plain one or unrolls 4 16-bytes blocks in the encryption loop – {1, 4}; **Regs** determines the maximum size of the registers being used – {128, 256, 512}; **Inst** determines the type of code generation and the maximum instruction sets being used – {SSE, AVX, AVX2, AVX512, NEON}. For 128-SSE case we use up to SSE4.1 instructions.

Examples: SNOW-Vi/1/128-SSE, SNOW-V/4/256-AVX512.

4.2 New test environment

The test environment that we adopted in [EJMY19] was using Windows-specific calls and thus it was not portable. In order to perform a much wider software evaluation on various platforms we decided to make a new and more generic test environment where we utilise the standard C function `time(NULL)`. The granularity of `time()` is 1 second, so that before each test we are waiting for the start of a "fresh" second, then in the loop we are waiting for the start of the next second, while performing a lot of encryptions with a selected algorithm in a loop and counting the number of encryptions processed. This, of course, has some impact on the received performance numbers. We, however, tried to balance it by calling the function `time()` only after 1024 encryptions. The total count is still magnitudes higher so this approach should not affect the accuracy of the measurements, but partly reduces the impact of the system calls of `time()`.

Table 3: New test environment, previous and new benchmarks.

Encryption speed (Gbps)	Plaintext length				
	16384	4096	1024	256	64
P1(a): Work laptop, Win10, Intel Core i7-8650U @ 4.2GHz / AVX2					
	Previous benchmarks from [EJMY19]				
AES-256-CTR/OpenSSL 1.1.1j	35.06	34.16	30.95	22.67	11.32
SNOW-V (C++)	58.25	54.60	45.28	26.37	9.85
	New code in the new test environment				
SNOW-V/1/256-AVX2	56.10	52.28	44.05	26.10	9.94
SNOW-Vi/1/256-AVX2	**77.04**	71.54	57.95	33.01	12.25

In Table 3 we present the previous results from [EJMY19] and the new results under the new benchmarking system, so that a relative comparison can be made. The new variant SNOW-Vi reaches the speed of 77 Gbps on that particular platform.

4.3 Impact of unrolling and code generations

Table 4: Impact of unrolling and SSE/AVX instruction encodings with 128-bit code.

Encryption speed (Gbps)	Plaintext length				
	16384	4096	1024	256	64
P1(b): Work laptop, Win10, Intel Core i7-8650U @ 4.2GHz / AVX2					
SNOW-Vi/1/128-SSE	55.16	52.14	43.52	26.11	10.04
SNOW-Vi/4/128-SSE	68.85	65.93	55.42	33.60	13.12
SNOW-Vi/1/128-AVX	62.28	58.82	50.31	30.93	12.12
SNOW-Vi/4/128-AVX	70.33	66.71	56.59	34.36	13.31

In Table 4 we demonstrate the difference between a "usual" and "unrolled" implementations with basically the same 128-bit friendly code for SNOW-Vi. There we can see a significant speedup when unrolling loops in SSE-type of code generation.

4.4 Performance results

In Table 5 we provide the reader with more performance benchmarks on a number of other platforms and for various use cases.

Table 5: Performance measurements on various platforms.

Encryption speed (Gbps)	Plaintext length				
	16384	4096	1024	256	64
P1: Work laptop, Win10, Intel Core i7-8650U @ 4.2GHz / AVX2 (speed up +37%)					
AES-256-CTR/OpenSSL 1.1.1j	35.06	34.16	30.95	22.67	11.32
SNOW-V/1/256-AVX2	56.10	52.28	44.05	26.10	9.94
SNOW-Vi/1/256-AVX2	77.04	71.54	57.95	33.01	12.25
P2: Home laptop, Win10, Intel Core i7-1065 G7 @ 3.9GHz / AVX512 (+58%)					
AES-256-CTR/OpenSSL 3.0.0	68.09	66.07	57.85	38.73	16.42
SNOW-V/4/256-AVX512	58.52	55.57	45.92	27.16	10.33
SNOW-Vi/1/256-AVX512	**92.34**	85.97	69.16	38.60	14.12
P3: Work Station, Ubuntu, AMD Ryzen 5 3600 @ 4.2GHz / AVX2 (+44%)					
AES-256-CTR/OpenSSL 1.1.1f	68.84	67.03	58.35	33.69	18.89
SNOW-V/1/256-AVX2	55.16	51.77	42.45	24.05	8.81
SNOW-Vi/4/128-AVX	79.79	75.77	64.65	40.88	16.56
P4: Remote VM, Ubuntu, Intel Xeon E3-12xx / AVX (+45%)					
AES-256-CTR/OpenSSL 1.1.1	21.57	20.93	19.89	15.81	7.84
SNOW-V/1/128-SSE	22.01	20.87	17.84	11.13	4.37
SNOW-V/4/128-AVX	30.20	28.69	23.71	14.28	5.50
SNOW-Vi/1/128-SSE	33.55	31.57	25.85	16.06	6.18
SNOW-Vi/4/128-AVX	43.75	41.91	35.63	22.25	8.86
P5: Intel NUC7JY, Ubuntu, Intel Pentium Silver J5005 @ 2.8GHz / SSE4.2 (+59%)					
AES-256-CTR/OpenSSL 1.1.1	22.46	21.81	20.12	15.08	7.29
SNOW-V/1/128-SSE	13.56	12.92	10.91	7.14	2.94
SNOW-V/4/128-SSE	16.24	15.23	12.60	7.41	2.82
SNOW-Vi/1/128-SSE	19.06	18.15	15.49	10.57	4.35
SNOW-Vi/4/128-SSE	25.90	24.60	21.05	13.43	5.54
P6: Older laptop, Win7, Intel i7-3540M @ 3GHz / AVX (+40%)					
AES-256-CTR/OpenSSL 1.1.1i	26.33	25.62	23.25	16.77	7.41
SNOW-V/4/128-SSE	33.96	32.01	26.44	15.33	5.73
SNOW-V/4/128-AVX	38.52	36.57	30.30	17.96	6.79
SNOW-Vi/4/128-SSE	51.54	48.96	41.19	25.18	9.87
SNOW-Vi/4/128-AVX	53.96	51.14	43.08	26.19	10.18
P7: Mobile phone, iPhone X, ARM-based A11 Bionic @ 2.39GHz / NEON (+58%)					
AES-256-CTR/OpenSSL 1.1.1i	19.74	19.53	17.86	13.74	8.94
SNOW-V/1/128-NEON	22.25	21.39	18.51	11.72	4.80
SNOW-V/4/128-NEON	24.46	23.54	19.85	12.47	5.19
SNOW-Vi/1/128-NEON	35.42	34.07	29.79	19.18	8.11
SNOW-Vi/4/128-NEON	38.70	37.42	32.69	21.66	10.12
P8: Apple Mini, macOS, ARM-based Apple M1 @ 3.2GHz / NEON (+64%)					
AES-256-CTR/OpenSSL 1.1.1i	58.61	57.44	55.13	45.73	24.97
SNOW-V/1/128-NEON	32.48	30.97	26.47	16.74	6.80
SNOW-V/4/128-NEON	39.06	37.31	31.68	19.78	7.95
SNOW-Vi/1/128-NEON	50.47	48.15	41.21	26.09	10.84
SNOW-Vi/4/128-NEON	64.16	61.10	51.39	31.46	12.78

4.5 Reference implementation

A 128-SSE friendly C/C++ code of SNOW-Vi is given in Listing 1. It is not optimised for performance benchmarking but rather it serves as an "easy-to-read" reference implementation.

```
#define XOR(a, b)      _mm_xor_si128(a, b)
#define AND(a, b)      _mm_and_si128(a, b)
#define ADD(a, b)      _mm_add_epi32(a, b)
#define SET(v)         _mm_set1_epi16((short)v)
#define SLL(a)         _mm_slli_epi16(a, 1)
#define SRA(a)         _mm_srai_epi16(a, 15)
#define TAP7(Hi, Lo)   _mm_alignr_epi8(Hi, Lo, 7 * 2)
#define SIGMA(a)       _mm_shuffle_epi8(a, _mm_set_epi64x(\
                        0x0f0b07030e0a0602ULL, 0x0d0905010c080400ULL));
#define AESR(a, k)     _mm_aesenc_si128(a, k)
#define ZERO()         _mm_setzero_si128()
#define LOAD(src)      _mm_loadu_si128((const __m128i*)(src))
#define STORE(dst, x) _mm_storeu_si128((__m128i*)(dst), x)

struct SnowVi
{
    __m128i A0, A1, B0, B1;  // LFSR
    __m128i R1, R2, R3;      // FSM

    inline __m128i keystream(void)
    {
        // Taps
        __m128i T1 = B1, T2 = A1;

        // LFSR-A/B
        A1 = XOR(XOR(XOR(TAP7(A1,A0),B0),SLL(A0)),AND(SET(0x4a6d),SRA(A0)));
        B1 = XOR(XOR(SLL(B0),A0),XOR(B1,AND(SET(0xcc87),SRA(B0))));
        A0 = T2;
        B0 = T1;

        // Keystream word
        __m128i z = XOR(R2, ADD(R1, T1));

        // FSM Update
        T2 = ADD(XOR(T2, R3), R2);
        R3 = AESR(R2, ZERO());
        R2 = AESR(R1, ZERO());
        R1 = SIGMA(T2);

        return z;
    }

    template<int aead_mode = 0> inline void keyiv_setup(
        const unsigned char * key, const unsigned char * iv)
    {
        B0 = R1 = R2 = R3 = ZERO();
        A0 = LOAD(iv);
        A1 = LOAD(key);
        B1 = LOAD(key + 16);

        if (aead_mode)
            B0 = LOAD("AlexEkd JingThom");
        for (int i = 0; i < 15; ++i)
            A1 = XOR(A1, keystream());

        R1 = XOR(R1, LOAD(key));
        A1 = XOR(A1, keystream());
        R1 = XOR(R1, LOAD(key + 16));
    }
};
```

Listing 1: Reference implementation of SNOW-Vi

4.6 Further optimisations

An even faster implementation can employ other tricks, such as the call of the AES round function with T2 as the round key, thus XORing T2 with R3 "for free". One can also optimise the order of instructions for a better performance on a selected platform, see Listing 2 as an example of such efforts for SSE-type of code generation.

```
#define SnowVi_MMX_ROUND(mode, offset)\
T1 = B1, T2 = A1;\
A1 = XOR(XOR(XOR(TAP7(A1,A0),B0),AND(SRA(A0),SET(0x4a6d))),SLL(A0));\
B1 = XOR(XOR(B1,AND(SRA(B0),SET(0xcc87))),XOR(A0,SLL(B0)));\
A0 = T2; B0 = T1;\
if (mode == 0) A1 = XOR(A1, XOR(ADD(T1, R1), R2));\
else STORE(out + offset, XOR(ADD(T1, R1), XOR(LOAD(in + offset), R2)));\
T2 = ADD(R2, R3);\
R3 = AESR(R2, A1);\
R2 = AESR(R1, ZERO());\
R1 = SIGMA(T2);


// Note: in this implementation the length must be 16-bytes aligned
inline void SnowVi_encdec(int length, u8 * out, u8 * in, u8 * key, u8 * iv)
{
    __m128i A0, A1, B0, B1, R1, R2, R3, T1, T2;

    // Key/IV loading
    B0 = R1 = R2 = ZERO();
    A0 = LOAD(iv);
    R3 = A1 = LOAD(key);
    B1 = LOAD(key + 16);

    // Initialisation
    for (int i = -14; i < 2; ++i)
    {   SnowVi_MMX_ROUND(0, 0);
        if (i < 0) continue;
        R1 = XOR(R1, LOAD(key + i * 16));
    }

    // Bulk encryption
    for (int i = 0; i <= length - 16; i += 16)
    {   SnowVi_MMX_ROUND(1, i);
    }
}
```

Listing 2: A more efficient implementation of SNOW-Vi

A better optimisation may be achieved on the assembly level. At our best try, a single encryption/decryption of a 16-byte block data may be done with as low as 15 assembly instructions by utilising 12 XMM/YMM registers and up to AVX512 instruction sets. In the initialisation loop the main code can be shrunk down to 13 assembly instructions, see Listing 3; however, there we omit 2-3 extra instructions that are usually also needed to organise the loop itself.

```
;Input State:
;  Note: for a 256-bit register the pair of two 128-bit values are (Hi|Lo)
;ymm1 = hi = (B[128..255] | A[128..255])
;ymm2 = lo = (B[0..127] | A[0..127])
;xmm7 = R1
;xmm8 = R2
;xmm9 = R3 xor A[128..255]
;
;Constants & Derivatives:
;ymm5 = (A[0..127] | B[0..127]) = _mm256_permute4x64_epi64(lo, 0x4e)
;ymm4 = _mm256_set_epi64x(0xcc87cc87cc87cc87ULL, 0xcc87cc87cc87cc87ULL,
```

```
;                          0x4a6d4a6d4a6d4a6dULL, 0x4a6d4a6d4a6d4a6dULL);
;xmm10 = _mm_setzero_si128()
;xmm11 = _mm_set_epi64x(0x0f0b07030e0a0602ULL, 0x0d0905010c080400ULL)
;load the mask register k1 with 0x0000ffff and keep it all the way, e.g.:
;   mov        eax, 65535
;   kmovd      k1, eax
;
;Encryption/Decryption Loop for one 16-byte block:
1.  vmovdqu    ymm3, ymm1            ; save new lo = hi
2.  vpsraw     ymm6, ymm2, 15        ; arithmetical shift to the right by 15
3.  vpternlogd ymm6, ymm4, ymm5, 106 ; ymm6 = (ymm6 & ymm4) ^ ymm5
4.  vpalignr   ymm1 {k1}, ymm1, ymm2, 14; upper half of ymm1 is preserved
5.  vpsllw     ymm2, ymm2, 1         ; logical shift to the left by 1
6.  vpternlogd ymm1, ymm2, ymm6, 150 ; ymm1 = ymm1 ^ ymm2 ^ ymm6
7.  vmovdqu    xmm2, XMMWORD PTR[r8+rdx]; load xmm2 <= [in + i * 16]
8.  vpermq     ymm5, ymm3, 78        ; swap high/low halves of hi
9.  vpaddd     xmm12, xmm7, xmm5     ; xmm12 = R1 + T1, zeroize upper half
10. vpternlogd ymm2, ymm8, ymm12, 150; xmm2 = in ^ R2 ^ (R1 + T1)
11. vpaddd     xmm12, xmm8, xmm9     ; xmm12 = R2 + (R3^T2)
12. vaesenc    xmm9, xmm8, xmm1      ; R3' ^ T2' = AESR(R2, A'[128..255])
13. vaesenc    xmm8, xmm7, xmm10     ; R2' = AESR(R1, 0)
14. vmovdqu    XMMWORD PTR[rdx], xmm2; store [out + i * 16] <= xmm2
15. vpshufb    xmm7, xmm12, xmm11    ; R1' = Sigma(xmm12)

;Output State: same registers as inputs, except that the new ymm2 is now
    actually ymm3. One solution could be to add vmovdqu ymm2, ymm3; but a
    better way is to call the above code with swapped registers xmm2/ymm2
    and xmm3/ymm3. I.e., a 2-unrolled loop would be more efficient.

;Initialisation Loop: remove steps 7 and 14, and in the step 10 change ymm2
    to ymm1 (=hi). In the last 2 rounds one should XOR the Key to the
    register xmm7 (=R1).
```

Listing 3: Sketch for an assembly implementation.

Implementation tricks. The presented sketch of an assembly code has just a single 256-bit "swap" instruction `vpermq` (step 8) and none of `vextractf128` for extracting the taps, thus saving CPU latency since these instructions are costly. There is only one register copy `vmovdqu` (step 1), that we believe is the minimum and unavoidable. We use one of the AES round calls (step 12) with the next clock's value of the tap T2 as the "round key", thus we can skip one XOR instruction (`R3 xor T2`) during the next clock. We also efficiently utilise the fact that XMM/YMM registers are shared (step 10 in the initialisation loop) and we use AVX512's mask register `k1` (step 4) to avoid an extra `vpblendd`[2]. The above code adopts AVX512's ternary logic `vpternlogd` (steps 3, 6, 10) that effectively removes 3 extra instructions if we would do these steps with AVX2 set, instead. We can avoid the ending register copy (`vmovdqu ymm2, ymm3`) by implementing 2x-unrolled loops. The above 15 assembly steps demonstrate all these tricks.

Nevertheless, we would like to note that the smallest number of assembly instructions does not always mean the fastest speed in reality, due to there are other things to take care about such as instructions interleaving and stitching techniques. For example, one could utilise more than 12 register to convey a better instructions stitching and thus achieving a higher performance.

[2] One may also try to use SSE-legacy instruction in step 4: `palignr xmm1, xmm2, 14` – that would modify the lower half of `ymm1` while preserving it's upper half, as we actually want here; however, we are not sure about AVX-SSE switch penalty, and if there is any in case of assembly coding. This can be studied further.

5 Conclusions

In this paper we present a slightly modified version of the SNOW-V stream cipher called SNOW-Vi. The purpose of this change is to better accommodate a fast implementation in software on lower grade CPUs which only supports 128 bit wide SIMD registers. The only change made, is a small modification to the linear update functions of the taps $T1$ and $T2$. We thoroughly investigate the security implications of this change and go through all previously known analyses of SNOW-V, applying the changes to these security results. The conclusion is that the high security provided by SNOW-V is still intact, and in some cases even improved. Furthermore, we provide a more detailed software performance, comparing SNOW-Vi to both SNOW-V and AES-256-CTR on various CPU architectures. The results show that SNOW-Vi is significantly faster than SNOW-V on all platforms.

References

[CBB20] Andrea Caforio, Fatih Balli, and Subhadeep Banik. Melting SNOW-V: improved lightweight architectures. *Journal of Cryptographic Engineering*, 4 December 2020. https://doi.org/10.1007/s13389-020-00251-6.

[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/Inbox/LSs_from_external_bodies/SAGE/SAGE-20-12LStoSA3onindependentevaluationofSNOWV.zip.

[EJMY19] Patrik Ekdahl, Thomas Johansson, Alexander Maximov, and Jing Yang. A new SNOW stream cipher called SNOW-V. *IACR Transactions on Symmetric Cryptology*, 2019(3):1–42, Sep. 2019. https://tosc.iacr.org/index.php/ToSC/article/view/8356.

[JLH20] Lin Jiao, Yongqiang Li, and Yonglin Hao. A Guess-And-Determine Attack On SNOW-V Stream Cipher. *The Computer Journal*, 63(12):1789–1812, 03 2020. https://doi.org/10.1093/comjnl/bxaa003.

A Test vectors

```
== SNOW-Vi test vectors #1:
key =  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
       00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
iv  =  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Initialisation phase, z =
       00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
       63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63
       a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5
       4f 4f 4f 4f 4f 4f 4f 4f 4f 4f 4f 4f 4f 4f 4f 4f
       7a 5b 5a 5a 79 5b 5a 5a 5a 5a 5a 5a 5b 5a 5a 5a
       4d 51 be 6e 19 0a 0a a9 a4 fe fe ae f4 d6 d6 d6
       7a 97 fb 47 d2 57 62 46 8a ca df 1e d1 48 4d 3c
       8c 97 87 3e 90 00 38 d5 2d f3 46 c3 2f f7 97 0c
       10 89 37 a1 02 46 61 0a 67 07 b5 4e 94 1e 0e 3b
       94 36 b9 e3 3b 0f 10 9a dc 89 b3 d5 a3 ae f8 2d
       ba ea 9f d0 68 b9 a1 1e 43 62 67 f8 7f 4a 05 ac
       0c 15 12 c2 38 80 09 46 5a 55 ef f8 89 81 6c 97
       75 82 9e c8 a8 73 70 38 cd 5e c5 7e 21 9d 98 16
       ed 45 92 3c 43 7a d7 b0 e5 22 61 72 85 47 dc be
       e9 38 ac 0b 70 5c b9 85 2a 42 49 ba 0e 87 37 c3
       65 28 2c ef ab 7c a9 57 ae f8 d9 4e 29 38 c8 cd
Keystream phase, z =
       50 17 19 e1 75 e4 9f b7 41 ba bf 6b a5 de 60 fe
       cd a8 b3 4d 7e c4 c6 42 97 55 c1 9d 2f 67 18 71
       89 57 d3 26 cb 46 50 2c eb 81 4c cd 6e a5 3a ae
       dd 6c 92 fb f3 92 1e 8b d7 31 7b e2 20 15 31 bb
       09 3e e8 72 e9 eb 40 34 e9 b7 1a 4a c2 b5 4b d9
       f0 0f 5a dc 06 d2 e6 b5 9f b7 5a 01 be f6 13 14
       1c 8a b2 02 ee 38 e2 85 0c ca 60 6a b8 75 cd 12
       41 03 b3 2f a5 14 5d df 54 e7 a0 7b 0f 3e b7 7a


== SNOW-Vi test vectors #2:
key =  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
       ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
iv  =  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
Initialisation phase, z =
       ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
       9c 9c 9c 9c 9c 9c 9c 9c 9c 9c 9c 9c 9c 9c 9c 9c
       cf 09 cc 09 d6 10 d7 10 cc 36 cd 36 d6 10 d7 10
       e5 31 72 b0 e8 91 53 dd 75 b0 3e 31 54 dd d2 91
       22 b3 31 da e5 05 d7 91 66 7b 7d fb 3f 84 a3 ff
       cd d6 c9 02 9e 24 76 3a 19 82 bc 3c 79 d1 9d 62
       e1 a3 fb ac ea 2b 6d 68 a1 a7 51 04 3a 46 0b db
       b2 30 52 68 82 4b 88 09 ac 92 d5 7d 00 7e ad 0c
       79 74 7c eb 01 95 02 a7 1a 2f f5 07 7c 89 96 ad
       a1 06 eb d4 c1 d8 5f 12 61 81 e1 a9 55 1b 3b df
       aa 5d ff 5a 66 a3 67 16 f7 dc c2 ec 3f da 64 3d
       ad 4d ee 83 27 29 15 0a 3e f3 3c 9e d5 79 d9 79
       50 a4 a0 dd 21 a0 1c 40 68 31 e6 2e 9d 38 ef 0d
       d3 3c c5 72 1b 4d fa 2f 2c cd c9 1f b1 73 fb f3
       e8 d0 e3 f1 14 e3 2a 20 ff 56 df 09 7c ab f8 04
       1e 24 ae 32 56 9f 7b 08 82 30 4d 80 37 cb 23 b2
Keystream phase, z =
       18 71 53 c0 88 1d 00 e8 bf a0 e2 fa fe 71 5e a3
       8d e7 fd 87 a6 76 17 1c a1 5e 47 5b 4d a7 b8 7d
       ad 86 fc fd 9e 0f bb be ef 6a f4 5f 39 29 c1 23
       9b f3 e5 ef b7 d6 90 e6 9d 60 7d c5 c0 4f f4 77
       4c 9f 06 a2 b6 36 3e 52 fc b3 0b 8f d3 9f e7 6e
       11 64 a6 bd a4 73 4a 76 ee 5f e9 ff 28 ff c1 39
       f9 c6 f1 7d 48 43 0c 18 df 3c f4 5d 23 5e dc b3
       f6 d4 d1 0b f6 75 f4 ac c4 fb b0 88 cc 5e c4 90
```

```
== SNOW-Vi test vectors #3:
key =  50 51 52 53 54 55 56 57 58 59 5a 5b 5c 5d 5e 5f
       0a 1a 2a 3a 4a 5a 6a 7a 8a 9a aa ba ca da ea fa
iv  =  01 23 45 67 89 ab cd ef fe dc ba 98 76 54 32 10
Initialisation phase, z =
       0a 1a 2a 3a 4a 5a 6a 7a 8a 9a aa ba ca da ea fa
       38 ee a4 da 77 24 62 90 a5 ff 09 e3 6c 85 50 29
       c3 62 78 ce 97 43 29 97 7e b0 df 7c 2e 5b 9b a2
       ea ba cd 10 4a 5f 1d dd 71 58 96 16 11 e9 59 6e
       98 e8 c1 c4 30 18 9d f2 97 f0 0d ce 37 a1 69 bc
       d9 82 ee 9c db 03 04 cc 23 22 5e d1 8b dc ae ab
       30 00 67 12 44 dd 55 52 12 f4 ae 68 a0 da a3 d0
       87 48 b7 ac f4 67 00 37 ce 67 a7 42 71 4e e1 18
       91 27 9b f8 ca 8e a1 2d 82 6b 6c f7 b7 ef a9 ce
       b4 f0 16 c9 9d d9 7a 3e 76 30 71 f0 99 24 01 a7
       24 aa b3 0e d4 fc cf e8 41 8a c5 74 8f 53 c4 47
       14 7b fa 54 f5 2f ad 01 ab 96 d6 cc da 01 ee 86
       23 fd d5 4f 2b 8d d6 0d 6c d0 b3 de da 70 42 e1
       0c 73 a0 0f e2 87 78 1f 5c 1b 92 0c 00 16 b8 0c
       b1 49 b2 9c df da 0c 95 b9 d3 18 96 91 81 a2 ec
       ea ba d3 84 90 c8 cf b6 a1 f5 80 e0 6f d7 74 33
Keystream phase, z =
       3a 40 f5 40 f5 47 f0 0f 2d 6f e3 d0 01 c1 40 3a
       c7 05 9a 39 19 78 4f ab 41 4b be f7 59 25 e5 23
       7e 12 45 4a ea 9e 01 1c e4 46 29 ad f3 f7 a8 bb
       7e 26 bd 6c 42 95 ce 62 6a 70 b6 4b 41 48 f7 b3
       b4 e2 33 57 5a f9 ba 7a 76 34 a6 bb 22 c7 40 77
       3e be eb ed 5a 94 94 d5 3a 2b 95 86 03 0d 68 7d
       28 f9 7e c9 83 fd 76 41 3e d6 55 1b df 89 f1 eb
       30 c2 4d 1c 61 2d 5a 93 14 d7 64 d8 22 7e 4d bf
```

Listing 4: Test vectors for SNOW-Vi.