Introducing the ALF family: AES-NI-based length- and format-preserving encryption

Dachao Wang[1], Alexander Maximov[2] and Thomas Johansson[1]

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

Abstract. This paper introduces the ALF cipher family, designed for format-preserving encryption. The key strategy is to leverage AES-NI instructions to achieve a high software performance while also providing 128-bit security.
A central part of the paper is the design of an AES-related bit-oriented block cipher called ALF-n-t, with a block size ranging from 16 to 127 bits. The core of the design is byte-oriented, but allows appending up to 7 bits to a byte-oriented input. We introduce an approach for efficient implementation using AES-NI, even though the block size is not 128 bits. The block cipher family is turned into a format-preserving encryption by a new cycle-sliding transformation, improving slightly on traditional cycle-walking. Performance evaluation shows speed improvements compared to the standardised algorithm FF1 and the previous FF3.
As the input size may vary a lot between different cases, we also include the description of other members of the ALF family that cover different domain sizes. While these designs differ, they share a common theme: the use of AES-NI instructions.

Keywords: format-preserving encryption, AES-NI, cryptography

1 Introduction

Block cipher encryption through standards like AES provides an efficient and secure method for ensuring data confidentiality. In traditional block ciphers, the ciphertext occupies the same space and structure as the original plaintext, both identified as bit vectors of the same length. While this is suitable for binary data, it can become limiting when dealing with structured, non-binary data such as credit card numbers, mobile phone numbers, or identity identifiers in communication systems. In mobile communication systems, data formats and field sizes are often tightly constrained.

One of the solutions for these specific scenarios is variable-input-length ciphers [BR99], also called length-preserving encryption in some works [WFW05, CHB21]. Such ciphers map a key and a plaintext into a ciphertext that has the same length as the plaintext. In more restricted applications, Format-Preserving Encryption (FPE) was introduced to allow encryption of data while maintaining its original format and length in the ciphertext [BRRS09]. A common example would be a 16-digit credit card number encrypted into another 16-digit ciphertext number. The point is to preserve compatibility with existing systems and user interfaces. This can be an essential feature for many applications where space and input validation are critical.

An FPE takes a key and a message as input, as well as a specification of the format. In standard FPE, the format consists of a radix q (alphabet size) and a length N. The domain for the cipher then has cardinality $Q = q^N$. However, we can also consider the

more general case, as we do in this paper, where we consider the possibility of a specific modulus q_i for position i, giving a domain size Q of the form $Q = q_0 \cdot q_1 \cdot \ldots \cdot q_{N-1}$.

For details on the historic development of FPEs, we refer to the introduction of [DHHV21]. Two common FPEs based on Feistel networks have been standardised as FF1 and FF3 [BRRS09, BRS10]. Bellare et al. found a weakness in FF1 and FF3 [BHT16] and FF3 was subsequently broken by Durak and Vaudenay [DV17]. This was later improved by Hoang et al. in [HTT18, HMT19]. In 2025, NIST removed FF3-1 from the SP 800-38G Revision 1, recommending only the use of FF1 [Dwo25].

There also exist constructs based on substitution-permutation networks (SPN) [BSV07, GLP06], but they typically lack flexibility, e.g only designed to work on blocks of decimal digits, or restricted in other ways. The currently most promising and flexible SPN-based FPE might be FAST, an FPE scheme recently presented in [DHHV21].

One of the challenges in designing FPE schemes is that the size of the domain can vary a lot. For small domains, one can assume that the adversary knows the entire codebook. Thus, FPE schemes are typically strengthened using a tweak, as in the direction of tweakable block ciphers [LRW11]. FPE schemes for small domains versus large domains may look very different, and a particular construct may apply only in a restricted range.

Many FPE schemes are commercially developed and linked either to patents or at least use implementations that are not openly available for performance comparison [Bus, Piv, MSPb, MSPa, DHHV21]. Also, many algorithms, like FF1, do not have a good performance. There seems to be a general interest in industry for new secure FPE schemes that provide very good performance together with open implementations and free algorithms. The goal of this paper is to provide exactly this for any choice of domain size.

A key idea for our work is to use the AES-NI CPU instructions to achieve very fast FPE algorithms. AES-NI is widely available on modern CPUs and is designed to be efficient in order to support AES-based cryptographic algorithms, including not only AES itself but also other modern designs such as SMAC [WMEJ25], Rocca [SLN+21], etc., that achieve a very high speed due to the use of AES-NI. However, when it comes to domains smaller than 2^{128}, AES-NI is not directly suitable. The well-known cycle-walking method can in some cases be used to turn AES-NI into operations on such domain, while practical issues may occur, e.g., giving a runtime of a high variance.

In our work, we will focus on AES-NI-based solutions, also getting support from other SIMD instructions such as XOR, etc. We target for practical solutions for FPE, where the software performance is one of the major factors. Our approach aims to provide an efficient alternative to FF1/FF3 algorithms, while also expanding the variety of use cases.

In this paper we propose a family of **A**ES-round based **L**ength- and **F**ormat-preserving encryption algorithms, collectively called ALF.

The main algorithm is referred to as ALF-n-t, where $n \in [2, 15], t \in [0, 7]$. It is a length-preserving binary block cipher family with a width of 16 to 127 bits determined by $(8 \cdot n + t)$. Through a new cycle-sliding procedure it provides format-preserving encryption of any single plaintext modulo $Q \in (2^{15}, 2^{127}]$.

To cover other cases not in length 16 to 127 bits, we add in Appendix F the description of additional ALF members: ALF-0 and ALF-1-t, $t \in [1, 7]$, provide length-preserving encryption covering the range of 1 to 15 bits; ALF-16-t, $t \in [0, 16]$, provides length-preserving encryption with width of 128 to 144 bits; and ALF-L that provides a length-preserving encryption of arbitrary binary width for longer plaintexts. ALF-L proposes a joint encryption of multiple fields with possibly distinct moduli, and when the moduli are the same, then ALF-L covers use cases supported by FF1 and FF3.

Although ALF algorithms partly reuse well-known techniques with the difference of selected components and aligning them to speed up software performance, one of the main contributions in this paper is the design of ALF-n-t, showing an efficient way to utilise AES-NI for creating a binary block cipher with a block size varying between 16 and 127

bits, referred to as a family of length-preserving encryption algorithms. This cipher applies to the case where the plaintext is *binary*, i.e., the domain size $Q = 2^k$ for some $k \in [16, 127]$. The ALF-n-t round function is then enhanced to support a format-preserving encryption with an arbitrary *non-binary* domain size Q within the corresponding range. The ALF block cipher family is turned into a format-preserving encryption by a new cycle-sliding transformation, improving slightly on traditional cycle-walking.

Finally, although the proposed ALF already covers many use cases, its components can be adapted to a variety of other use cases not covered in this paper, e.g., when the moduli q_i are larger than 2^{16}, or a parallel encryption of a plurality of plaintexts that may need to be later decrypted individually, etc.

The paper is organised as follows. After some preliminaries ending this section, in Section 2 we present the main design, a length-preserving block cipher called ALF-n-t. Section 3 then presents the main security analysis of ALF-n-t, with more details to be found in the appendix. Section 4 suggests an improved approach to transform the length-preserving block cipher into format-preserving encryption. Appendix F contains the description of complementary ALF designs for domain sizes not covered by ALF-n-t. Finally, Section 5 presents the software performance evaluation, followed by our conclusions.

1.1 Preliminaries

By $\mathbb{N}_k^\ell$ we denote a vector of ℓ unsigned integers, each fits in k bits, and $\mathbb{B}_k^\ell$ is a vector of ℓ blocks each consisting of k bytes, can be identified as $\mathbb{B}_k^\ell \sim \{\mathbb{N}_8^k\}^\ell$. The notation $\{x\}^k$ is used to define a vector of k bytes each having the value x.

In this paper, we adopt a seamless mapping between different representations performed in the little-endian mapping. For example, a vector of bytes $\mathbb{B}_k$ can be seen also as a vector of $8k$ bits or a large integer $\mathbb{N}_{8k}$. When we have a block of k bytes, $X \in \mathbb{B}_k$, then individual bytes of X are referred by $(X_0, X_1, \ldots, X_{k-1})$ where X_0 is the least significant 8 bits of X. Meanwhile, $X_{[a,b]}$ denotes $(X_a, X_{a+1}, \ldots, X_b)$. When X is an operand in an arithmetic operation, it should be viewed as an integer that has the corresponding binary representation.

The concatenation expression $Z = X \| Y$ should be understood as the bits or bytes of X are placed in the least significant indices of the resulting vector Z, followed by the bits or bytes of Y. Given $X \in \mathbb{N}_n^m$, the expression $X \lll_n k$ means shifting each n-bit block of X by k bits towards the most significant bit while shifting in k zero bits. Similarly, $X \ggg_n k$ shifts each block of n bits of X towards the least significant bits by k bits. The expression $X|_t$ keeps the t least significant bits and zeroise the other bits. That is, $X|_t = X \mod 2^t$.

Shuffles. The "shuffle" operation is inspired by and leverages the x86 instruction `PSHUFB`.[1] It takes two inputs $X \in \mathbb{B}_{16}$ and a *shuffle control vector* $\alpha \in \mathbb{B}_{16}$ which contains integer values from the range $[-1, 15]$, [2] and outputs a vector of 16 bytes, each of which is from X and selected in accordance with α, while the index (-1) results in the constant 0. For simplicity, we call α as a *shuffle* and treat it as an operator acting on a 16-byte vector with these concrete control values, i.e., by $Y = \alpha(X)$ we get $Y_i = X_{\alpha_i}$ if $\alpha_i \in [0, 15]$ and $Y_i = 0$, otherwise, for $i \in [0, 15]$.

AES. There are four steps in the standard AES round function. We abbreviate them as SB, SR, MC, and RK for `SubBytes`, `ShiftRows`, `MixColumn`, and `AddRoundKey`, respectively. The inverse operations are SB', SR', MC', and RK'. Note that the order of SB and SR is interchangeable, since they operate on bytes. This property applies to any permutation π, i.e., $SB \circ \pi(X) = \pi \circ SB(X)$. We use $\texttt{aesenc}(X, K) = RK(K) \circ MC \circ SB \circ SR(X)$ and $\texttt{aesdec}(X, K) = RK'(K) \circ MC' \circ SB' \circ SR'(X)$ as

[1] For Intel/AMD and ARM CPUs, the intrinsics are `_mm_shuffle_epi8` and `vqtbl1q_s8`, respectively.

[2] In practice, -1 is the value 255 or `0xff`; we skip the behaviour of the shuffle instruction for other values in the range $[16, 254]$ since we do not need them.

defined in AES-NI, where K is the round key XORed to the state in the last step RK. In certain places the round key will be skipped to simplify the description, but assumed to be present. Finally, `aesenclast` and `aesdeclast` are the encryption and decryption rounds but without MC/MC'.

2 The design of ALF-n-t

In this section, we propose a family of length-preserving block ciphers called ALF-n-t, where the length consists of $n \in [2, 15]$ full bytes and $t \in [0, 7]$ unaligned bits, supporting widths from 16 to 127 bits. The target security is 128 bits. As examples, encryption of IPv4 and MAC addresses can directly use ALF-4-0 and ALF-6-0, respectively.

We start in Section 2.1 with a high-level overview of AES-n-t, followed by design ideas and challenges to adapt AES round for an n-byte width block cipher in Section 2.2, then adding t bits to the total width in Section 2.3. In Section 2.4 we derive *encryption and decryption rounds* for ALF-n-t based on the ideas outlined in Sections 2.2 and 2.3. A more formal definition of ALF-n-t is then given in Section 2.5, followed by details on parameters in Sections 2.6 to 2.8, aligned with a 128-bit security as the main target. Afterwards, we give details on the round keys for both encryption and decryption directions in Sections 2.9 and 2.10. Finally, we end this section with security claims and limitations in Section 2.11 and an example of ALF-2-7 in Section 2.12. In order to make it simpler to instantiate and study ALF-n-t, Appendix C contains precomputed parameters and vectors discussed in Sections 2.6 to 2.10, and also an exampled instantiation in C++.

2.1 General overview

ALF-n-t is an $(8n + t)$-bit width block cipher that encrypts an input plaintext by repeating a well-defined round function for a number of times, where each call to the round function is supplied with its own round key. In order to reach high performance, the encryption round leverages the instruction `aesenc` as a nonlinear component, while extended with a few additional lightweight linear operations. The core idea to design ALF-n-t was first to find a suitable n-byte width round function, then extend the block width with additional t bits, and finally pick the right number of rounds to meet security targets and margins. For a simpler definition of ALF-n-t, the state of ALF-n-t is split in two parts: n full bytes are stored in a 128-bit register X, and t unaligned bits are stored in another 128-bit register E, both occupying least significant bits and bytes of the registers.[3]

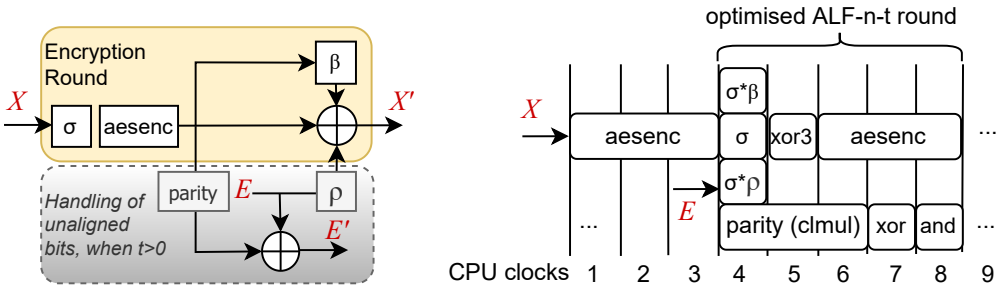

Figure 1: ALF-n-t encryption flow (left) and a possible round optimisation (right).

ALF-n-0 design overview. While the standard AES round operates over a 16-byte state, additional instructions between these rounds are required in order to preserve the

[3] In another implementation, E can be replaced by a single byte, but for an efficient data structure, we describe the algorithm with E being represented by a 128-bit XMM register.

n-byte width of input and output blocks. To minimise the performance impact, each call to `aesenc` is extended by a minimum number of lightweight instructions having a low latency and throughput. The order of these instructions is also designed to ensure a short critical path in an optimised implementation. Figure 1 (left) depicts an overview of the encryption round of ALF-n-t; here, `aesenc` is applied to the 128-bit state X after a well-defined constant shuffle σ, followed by the second shuffle β and an XOR operation. The resulting 128-bit state X' is then truncated down to n bytes, i.e., $(16 - n)$ most significant bytes of X' can be simply ignored. The set of parameters, such as σ and β, is different for different ns. Finally, the decryption round function share a similar design look and properties as for the encryption direction, and more details will be given in Section 2.5.

ALF-n-t design overview. When $t \neq 0$, the round function needs to be supplied with a solution for an entropy exchange between X (n bytes) and E (t bits) parts of the state. For this, a *parity* function Eq. (3) is applied to the intermediate state X and a shuffle ρ to E, followed by an XOR that completes the exchange of entropy between X and E.[4] Thanks to the independence of these two parts since the operations after the AES round function can be executed *in parallel* on modern CPUs and this way maintain a low latency by leveraging CPU stitching and interleaving effects. The total latency of the ALF-n-t round function is thus roughly doubled compared to a single AES-NI *instruction*. The round optimisation hinted by Figure 1 (right) demonstrates that the ALF round may have latency of only 5 CPU clocks versus 3 clocks for the standard AES round. For one round of a length-preserving block cipher this is a relatively small cost.

State. The state representation of ALF-n-0 follows the standard AES, arranging n bytes in a 4×4 state matrix where the meaningful n bytes fill the columns one by one and unoccupied cells are filled with 0s. The last used column can be not full, but is still viewed as *occupied* for the round computation. For example, ALF-5-0 occupies two full columns (8 bytes) although the plaintext is only 5 bytes in the state register. We denote the number of bytes in occupied columns by m, derived by $m = 4 \cdot \lceil n/4 \rceil$.

Parameters. The number of rounds for ALF-n-t is denoted by $r^{(n,t)}$. This superscript notation also applies to other parameters, e.g., shuffles $\sigma^{(n,t)}$, $\beta^{(n,t)}$, and $\rho^{(n,t)}$. When "t" is missing, it means the parameter works for all t, similarly for "n".

In the next subsections we describe techniques used to control the width of ALF-n-t, followed by a more formal definition and the exact parameters and shuffles.

2.2 Adapting AES-NI for an n-byte wide block cipher

The operations σ, β and XOR help to amend `aesenc` in order to process an n-byte input and result in an n-byte *preserved* output, i.e., where the first n bytes of the output are meaningful and other bytes are ignored. SR can be skipped since it can be cancelled by σ, thus, only the MC operation changes the length of the input. Specifically, given an n-byte input, MC expands it to an m-byte output in $\lceil n/4 \rceil$ full columns. Therefore, we will merely focus on fixing MC and MC', as depicted by Figure 2 (left).

Let us introduce a new shuffle α in order to simplify further explanations, while σ to be derived later. The idea is that α expands the meaningful n input bytes into m bytes to be processed by MC, while β can be seen as a compensation vector to the state that shrinks m resulting bytes down to n meaningful output bytes (in addition, β can zeroise unused bytes). The above argument brings the requirement on α and β: given an n-byte X, the output $X' = MC(\alpha(X)) \oplus \beta(MC(\alpha(X)))$ should result in the first n bytes as the meaningful output vector, while the reverse decryption procedure having a similar scheme *exists*.

However, peculiarities of AES-NI on Intel/AMD CPUs bring new restrictions on the shuffles, since the instructions `aesenc` and `aesdec` are not aligned as demonstrated in

[4] In the reference implementation of ALF we have used an alternative method to compute the parity and thus stored the t unaligned bits in the fourth byte E_3, instead of E_0.

Figure 3. I.e., the 128-bit state register X is only available for a width-control modification after[5] the atomic call to `aesenc` in software, but that exact state falls in the middle of the atomic instruction `aesdec` and therefore cannot be directly accessed for a reverse modification on the decryption flow. Instead, the state that is actually in the middle of `aesenc` can be reverse-modified at the point where `aesdec` ends, i.e., the "state recovery" on the decryption flow can only be done *indirectly*, and any ALF's method for the width control should take into account this unalignment.

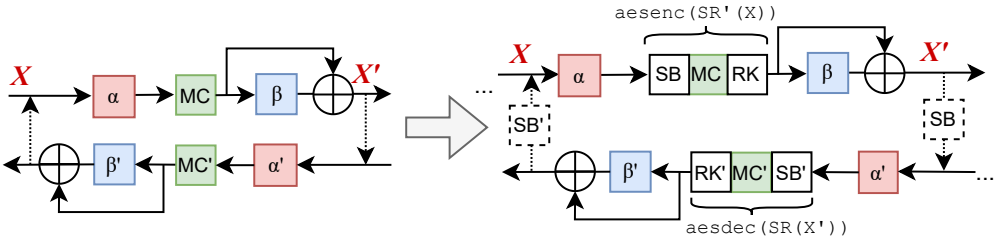

Figure 2: MixColumn bypass methods (left) and mapping to the design of ALF-n-t (right).

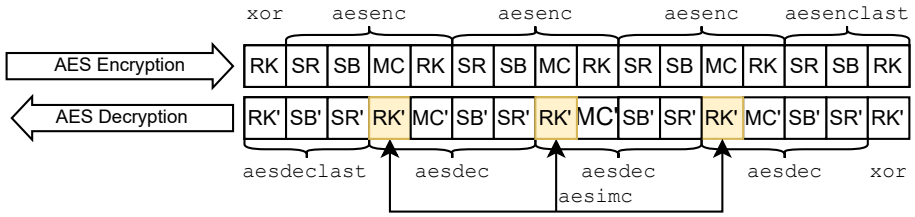

Figure 3: AES with 4 rounds using AES-NI (Intel/AMD).

That is, we first need to find shuffles $(\alpha, \beta, \alpha', \beta')$ such that the whole operation between X and X' depicted on Figure 2 (left) is itself invertible given the preferred design sketch of the low-latency encryption and decryption rounds, i.e., the cycle $X \xrightarrow{Enc} X' \xrightarrow{Dec} X''$ should work such that $X'' = X$ for any X. Afterwards, that cycle can be mapped to actual encryption and decryption rounds of ALF-n-t while using unaligned AES-NI instructions, as shown on Figure 2 (right). The exact set of found shuffles that meet these requirements are given in Section 2.6.

We note that Figure 2 (right) depicts the actual situation in practice where there are SB and RK operations. Since the shuffle α may result in the constant '0' in some positions, these 0s become `0x63` after SB. On MC', they are `0x52` from the inverse Sboxes of SB'. These side effects will be offset by the decryption round keys, see Sections 2.4 and 2.10.

2.3 Unaligned t-bit entropy exchange

While Section 2.2 paves the way to build an n-byte width block cipher, here we extend the design by adding unaligned $t \in [0, 7]$ bits to the width. Every round of ALF-n-t is thus extended by additional lightweight instructions to exchange t bits of entropy between the main state X having n bytes and the extra register E storing unaligned t bits, and we call these methods a *t-bit exchange*. Clearly, no additional instructions are needed when $t = 0$.

Due to the non-alignment of the AES-NI instructions, any t-bit exchange method must be able to pass through the MC operation, i.e., the t-bit exchange on the encryption

[5] And/or *before*, which can be seen as modifications applied *after* the *previous* `aesenc`.

direction would happen *after* the MC block but on the decryption we can only revert the procedure *before* that corresponding MC block (or vice versa). The method that we found to satisfy these requirements with a simple implementation exploits the parity property of MC. Let $X \in \mathbb{B}_{16}$ and

$$\texttt{parity}_i(X) = X_i \oplus X_{i+1} \oplus X_{i+2} \oplus X_{i+3} \tag{1}$$

be the *parity function* over one of the four columns of X, where $i \in \{0, 4, 8, 12\}$. For ALF-n-t we fix $i = 0$, i.e. always taking the parity from the first column, thus i can be omitted in further formulae. It is easy to check that the parity value remains the same before and after MC (also MC'), i.e.,

$$\begin{aligned} \forall X: &\quad \texttt{parity}(X) = \texttt{parity}(MC(X)) = \texttt{parity}(MC'(X)), \\ \forall X, K: &\quad \texttt{parity}(X \oplus K) = \texttt{parity}(X) \oplus \texttt{parity}(K). \end{aligned}$$

A truncated parity value is added to the t bits while in parallel these bits are added to certain bytes of the main state following a new shuffle ρ. Given that E_0 is the byte that stores the unaligned t bits, the shuffle $\rho(E)$ copies that byte to appropriate positions for a subsequent addition to the main state. The summary on the adopted t-bit exchange:

$$\begin{array}{lll} \text{Forward:} & U = MC(\alpha(X)) & \text{Reverse:} \quad W = MC'(\alpha'(X')) \\ & E' = E \oplus \texttt{parity}(U)|_t & \quad\quad E = E' \oplus \texttt{parity}(W)|_t \\ & X' = U \oplus \beta(U) \oplus \rho(E) & \quad\quad X = W \oplus \beta'(W) \oplus \rho'(E) \end{array} \tag{2}$$

The shuffle instruction takes 1 CPU clock, and $\texttt{parity}(U)$ can be implemented by, e.g., two 32-bit shifts and two XORs [6]:

$$Z = U \oplus (U \gg_{32} 16) \quad \Rightarrow \quad \texttt{parity}(U) = Z \oplus (Z \gg_{32} 8). \tag{3}$$

The output is a 128-bit value where the first byte of each column stores the parity of the corresponding column of U. The t-bit truncation can be implemented by a bitwise AND operation with the mask $M^{(t)} \in \mathbb{B}_{16}$:

$$M_i^{(t)} = 2^t - 1 \quad \text{for } i = 0, \quad \text{otherwise } 0. \tag{4}$$

The reason why ALF-n-t always takes the first column for the t-bit exchange is a simpler implementation, as well as the possibility to implement a parallel encryption of multiple symbols while sharing the same state registers (X, E), see Appendix A for more details. We forced $\rho = \rho'$ in our search so that the number of needed parameters is reduced.[7] In the rest of this paper, we will use ρ for both encryption and decryption directions.

2.4 Deriving encryption and decryption rounds for ALF-n-t

In this section we derive *encryption and decryption rounds* for ALF-n-t, based on the ideas presented in Sections 2.2 and 2.3. We call rounds that are not yet final for ALF by *forward and reverse rounds*. **Assume** the forward round as shown in Eq. (2) (with the support of Figure 2(left)) is injective and the reverse flow is also injective such that (X', E') is correctly evaluated to the initial state (X, E) through the reverse round (we leave the proof of this assumption till later). Then assuming (X, E) is a random input with X having n

[6] An alternative method is to use a carry-less multiplication instruction $V =$`pclmulqdq`$(X,$ `0x01010101`$)$ on Intel/AMD CPUs with latency 3; then the parity of the first and second columns will be in the bytes V_3 and V_7, respectively. A similar instruction exists on ARM CPUs – `VMULL/PMULL`.

[7] Requiring $\rho = \rho'$ may not be optimal for some ns, but it is optimal for most of the cases.

bytes and E having t bits, we can draw a compute program that simulates these forward (F1) and reverse (R1) rounds, as follows:

Forward **F1:**	Reverse **R1:**
$X = MC(\alpha(X))$	$X = MC'(\alpha'(X))$
$P = \texttt{parity}(X)$	$P = \texttt{parity}(X)$
$X = X \oplus \beta(X) \oplus \rho(E)$	$E = E \oplus P\|_t$
$E = E \oplus P\|_t$	$X = X \oplus \beta'(X) \oplus \rho(E)$

After a sequential call of the forward round F1 for $r \geq 1$ times, then continuing with the reverse round R1 also calling it for r times, we would get the resulting output values of (X, E) to coincide with the initial input values due to the initial assumption of correctness, and it holds for any valid input.

Our approach to derive ALF encryption and decryption functions is to add bijective operations to the above rounds step-by-step, and/or perform equivalent modifications, while keeping the properties of injectivity (decryptability) and reversibility (correctness).

Although the exact vectors and parameters will be given later in Sections 2.6 to 2.8, we note the property $\forall n : \texttt{parity}(\alpha'(\rho(E))) = 0$ that we imposed in our simulations. Then based on the assumption of correctness of the above F1/R1, the following relation between α' and β holds:

$$\begin{aligned} P =& \texttt{parity}^{(\text{from F1})}(X) = \texttt{parity}^{(\text{from R1})}(MC'(\alpha'(X \oplus \beta(X) \oplus \rho(E)))) \\ =& \texttt{parity}(\alpha'(X \oplus \beta(X))) \end{aligned} \tag{5}$$

F2/R2. In order to obtain a larger space of candidates for ALF-n-t an n-byte permutation π is added to the rounds. It enables us to further improve security after deciding on other parameters. Intuitively, α, β, and ρ guide the main structure and shrink down the design space of ALF-n-t instances while π explores candidates inside this space.

Thus, as the first modification, we add to the rounds F1/R1 the steps of π and SB. These steps are injective and invertible; thus, the modified forward-reverse cycle is still injective and invertible. Then we also add a round key to the scheme K in the encryption flow, so that a corresponding key K' introduced in the reverse function compensates the offset $X \oplus K$. Note that the locations where K and K' are inserted do not affect the parity P and thus E will be correctly recovered on the reverse flow. The offset K' can be derived by following the sequence of linear operations between the steps where we add K and K':

$$K' = MC'(\alpha'(K \oplus \beta(K))) \tag{6}$$

F2:	**R2:**	**F3:**	**R3:**
$X = \pi(X)$	$X = SB(X)$	$X = \pi(X)$	$X = SB(X)$
$X = SB(X)$	$X = SB'(X)$	$X = \alpha(X)$	$X = \alpha'(X)$
$X = \alpha(X)$	$X = \alpha'(X)$	$X = SB(X) \oplus C$	$X = SB'(X) \oplus C'$
$X = MC(X)$	$X = MC'(X)$	$X = MC(X)$	$X = MC'(X)$
$\Rightarrow P = \texttt{parity}(X)$	$X = X \oplus K'$	$X = X \oplus K$	$P = \texttt{parity}(X)$
$X = X \oplus K$	$\Rightarrow P = \texttt{parity}(X)$	$P = \texttt{parity}(X)$	$X = X \oplus K'$
$X = X \oplus \beta(X)$	$\Rightarrow E = E \oplus P\|_t$	$X = X \oplus \beta(X)$	$E = E \oplus P\|_t$
$\quad \oplus \rho(E)$	$X = X \oplus \beta'(X) \oplus \rho(E)$	$\quad \oplus \rho(E)$	$X = X \oplus \beta'(X) \oplus \rho(E)$
$\Rightarrow E = E \oplus P\|_t$	$X = SB'(X)$	$E = E \oplus P\|_t$	$X = SB'(X)$
	$X = \pi'(X)$		$X = \pi'(X)$

F3/R3. As the next step of the program re-ordering, we swap the steps SB and α in F2, the steps SB' and α' in R2, as well as the steps with $\texttt{parity}$ and "$\oplus K$" in both F2 and R2. Consider that $\alpha(SB(X)) = SB(\alpha(X)) \oplus C$, where C is a constant vector such that $C_i = \texttt{0x63}$ where $\alpha_i = -1$, otherwise zero. Also consider $\alpha'(SB'(X)) = SB'(\alpha'(X)) \oplus C'$,

with $C'_i = \texttt{0x52}$ where $\alpha'_i = -1$, otherwise zero. The last swap is seamless for E since $\texttt{parity}(K) = \texttt{parity}(K')$ due to Eqs. (5) and (6). The result of these swaps is F3/R3.

ALF encryption round. The constant C in F3 can be moved a few steps down where the round key is added, i.e., to the step $X = X \oplus K \oplus MC(C)$. After the variable substitution $K \to K \oplus MC(C)$ in Eq. (6) and replacing K' in R3 by:

$$K' = MC'(\alpha'(K \oplus MC(C) \oplus \beta(K \oplus MC(C)))) \tag{7}$$

we then get the following ALF encryption round F4 (here we also insert the identity $SR \circ SR'$ to facilitate decomposition):

F4:	**R4:**
	$X = \pi(X)$ (A)
	$X = SB(X)$
$X = \underbrace{RK \circ MC \circ SB \circ SR}_{\texttt{aesenc}(X,K)} \circ \underbrace{SR' \circ \alpha \circ \pi}_{\sigma}(X)$	$X = \underbrace{MC' \circ SB' \circ SR'}_{\texttt{aesdec}(X,0)} \circ \underbrace{SR \circ \alpha' \circ \pi'}_{\sigma'}(X)$
$P = \texttt{parity}(X)$	$P = \texttt{parity}(X) \oplus B$ (B)
	$X = X \oplus K'$
$X = X \oplus \beta(X) \oplus \rho(E)$	$E = E \oplus P\|_t$
$E = E \oplus P\|_t$	$X = X \oplus \beta'(X) \oplus \rho(E)$
	$X = SB'(X)$
	$X = \pi'(X)$ (C)

The above already proves the decryptability under the initial assumption, since all additions to the rounds were injective and invertible, and/or re-orderings were equivalent. The reverse round R3 with the fixed K' per Eq. (7) correctly decrypts the encryption round F4.

ALF decryption round. What remains is to show how the reverse program R3 is mapped to the ALF decryption round. Consequently, the constant C' in R3 can be moved down for a few steps and become a part of K', and the parity P is compensated with a constant B:

$$\begin{aligned} K' &= MC'(C') \oplus MC'(\alpha'(K \oplus MC(C) \oplus \beta(K \oplus MC(C)))) \\ &= MC'(\alpha'(K \oplus \beta(K)) \oplus A) \\ B &= \texttt{parity}(MC'(C')) = \texttt{parity}(C') \end{aligned}$$

where $A^{(n)}$ is a compensation constant that can be (pre-)computed, see Eq. (10).

The final re-ordering is to insert SR/SR', π/π', while preserving the functionality since F4 is now locked; then swap the first two steps SB and π, resulting in the final pair F4/R4. Here we see that the reverse round R4 is split into three parts: the first part (A) becomes *the auxiliary step* of the ALF decryption, the middle part (B) is the actual ALF *decryption round*, and the last part (C) is the *State Retrieval Function* (SRF), the purpose of which is to get the state that is aligned with the ALF encryption round F4.

Note that when r consecutive reverse functions are called, the last part of the k-th call is cancelled by the first part of the $(k+1)$-th call, thus the decryption round (B) in the middle part of R4 will be called r times *consecutively*, while the first and last parts to be called only once.

Decryptability and correctness. The final programs F4/R4 are injective and reversible derivations from the starting F1/R1. What remains is to prove that the initial assumption in this section is correct. This, however, can only be verified given concrete vectors $\alpha^{(n)}, \beta^{(n)}$, et cetera, for each (n, t). The initial assumption is made on the forward and reverse rounds F1/R1, which implement only linear operations. Thus, one can derive two corresponding $(8n+t) \times (8n+t)$ binary matrices $L_E^{(n,t)}$ and $L_D^{(n,t)}$, then check that

$L_E^{(n,t)}$ has a full rank – this proves decryptability, and that $L_D^{(n,t)} \cdot L_E^{(n,t)} = \mathbf{I}$ – this proves the correctness. More details on this last verification step is given in Appendix B.

2.5 Definition of ALF-n-t

Having the derived rounds F4 and R4 from Section 2.4, we are ready to define ALF-n-t more formally.

The encryption and decryption procedures are defined by Algorithm 1 and depicted in Figure 4. In order to align with FPE needs in Section 4, there is no whitening key before the first round, and the last round also keeps the MC operation. The auxiliary step can be implemented as, e.g., $SB \circ \pi(X) = SB \circ SR \circ SR' \circ \pi(X) = \texttt{aesenclast}(\sigma(X), \{0\}^{16})$. Note that $\sigma = SR' \circ \alpha \circ \pi$, but since for each n we have $\alpha^{(n)}|_{8n} = \mathbf{I}|_{8n}$ in the first n bytes, then we can use σ to actually implement $SR' \circ \pi$ in this place. On the other hand, the SRF can be implemented as $SB' \circ \pi'(X) = SB' \circ SR' \circ SR \circ \pi'(X) = \texttt{aesdeclast}(\tau'(X), \{0\}^{16})$, where $\tau' = SR' \circ \pi'$. The reason of having the auxiliary step and SRF is to fix the unalignment of AES-NI on Intel/AMD CPUs. In NEON, the auxiliary step and SRF could be avoided thus leading to the same performance for both encryption and decryption. Finally, when $t = 0$ then the actions involving `parity` and E can be simply ignored.

Algorithm 1 ALF-n-t encryption and decryption

function $\texttt{ALFntUpdateE}(E, U, B, M) \to E$
  $U = U \oplus (U \gg_{32} 16)$
  $E = E \oplus B \oplus U$ ▷ B is a constant for decryption
  return $E = (E \oplus (U \gg_{32} 8))$ `AND` M
function $\texttt{ALFntEncrypt}(n, t, P \in \mathbb{N}_{8n+t}, RK[0..r-1]) \to C \in \mathbb{N}_{8n+t}$
  Split P as $X = P|_{8n}$, $E = (P \gg 8n)$
  Retrieve parameters $\alpha, \beta, \rho, \pi, r, M$ for (n, t), and derive $\sigma = SR' \circ \alpha \circ \pi$
  for $i = 0..r-1$ **do**
    $U = \texttt{aesenc}(\sigma(X), RK[i])$
    $X = U \oplus \beta(U) \oplus \rho(E)$
    $E = \texttt{ALFntUpdateE}(E, U, \{0\}^{16}, M)$ ▷ skipped when $t = 0$
  return $C = E_0 \cdot 2^{8n} + X|_{8n}$
function $\texttt{ALFntSRF}(\pi, X) \to Y$
  return $Y = \texttt{aesdeclast}(SR \circ \pi^{-1}(X), \{0\}^{16})$
function $\texttt{ALFntDecrypt}(n, t, C \in \mathbb{N}_{8n+t}, RK'[0..r-1]) \to P \in \mathbb{N}_{8n+t}$
  Split C as $X = C|_{8n}$, $E = (C \gg 8n)$
  Retrieve parameters $\alpha', \beta', \rho, \pi, r, B, M$ for (n, t), and derive $\sigma' = SR \circ \alpha' \circ \pi^{-1}$
  $X = \texttt{aesenclast}(SR' \circ \pi(X), \{0\}^{16})$ ▷ auxiliary step
  for $i = r-1..0$ **do**
    $W = \texttt{aesdec}(\sigma'(X), RK'[i])$
    $E = \texttt{ALFntUpdateE}(E, W \oplus RK'[i], B, M)$ ▷ skipped when $t = 0$
    $X = W \oplus \beta'(W) \oplus \rho(E)$
  return $P = E_0 \cdot 2^{8n} + \texttt{ALFntSRF}(\pi, X)|_{8n}$

Engine mixing. The design of ALF-n-t has a useful property that enables running multiple instances in parallel while sharing the same state registers. As mentioned before, ALF-n-t occupies only m bytes in the 16-byte state. The core idea of parallelisation is to reuse the remaining $16 - m$ bytes for additional encryption engines. The encryption and decryption algorithms are the same but the masks and shuffles need to be combined. More details on engine mixing and the way to combine shuffles is given in Appendix A.

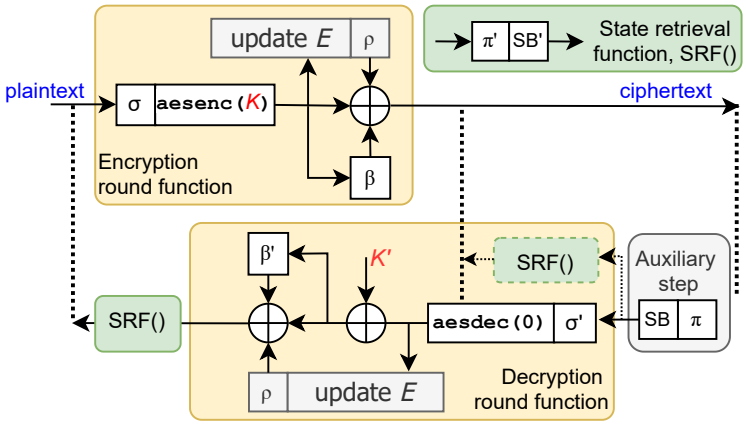

Figure 4: ALF-n-t encryption and decryption flows.

Table 1: MixColumn bypass control vectors $\alpha, \beta, \alpha', \beta'$, and ρ. The remaining bytes of 16-byte vectors are filled with (-1)s.

n	$\alpha^{(n)}$	$\beta^{(n)}$	$\alpha'^{(n)}$	$\beta'^{(n)}$	$\rho^{(n)}$
2	$(0,1)$	$(2,3)$	$(0,1)$	$(2,3)$	$(0,0)$
3	$(0,1,2)$	$(3,3,3)$	$(0,1,2)$	$(3,3,3)$	$(0,-1,0)$
4	$(0,1,2,3)$	$()$	$(0,1,2,3)$	$()$	$(0,0,0,0)$
5	$(0,1,2,3,4,1,-1,1)$	$(4,4,4,4,7)$	$(0,1,2,3,4,-1,4)$	$(5,5,7,5,1)$	$(0,-1,0,-1,0)$
6	$(0,1,2,3,4,5)$	$(2,3,6,7,2,3)$	$(0,1,-1,-1,2,3,4,5)$	$(4,5,4,5,6,7)$	$(0,0,-1,-1,0,0)$
7	$(0,1,2,3,4,5,6)$	$(1,7,1,1,1,1,1)$	$(0,-1,2,3,4,5,6,1)$	$(7,7,7,7,7,7,7)$	$(0,-1,0,-1,0,-1,0)$

2.6 α, β, and ρ shuffles

The set of parameters consists of σ, β, ρ, and the number of rounds, but since σ has its own structure we provide α and π instead so that σ is a derivative. In practice, only σ is (pre-)computed and stored, in case an engine mixing is not needed.

The search space for α and β increases exponentially when n becomes larger, especially when $n \geq 8$. Therefore, we focus on finding α and β for $n \in [2,7]$, then derive parameters for $n \geq 8$ following a simple rule. Thus, both shuffles should provide a good security level not only for small ns but also for large ns. Having a set of candidates we follow the results of differential and linear cryptanalysis and select those that require the fewest number of rounds to reach the 128-bit security in both analyses. Under this idea, we found the final α and β in Table 1 where trailing (-1)s are removed to keep the table clean. The formulae to derive for larger $n \in [8,15]$ are

$$\begin{aligned}
\alpha^{(n)} &= \texttt{ins4}(\texttt{add4}(\alpha^{(n-4)}), (4,5,6,7)), \quad \text{for } n \in [8,15], \\
\beta^{(n)} &= \texttt{ins4}(\texttt{add4}(\beta^{(n-4)}), (-1,-1,-1,-1)), \quad \text{for } n \in [8,15] \setminus 13, \\
\beta^{(n)} &= \texttt{ins4}(\texttt{add4}(\beta^{(n-4)}), (8,9,10,11)), \quad \text{for } n = 13,
\end{aligned}$$

where `ins4` and `add4` are defined by

$$\begin{aligned}
\gamma = \texttt{add4}(\mu) &:= \forall i \in [0,15]: \ \gamma_i = \mu_i \text{ if } \mu_i < 4, \text{ otherwise } \gamma_i = \mu_i + 4, \\
\gamma = \texttt{ins4}(\mu, \xi) &:= \gamma = \mu_{[0,3]} \| \xi_{[0,3]} \| \mu_{[4,11]}.
\end{aligned}$$

I.e., `add4` increases by 4 all values of μ that are in the range $[4,15]$, and `ins4` inserts ξ into μ after μ_3 and shifts out $\mu_{[12,15]}$. The same rules are applied to derive α' and β'.

For $n \geq 8$, these rules view the first and last columns of the state as consecutive bytes and take shuffles according to the total number of bytes in them. For example, when

$n = 10$, the first column has 4 bytes and the last (third) column has 2 bytes. Then, we "virtually" view these two columns as the case $n = 6$ and use corresponding shuffles. For $n \in \{8, 12\}$, shuffles of the case $n = 4$ only apply to the first column. This idea also ensures that the parity function is always computed on the first column, which makes it easier to combine multiple engines, see Appendix A. The case $n = 13$ is an "exception" from these rules due to a weak diffusion, the solution is that the third column is added to the second column that improves the diffusion and provides a better resistance to differential and linear attacks.[8] Overall, all cases reach a full diffusion within 2 rounds – same as in the standard AES.

A similar idea is applied to ρ, where the shuffles are determined for $n \in [2, 7]$, listed in the last column of Table 1, and then the following rule is taken to derive ρ for larger ns.

$$\rho^{(n)} = \texttt{ins4}(\rho^{(n-4)}, (-1, -1, -1, -1)), \quad \text{for } n \in [8, 15]. \tag{8}$$

2.7 π shuffles

We mainly tried to enumerate all possible permutations π for each n and test their properties under various cryptanalyses in Section 3. However, since each test costs more time when n grows, finishing an exhaustive search was only feasible for $n < 11$. For larger ns, we tested as many as we could and selected the best permutations found. Our decision on permutations was based on the following criteria: (1) when $t = 0$, a full diffusion should be achieved within two rounds; (2) the number of rounds to achieve a 128-bit security in both differential and linear cryptanalyses is minimised; (3) take the option where the number of active Sboxes grows faster. The final selection is given in Table 2.

Table 2: π shuffles.

n	$\pi^{(n)}$	n	$\pi^{(n)}$	n	$\pi^{(n)}$
–	—	6	$(2, 1, 0, 5, 4, 3)$	11	$(5, 7, 9, 8, 1, 0, 2, 4, 6, 3, 10)$
2	$(0, 1)$	7	$(0, 4, 3, 6, 2, 5, 1)$	12	$(0, 5, 6, 10, 4, 9, 2, 11, 8, 1, 3, 7)$
3	$(0, 2, 1)$	8	$(0, 1, 4, 5, 2, 3, 6, 7)$	13	$(5, 7, 8, 1, 2, 9, 4, 10, 11, 0, 6, 12, 3)$
4	$(0, 1, 2, 3)$	9	$(2, 4, 1, 5, 3, 6, 7, 8, 0)$	14	$(4, 11, 8, 3, 5, 1, 9, 2, 6, 0, 10, 13, 7, 12)$
5	$(1, 0, 4, 2, 3)$	10	$(7, 4, 1, 3, 0, 2, 5, 9, 6, 8)$	15	$(4, 8, 12, 7, 5, 9, 13, 2, 6, 10, 1, 3, 14, 0, 11)$

Accumulating the above, and as an example, we demonstrate the encryption function for ALF-13-0 with precomputed vectors using AES-NI and SSE intrinsics, i.e., implementing a 13-byte wide block cipher. The encryption round can be further optimised, as hinted in Figure 1 (right), down to only 5 CPU clocks.

```
__m128i encSigma13 = _mm_setr_epi8( 5, 7, 6,10, 2, 7,-1,12,11, 9, 8, 7, 3, 0, 4, 1);
__m128i encBeta13  = _mm_setr_epi8(12,12,12,12, 8, 9,10,11,-1,-1,-1,-1,15,-1,-1,-1);
__m128i X = _mm_loadu_si128((__m128i*)(in));   // the first 13 bytes are the input
for (int i = 0; i < 12; i++)
{   X = _mm_aesenc_si128(_mm_shuffle_epi8(X, encSigma13), RK[i]);
    X = _mm_xor_si128(X, _mm_shuffle_epi8(X, encBeta13));
}
_mm_storeu_si128((__m128i*)(out), X); // the first 13 bytes are the output
```

2.8 Number of rounds

The number of rounds should be good enough to prevent all attacks found by cryptanalysis, plus some security margin. If there is an attack on r' rounds then the cipher should have at least $r \geq r' + 1$ rounds. The value of r' can be found in Tables 5 to 7.

For the security margin we add 3 more rounds when $t = 0$, and 4 extra rounds when $t \neq 0$, since the t-bit exchange has a weaker diffusion property and one more round is therefore needed. Finally, the number of rounds for each (n, t) are determined and given

[8] Without that extra XOR operation, $n = 13$ needs 3 rounds to achieve a full diffusion.

in Table 3. In order to align the cipher for the FPE transformation, see Section 4, each number is set to the smallest *even* number, but not less than the derived minimum. We note that for different $t \in [1, 7]$ under the same n the same number of rounds is adopted, since we found that these cases share similar properties.

Table 3: The number of rounds $r^{(n,t)}$ for each ALF-n-t instance.

n	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$t = 0$	20	16	14	14	14	14	12	12	12	12	12	12	12	12
$t \neq 0$	28	24	18	18	16	16	16	16	14	14	14	14	14	12

2.9 Round keys for encryption

In this section we specify how round keys for the encryption direction $RK[0..r-1] \in \mathbb{B}_n^r$ are derived from the secret key, the tweak, and other parameters. Each round key is an n-byte chunk stored in a 128-bit register, while the upper $(16-n)$ remaining bytes of every round key are zeroised. We assume there is a single plaintext symbol ($N = 1$) modulo Q.

We have selected SMAC-3/4 [WMEJ25] as the basis for the round keys generation that provides security of at least 152 bits, which is more than needed. SMAC-3/4 is a modern AES-round-based MAC algorithm that is fast in software and small in state, thus leaving enough of registers' space for other operations. One of the peculiarities of SMAC-3/4 is that it requires a compression step of a constant message block after at most 3 varying messages.

The state of SMAC is three 128-bit registers $(A1, A2, A3)$, and $\texttt{SMAC}_R(A1, A2, A3, M)$ is a function to compress a 128-bit message M and update the state, defined by:

$$\begin{aligned}
&\texttt{SMAC}_R(A1, A2, A3, M) \to (A1', A2', A3') := \\
&\quad A1' = \sigma_{42}(A2 \oplus A3 \oplus M) \\
&\quad A2' = \texttt{aesenc}(A1, M) \\
&\quad A3' = \texttt{aesenc}(A2, M) \\
&\text{where } \sigma_{42} = (7, 14, 15, 10, 12, 13, 3, 0, 4, 6, 1, 5, 8, 11, 2, 9)
\end{aligned}$$

The first step is the `KeyInit` function that absorbs the 128-bit secret key and a 64-bit application ID $AppID$ (for domain separation on the application level, can be set to zero if not used). `KeyInit` starts with setting the initial values of the state:

$$A1 = AppID \mid (1 \ll 64), \qquad A2 = key, \qquad A3 = Q - 1,$$

in the little-endian encoding. The internal state is then shuffled by calling the function $\texttt{InitFinal}(A1, A2, A3, 1)$ in order to bring it to a pseudo-random state, defined by

$$\begin{aligned}
&\texttt{InitFinal}(A1, A2, A3, C) \to (A1', A2', A3') := \\
&\qquad \text{return } (A1, A2, A3) \oplus \texttt{SMAC}_R^9(A1, A2, A3, C)
\end{aligned}$$

where C is a nonzero constant, and the compression function $\texttt{SMAC}_R$ is called 9 times.

The next function is `TweakInit` that generates the actual round keys for underlaying ALF variants. As the first step, the 128-bit tweak is compressed. For ALF-n-t, there are a few *parallel* calls to $\texttt{InitFinal}(A1, A2, A3, C)$ with $C = 1, 2, \ldots$ being a counter, until the required number of bytes are collected. Each of these calls gives 48 bytes from the resulting state $(A1, A2, A3)$. When enough bytes are collected, we then consecutively take n-byte blocks as round keys for the relevant ALF instance.

2.10 Round keys for decryption

The decryption round keys, $RK'[]$s, are derived from $RK[]$s by incorporating compensation operations and constants, enabling the techniques described in Section 2.2 and 2.3 to be applied. Following the flow depicted on Figure 2, for each $i \in [0, r-1]$:

$$RK'[i] = MC'(\alpha' \circ \beta(RK[i]) \oplus \alpha'(RK[i]) \oplus A), \tag{9}$$

where A is the compensation constant needed to negate the effects of $Sbox(0)$ and $Sbox^{-1}(0)$, mentioned in Section 2.2, that can be (pre-)computed as follows:

$$\begin{aligned} H &= \{\texttt{0x52}\}^m || \{\texttt{0x00}\}^{16-m}, \\ G &= \texttt{aesenc}(SR' \circ \alpha(H), \{\texttt{0x00}\}^{16}), \\ A &= \alpha'(G \oplus \beta(G) \oplus H) \oplus H. \end{aligned} \tag{10}$$

For the unaligned t bits, an Sbox compensation is derived by

$$B = \texttt{parity}(\alpha'(H)) \quad \Rightarrow \quad B = \texttt{0x52}, \text{ for } n = 3, 7, 11, 15, \text{ otherwise } B = 0. \tag{11}$$

2.11 Security claims and limitations

The key size and tweak size of ALF are 128 bits. Based on our security analysis, we claim 128-bit security against key-recovery attacks for each ALF-n-t instance. This means that there should be no method to find the key significantly more efficient than testing different keys in encryption/decryption. Standard attack assumptions are assumed, like chosen plaintext or chosen ciphertext. There is no data limitation or other restrictions for the adversary. For attacks of distinguishing type, we claim there is no significantly more efficient distinguisher compared to an ideal block cipher of the same width. Also, we do not consider security in a side-channel scenario.

2.12 Example: ALF-2-7

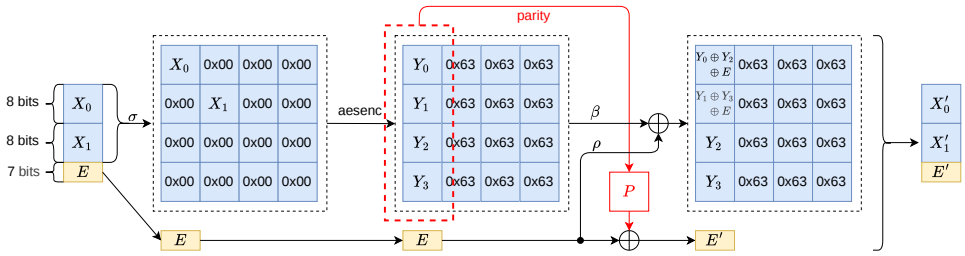

Figure 5: ALF-2-7 encryption round.

In this subsection, we take the 23-bit block cipher ALF-2-7 as an example to give a more concrete view of how ALF-n-t works. The encryption flow of the round function is shown in Figure 5, which should be read from left to right.

The input is 2 bytes and 7 bits. As explained in previous subsections, operations π, SR', and α are merged into σ which shuffles the input bytes into the first AES matrix state while the unaligned t bits in E remain unchanged. The AES round function `aesenc` is then applied to the 16 bytes that outputs one meaningful column and three useless columns; note that $\text{Sbox}(\texttt{0x00}) = \texttt{0x63}$. By the next step, a one-byte parity value is computed over the first column to produce $P = Y_0 \oplus Y_1 \oplus Y_2 \oplus Y_3$. The lower 7 bits of P are XORed into E to update it to $E' = E \oplus P|_7$. Finally, the shuffles β, ρ, and the following XOR mix

the first column and the old unaligned bits E into a new state, after which the bytes Y_2 and Y_3 are dropped. After further dropping of useless columns, the output of the round function maintains the width of the block cipher, that is 2 bytes and 7 bits.

3 Security analysis of ALF-n-t

In this section, we analyse the proposed ALF-n-t given the parameters in Sections 2.6 to 2.8. The security analysis includes the full diffusion round count, differential, linear, and integral analysis, and we use well-known automatic search techniques based on Constraint Programming (CP) to find optimal attacks. Appendix D provides more detailed results, including the number of active Sboxes for every number of rounds, but here we only present relevant extracts. The source code to perform these analyses is publicly available at `https://github.com/ONG/ALF-tools`, which takes OR-Tools [PD] as the solver.

3.1 Full diffusion

The *avalanche effect* is a property of block ciphers whereby a one-bit change in the plaintext or key causes approximately half of the ciphertext bits to change [Sta17, WT86], on average. In the context of the ALF design, we are looking for *full diffusion*, which is described by the minimum number of rounds for which every input bit is propagated to every output bit. Typically, the smaller number of rounds needed to reach the full diffusion, the smaller *total* number of rounds needed to satisfy the avalanche effect. For ALF-n-0, we targeted that full diffusion is achieved in at most 2 rounds, which is the same as for the standard AES [DR02]. When $t \neq 0$, full diffusion is achieved in at most 3 rounds.

In our analysis, we traced every bit of the state and derived the dependence of the output bits on the input bits. For example, given three bits b_0, b_1, and b_2 as well as the equation $b_2 = b_0 \oplus b_1$, we record that b_2 depends on b_0 and b_1. After a certain number of rounds, it can be seen that each output bit comes from all input bits. The specific number of rounds are presented in Table 4.

Table 4: The minimum number of rounds for a full diffusion.

n	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$t = 0$	1	2	1	2	2	2	2	2	2	2	2	2	2	2
$t \neq 0$	2	3	2	3	3	3	3	3	3	3	3	3	3	3

3.2 Differential analysis

In differential analysis, we created CP models and used an existing CP solver, OR-Tools, to find differential trails. Such a technique is proposed in [SGL+17]. For each intermediate value before and after each operation in ALF-n-t, a set of binary variables is created to represent the input and output bits. For linear operations, the constraints between these variables are simply the same linear equations as the linear operations. For example, if the operation is XOR between two values x and y and the result is z, then the constraint is also $z = x \oplus y$ in the model. In particular, the t-bit exchange operation is also a linear operation that involves copying of variables, permutations, and XORs. For XORs, the model is identical to that in the previous example. When copying a variable x into two new variables y_0 and y_1, the constraints $y_0 = x$ and $y_1 = x$ are created. When the state is permuted, we simply permute variables in the model. For each Sbox, we created constraints that, if any input bit takes the value 1, there must be at least one output bit being 1, and, if all input bits are 0, all output bits must also be 0. Meanwhile, if any input bit is 1, the Sbox is marked as active. Then, the objective of the model is to minimise the number of active Sboxes, which corresponds to the best differential trail. If a trail has k active

Sboxes, its probability is upper bounded by 2^{-6k} due to the property of the AES Sbox, and the complexity of a differential distinguisher can be as low as $O(2^{6k})$, see Appendix E for an example. Since the results where $t \neq 0$ are mostly the same for the same n and r, we only consider the smallest number of active Sboxes among them. The complete results are given in long tables Table 9 and Table 10, and some example trails in Appendix E. All results start from the second round since linear operations at the beginning of ALF-n-t have no effect for a differential attack. Thus, the first round can be viewed as a whitening key. As a comparison example, the final ALF-14-0 (where $r^{(14,0)} = 12$, see Table 3) has 52 active Sboxes (Table 10) and the standard AES has 55.

In a more advanced differential attack, two short differential trails can be combined in some way to form an attack covering more rounds. The boomerang attack is a common attack in this style [Wag99, CHP+18]. The target cipher E is decomposed into two parts E_1 and E_2 such that $E = E_2 \circ E_1$. Suppose that there is a r_1-round differential (Δ_i, Δ_o) with probability p for E_1, and a r_2-round differential (∇_i, ∇_o) with probability q for E_2. A $(r_0 + r_1)$-round boomerang attack can be constructed by using these differentials and the probability of the distinguisher is $\Pr\left[E^{-1}\left(E(X) \oplus \nabla_o\right) \oplus E^{-1}\left(E(X \oplus \Delta_i) \oplus \nabla_o\right) = \Delta_i\right] = p^2q^2$. Searching for all advanced differential attacks is a much more complicated task than the above differential analysis. Therefore, we have estimated probabilities of potential advanced differential trails based on our results on the number of active Sboxes. When using two short differential trails, attackers have to consider the success probabilities of both trails at the same time. Since both probabilities depend on the number of active Sboxes, the combined attack activates at least the total number of active Sboxes from both trails. In our estimation, we took the worst case scenario where an attacker could directly concatenate two optimal differential trails without loss to maximise the success probability. For an r-round advanced trail on ALF, we split it into r_1 and r_2 such that $r = r_1 + r_2$. Then, the power of the attack is estimated as the sum of the number of active Sboxes over r_1 and $r_2 + 1$ rounds. The "+1" appears since those results regard the first round as a whitening key, while this is not true for the second trail because attackers cannot access the middle rounds. We checked all possible combinations and list the minimum number of rounds giving at least 22 active Sboxes in Table 5.

Table 5: The minimum number of rounds to prevent differential-like attacks.

n		2	3	4	5	6	7	8	9	10	11	12	13	14	15
$t = 0$	rounds	16	13	11	11	10	10	9	9	9	9	8	8	8	8
	active Sboxes	22	22	22	24	23	24	24	22	24	24	23	23	27	26
$t \neq 0$	rounds	24	20	14	14	12	11	11	11	9	9	9	9	9	8
	active Sboxes	22	22	23	23	23	22	24	22	22	24	23	24	25	23

3.3 Linear analysis

Linear cryptanalysis is yet another common tool that aims to find a linear correlation between the plaintext and the ciphertext. Similarly to the differential analysis, we construct CP models where the constraints for Sboxes are the same and constraints for linear operations are from [SHW+14]. Note that the t-bit exchange is also a linear operation, so its model also follows those basic models of linear operations from previous works. We count the minimum number of active Sboxes, which should be at least 22 for a 128-bit security. The complete results are also given in Table 9 and Table 10. Given the number of active Sboxes k, the complexity of a distinguisher is also $O(2^{6k})$ due to the property of AES Sbox. Comparing to AES, and as an example, ALF-14-0 has 52 active Sboxes while AES has 55.

For a conservative estimate of the security under a potentially more powerful attack, we also consider a differential-linear attack [Lu12] where a differential trail and a linear trail are "concatenated" to form a differential-linear distinguisher, assuming the best trails

are used. The minimum number of active Sboxes for this attack can again be derived from Table 9 and Table 10. Table 6 lists the minimum number of rounds to prevent both linear and differential-linear attacks for each instance of ALF-n-t.

Table 6: The minimum number of rounds to prevent (differential-)linear attacks.

n		2	3	4	5	6	7	8	9	10	11	12	13	14	15
$t=0$	rounds	16	13	11	11	10	10	9	9	9	9	8	8	8	8
	active Sboxes	22	22	22	22	23	23	24	22	24	25	23	23	27	26
$t\neq 0$	rounds	24	19	14	13	12	11	11	11	10	9	9	9	9	8
	active Sboxes	22	23	23	22	22	22	24	23	24	22	23	24	23	23

3.4 Meet-in-the-middle attack

For small domains, the Meet-in-the-Middle (MitM) [DH77] attack is a powerful *key-recovery* attack. For example, let us take ALF-2-0 with an even (for simplicity) number of rounds r. Assume the attacker queries $k \approx r$ pairs of (plaintext, ciphertext), then enumerates all possible $RK[]$s for the first $r/2$ rounds by looping through $2 \cdot r/2$ bytes, and computes intermediate values by encrypting all k plaintexts but only up to $r/2$ rounds. This creates a table having $2^{8\cdot 2\cdot r/2} = 2^{8r}$ records mapping $RK[0..r/2-1]$ to the vector of $2k$ intermediate bytes. A similar table is then created covering the remaining $r - r/2$ rounds by looping through all possible round keys $RK[r/2..r-1]$ and decrypting the ciphertext part of the queries for deriving vectors of $2k$ intermediate bytes.

Obviously, these two tables should have at least one matching record that would reveal the complete set of round keys. When $k = r + c$ for a small $c > 0$ the probability that there is only one matching pair is close to 1. There are many ways of finding a matching record starting from sorting these tables or using a hash table, but we can conservatively assume that time is linear. Thus, for any ALF-n-0 and an even r, the complexity of this attack is $k \cdot 2^{8n\cdot r/2} \approx r \cdot 2^{4nr}$ when $k \approx r$.

To prevent such a key-recovery attack, we need at least 15 rounds for ALF-2-0 and 10 rounds for ALF-3-0, which is satisfied by Table 3. However, the above attack merely recovers $RK[]$s while recovering the master key from these $RK[]$s can still have a time complexity higher than the exhaustive key search.

3.5 Integral attack

An integral attack is one of the well-known analysis methods on the original AES cipher [BS01]. In order to check the security of ALF-n-t under integral attacks, we adopt the division property proposed by Todo [Tod15] which suggests analysing division trails. We again construct CP models and use OR-Tools to find division trails in ALF-n-t. The constraints are more complicated and follow the rules in [SWW17]. For details of the constraints, we refer readers to that work. Once the model is constructed, the solver can search the trail for a certain number of rounds. If there are trails following the rules in [SWW17], integral attacks covering the same number of rounds are also found. Table 7 lists the maximum number of rounds for which we found an integral attack exists. Since integral attacks enumerate bits in plaintexts, the complexity of a meaningful integral attack is always smaller than 2^{8n+t} for any ALF-n-t. Once an integral attack is found to cover all rounds, the cipher is broken no matter what complexity is. Thus, for the design of ALF, we only need to make sure that the final number of rounds in Table 3 is larger than those covered by integral attacks in Table 7, adding a few more rounds as the security margin. For AES, the longest integral distinguisher covers 4 rounds.

Table 7: The number of rounds covered by integral attacks.

n	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$t = 0$	3	3	3	3	4	4	5	5	5	5	5	5	5	5
$t \neq 0$	4	4	4	4	4	5	6	6	6	6	6	6	6	6

3.6 Other attacks

For more advanced cryptanalysis such as zero-correlation and impossible differential, they were proven to have a strong relation to integral attacks [SLR+15]. Since ALF-n-t has a strong resistance to integral attacks, we assume that it is also secure under these advanced analyses. The detailed analysis is left to future work. For algebraic attacks, the AES Sbox used in ALF-n-t raises the algebraic degree by 7 in each round. Since the increment of degrees is delayed by one round for the unaligned t bits, the term with the lowest degree always stays in the unaligned bits. Thus, after n rounds, the potential low-degree term could have degree 7^{n-1}, which is already high enough to prevent algebraic attacks given the number of rounds in each instance. Also, the algebraic approach of writing the entire cipher as a large overdefined system of binary quadratic equations seems fruitless, as the number of rounds used in ALF is larger than in AES, and we know that this approach was not successful against AES.

4 Extending ALF-n-t for format preserving encryption

The cycle-walking transformation is a natural way to turn a length-preserving encryption into a format-preserving encryption. The technique is well-known since at least 2001, e.g. [BR02] "*A second method enciphers a message $m \in \mathcal{M}$ by repeatedly applying the block cipher, starting at m, until one gets back to a point in $\mathcal{M}$.*"

In this section we propose an improved cycle-walking transformation called *cycle-sliding* that is more practical for specific use cases with stricter constraints on the encryption/response time. Since the original cycle-walking can be included as a special case of cycle-sliding, we will start with the description of cycle-sliding and then compare both transformations.

4.1 The cycle-sliding transformation

Let $\mathcal{X}$ be a set of integers in $[0, Q)$, u be the smallest power of 2 such that $2^u \geq Q$, and $E_K(\cdot)$ be a u-bit block cipher with r rounds whose i-th round is denoted by $E_K^i(\cdot), i = 0..r-1$. Cycle-sliding aims to transform E_K to a cipher on $\mathcal{X}$. Given an integer k that divides r, cycle-sliding assures that the intermediate values are in $\mathcal{X}$ after every k rounds by repeating each k rounds until it reaches a value in $\mathcal{X}$. Algorithm 2 depicts this procedure.

Algorithm 2 Cycle-sliding.

```
1: function CycleSliding(E_K, X, k, x) → x'
2:     x' = x
3:     for i = 0 ... r/k − 1 do
4:         do
5:             for j = 0 ... k − 1 do
6:                 x' = E_K^{i·k+j}(x')
7:         while x' ∉ X
8:     return x'
```

Let us show an example. Assume $\mathcal{X} = [0, 9], Q = 10, u = 4$ and $E_K(\cdot)$ is a 4-bit block cipher with $r = 6$ rounds. Also, we take $k = 2$ in this example. Suppose the plaintext "1" is to be encrypted and it yields $E_K^1(E_K^0(1)) = 10$ after the first two rounds. Then we repeat the first two rounds, getting $E_K^1(E_K^0(10)) = 8$ which is a value in $\mathcal{X}$. After seeing this valid value, we continue with the next two rounds, i.e., $E_K^3(E_K^2(8))$, and so on until all round functions are completed. The decryption is done by reverting the procedure.

Since block ciphers are keyed permutations and permutations are made up of cycles, the repetition of a k-round encryption goes through the current cycle to get the next value in $\mathcal{X}$. It guarantees that there is always a valid value and the decryption is always correct. Cycle-sliding also guarantees at least r rounds of E_K, which assures that it cannot be less secure that E_K.

When $k = r$, we obtain a special case of the cycle-sliding which is identical to the cycle-walking. From this perspective, the cycle-sliding allows to move carefully with a smaller "distance" in the cycles. It's like sliding in the cycle, which inspires the name.

4.2 Comparison of cycle-sliding and cycle-walking

In this subsection we compare cycle-sliding and its special case cycle-walking to demonstrate the benefit of using a smaller $k < r$. For simplicity, we call CS (resp. CW) a cipher to implicitly mean the resulting cipher of E_K after the cycle-sliding (resp. cycle-walking) transformation. We also assume that, given a random plaintext $x \in \mathcal{X}$, the probability of getting an encrypted value in $\mathcal{X}$ after k rounds is the same for all valid k, denoted by $p = Q/2^u = Q/2^{\lceil \log_2 Q \rceil} \in (0.5, 1]$, where $Q = |\mathcal{X}|$.

The expected number of rounds for both ciphers CS and CW are the same. For CW, the average invocation of E_K is p^{-1}, and thus the average number of rounds is rp^{-1}. For CS, the average invocation of each k-round step is also p^{-1}, so the average number of rounds is $\frac{r}{k} \cdot kp^{-1} = rp^{-1}$. In this regard CS and CW have the same performance *in average* in terms of the total number of rounds.

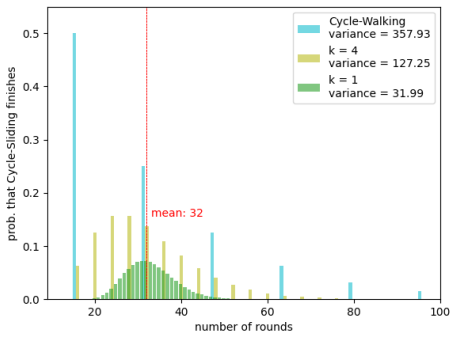

Figure 6: Probability density functions of cycle-sliding and cycle-walking.

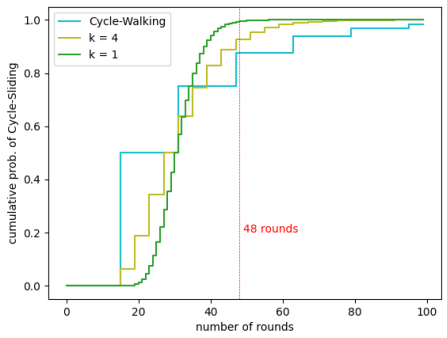

Figure 7: Cumulative distribution functions of cycle-sliding and cycle-walking.

However, CS has a smaller statistical variance of the number of rounds than CW. The probability density function for the number of rounds is computed by

$$\Pr[\text{CW finishes in } R \text{ rounds}] = (1-p)^{R/r-1} \cdot p \quad \text{if } r \mid R, \quad \text{otherwise } 0,$$

i.e., R/r failed checks and 1 successful check in the end. Similarly, CS has R/k checks and exactly r/k of them should be successful:

$$\Pr[\text{CS finishes in } R \text{ rounds}] = \binom{r/k-1}{R/k-1} \cdot (1-p)^{R/k-r/k} \cdot p^{r/k}, \quad \text{if } k \mid R, \quad \text{otherwise } 0.$$

Figure 6 plots the probability density functions for CS and CW given $p = 0.5$, $r = 16$, and $k \in \{1, 4\}$, where the variance decreases for a smaller k. And by Figure 7 the corresponding cumulative distribution function demonstrates that a smaller k leads to a faster convergence to 1. This also means that the encryption time in CS is closer to a

constant than CW's time, which can be beneficial in use cases where the response time is critical.

On the other hand, a larger k could benefit from the CPU pipelining, where the compiler can unroll the loop for k rounds allowing the CPU to form a better pipeline and run faster. Also, the routine to check whether $x \in \mathcal{X}$ could also be costly in certain cases. This is a clear trade-off between the mean and variance of the *total encryption time*.

ALF-n-t for FPE. After a number of considerations we decided to adopt $k = 2$ for all ALF instances where CS is relevant. This requires that the number of rounds r must be an even number, and the choice provides a good trade-off for the running time.

5 Software evaluation

For the purpose of software evaluation, we have created a set of simple performance test scenarios, such as a 'Credit card' encryption, 'IPv6', 'IMSI' encryption. For each case the base version of ALF is derived, adding 'n' and 'b' to denote a (**n**on-)**b**inary case, respectively. Our implementation is also open-sourced and available at the same place as the one in Section 3.

In our implementation we made a split of ALF API for the full encrypt-decrypt sequence by the methods `EngineInit`, `KeyInit`, `TweakInit`, `Encrypt`, `PrepareDecrypt`, `Decrypt`, where the method `EngineInit` selects the right ALF instance, initialises internal values, shuffles, etc. This separation is implemented to ensure high processing speed in various use cases. For example, the key and case configuration may be pre-initialised, while the tweak and the actual data can change more frequently.

Table 8: Performance measurements for a full cycle (encrypt-decrypt, including key setup etc). The two values 'max/avr' of each performance point correspond to the maximum and average speeds (in Msps), respectively.

Test scenario	ALF (variant)	FF1	FF3
C1. 'Credit card', $Q = 10^{16}$	28/25 (ALF-6-6n)	2.34/1.66	20.3/14.4
C2. 'IPv6', $Q = (2^{16})^6$	36/30 (ALF-12-0b)	1.53/1.06	9.35/7.06
C3. 'IMSI', $Q = 10^{11}$	25/22 (ALF-4-5n)	2.09/1.48	15.44/11.44

The speed is given in terms of *Million symbols per second* (Msps), and all tests were performed on a real cloud environment equipped with various CPUs (such as AMD EPYC 9354@3.2GHz, Intel(R) Xeon(R) Gold 6254 CPU @ 3.10GHz), using a single-thread process. We provide the maximum and the average speed values for each test point. Since the performance may also depend on the input plaintext P, we then run in a loop the encryption $Enc(Enc(Enc(P))\ldots)$, and count how many of these nested encryptions were completed in a given time slot – this guarantees pseudo-random input plaintext for each call of `Encrypt`. Similar technique is used for the decryption and full-cycle performance tests where we again make sure that the input is pseudo-random for each call. The results of the performance evaluation are given in Table 8, and more details can be found in Table 13 of Appendix F.6.

For comparison with other FPE implementations, we took the existing library[9] with FF1/FF3. We compare ALF versus FF1 and FF3, and measure the speed of a full cycle, i.e., that includes all API stages from `EngineInit` to `Encrypt` and `Decrypt`. This compares FPE solutions when the engine is used for random cases and inputs at each call.

[9]The code used in [DHHV21]: https://github.com/comForte/Format-Preserving-Encryption.

6 Conclusions

We introduced the ALF cipher family, designed for FPE. The key strategy was to use the AES-NI instructions in the design in order to achieve good performance while supporting a 128-bit security level.

Future work would be to further analyse the security of the ALF family, providing an even stronger foundation for its claimed security, or detecting possible security weaknesses through new types of attacks not considered in this paper.

Acknowledgements

We thank anonymous reviewers for providing constructive feedback. This work was supported by the Swedish Foundation for Strategic Research (Grants No. SM22-0050) and the ELLIIT program.

References

[BHT16] Mihir Bellare, Viet Tung Hoang, and Stefano Tessaro. Message-recovery attacks on feistel-based format preserving encryption. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 444–455, 2016.

[BR99] Mihir Bellare and Phillip Rogaway. On the construction of variable-input-length ciphers. In Lars R. Knudsen, editor, *FSE'99*, volume 1636 of *LNCS*, pages 231–244. Springer, Berlin, Heidelberg, March 1999.

[BR02] John Black and Phillip Rogaway. Ciphers with arbitrary finite domains. In Bart Preneel, editor, *CT-RSA 2002*, volume 2271 of *LNCS*, pages 114–130. Springer, Berlin, Heidelberg, February 2002. IACR ePrint version – https://eprint.iacr.org/2001/012.

[BRRS09] Mihir Bellare, Thomas Ristenpart, Phillip Rogaway, and Till Stegers. Format-preserving encryption. In Michael J. Jacobson, Jr., Vincent Rijmen, and Reihaneh Safavi-Naini, editors, *SAC 2009*, volume 5867 of *LNCS*, pages 295–312. Springer, Berlin, Heidelberg, August 2009.

[BRS10] Mihir Bellare, Phillip Rogaway, and Terence Spies. The ffx mode of operation for format-preserving encryption. *NIST submission*, 20(19):1–18, 2010.

[BS01] Alex Biryukov and Adi Shamir. Structural cryptanalysis of SASAS. In Birgit Pfitzmann, editor, *EUROCRYPT 2001*, volume 2045 of *LNCS*, pages 394–405. Springer, Berlin, Heidelberg, May 2001.

[BSV07] Thomas Baignères, Jacques Stern, and Serge Vaudenay. Linear cryptanalysis of non binary ciphers. In Carlisle M. Adams, Ali Miri, and Michael J. Wiener, editors, *SAC 2007*, volume 4876 of *LNCS*, pages 184–211. Springer, Berlin, Heidelberg, August 2007.

[Bus] Christopher Gerhard Busch. Format preserving encryption. https://patents.google.com/patent/US10185836B2/en.

[CHB21] Paul Crowley, Nathan Huckleberry, and Eric Biggers. Length-preserving encryption with HCTR2. Cryptology ePrint Archive, Report 2021/1441, 2021.

[CHP+18] Carlos Cid, Tao Huang, Thomas Peyrin, Yu Sasaki, and Ling Song. Boomerang connectivity table: A new cryptanalysis tool. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part II*, volume 10821 of *LNCS*, pages 683–714. Springer, Cham, April / May 2018.

[DH77] W. Diffie and M.E. Hellman. Special feature exhaustive cryptanalysis of the nbs data encryption standard. *Computer*, 10(6):74–84, 1977.

[DHHV21] F. Betül Durak, Henning Horst, Michael Horst, and Serge Vaudenay. FAST: Secure and high performance format-preserving encryption and tokenization. In Mehdi Tibouchi and Huaxiong Wang, editors, *ASIACRYPT 2021, Part III*, volume 13092 of *LNCS*, pages 465–489. Springer, Cham, December 2021.

[DR02] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard (Information Security and Cryptography)*. Springer, 1 edition, 2002.

[DV17] F. Betül Durak and Serge Vaudenay. Breaking the FF3 format-preserving encryption standard over small domains. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part II*, volume 10402 of *LNCS*, pages 679–707. Springer, Cham, August 2017.

[Dwo25] M. J. Dworkin. Recommendation for block cipher modes of operation: Methods for format-preserving encryption., 2025. Technical Report NIST Special Publication 800-38G Revision 1 (Second Public Draft), National Institute of Standards and Technology, Second Public Draft released February 3, 2025.

[GLP06] Louis Granboulan, Éric Levieil, and Gilles Piret. Pseudorandom permutation families over Abelian groups. In Matthew J. B. Robshaw, editor, *FSE 2006*, volume 4047 of *LNCS*, pages 57–77. Springer, Berlin, Heidelberg, March 2006.

[HMT19] Viet Tung Hoang, David Miller, and Ni Trieu. Attacks only get better: How to break FF3 on large domains. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part II*, volume 11477 of *LNCS*, pages 85–116. Springer, Cham, May 2019.

[HTT18] Viet Tung Hoang, Stefano Tessaro, and Ni Trieu. The curse of small domains: New attacks on format-preserving encryption. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 221–251. Springer, Cham, August 2018.

[Lem19] Daniel Lemire. Fast random integer generation in an interval. *ACM Trans. Model. Comput. Simul.*, 29(1), January 2019.

[LRW11] Moses Liskov, Ronald L. Rivest, and David Wagner. Tweakable block ciphers. *Journal of Cryptology*, 24(3):588–613, July 2011.

[Lu12] Jiqiang Lu. A methodology for differential-linear cryptanalysis and its applications - (extended abstract). In Anne Canteaut, editor, *FSE 2012*, volume 7549 of *LNCS*, pages 69–89. Springer, Berlin, Heidelberg, March 2012.

[MSPa] Luther W. Martin, Terence Spies, and Matthew J. Pauker. Data processing systems with format-preserving encryption and decryption engines. https://patents.google.com/patent/US8855296B2/en.

[MSPb] Luther W. Martin, Terence Spies, and Matthew J. Pauker. Format preserving encryption methods for data strings with constraints. https://patents.google.com/patent/US8938067B2/en.

[NFI24] Yuto Nakano, Kazuhide Fukushima, and Takanori Isobe. Encryption algorithm Rocca-S, 2024. https://datatracker.ietf.org/doc/draft-nakano-rocca-s/.

[PD] Laurent Perron and Frédéric Didier. CP-SAT (v9.9). https://developers.google.com/optimization/cp/.

[Piv] Eugene Pivovarov. Fast format-preserving encryption for variable length data. https://patents.google.com/patent/US10951392B2/en.

[SGL+17] Siwei Sun, David Gerault, Pascal Lafourcade, Qianqian Yang, Yosuke Todo, Kexin Qiao, and Lei Hu. Analysis of AES, SKINNY, and others with constraint programming. *IACR Trans. Symm. Cryptol.*, 2017(1):281–306, 2017.

[SHW+14] Siwei Sun, Lei Hu, Peng Wang, Kexin Qiao, Xiaoshuang Ma, and Ling Song. Automatic security evaluation and (related-key) differential characteristic search: Application to SIMON, PRESENT, LBlock, DES(L) and other bit-oriented block ciphers. In Palash Sarkar and Tetsu Iwata, editors, *ASIACRYPT 2014, Part I*, volume 8873 of *LNCS*, pages 158–178. Springer, Berlin, Heidelberg, December 2014.

[SLN+21] Kosei Sakamoto, Fukang Liu, Yuto Nakano, Shinsaku Kiyomoto, and Takanori Isobe. Rocca: An efficient AES-based encryption scheme for beyond 5g. *IACR Trans. Symm. Cryptol.*, 2021(2):1–30, 2021.

[SLR+15] Bing Sun, Zhiqiang Liu, Vincent Rijmen, Ruilin Li, Lei Cheng, Qingju Wang, Hoda AlKhzaimi, and Chao Li. Links among impossible differential, integral and zero correlation linear cryptanalysis. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part I*, volume 9215 of *LNCS*, pages 95–115. Springer, Berlin, Heidelberg, August 2015.

[Sta17] William Stallings. *Cryptography and Network Security: Principles and Practice*. Pearson, 7th edition, 2017.

[SWW17] Ling Sun, Wei Wang, and Meiqin Wang. Automatic search of bit-based division property for ARX ciphers and word-based division property. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part I*, volume 10624 of *LNCS*, pages 128–157. Springer, Cham, December 2017.

[Tod15] Yosuke Todo. Structural evaluation by generalized integral property. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 287–314. Springer, Berlin, Heidelberg, April 2015.

[Wag99] David Wagner. The boomerang attack. In Lars R. Knudsen, editor, *FSE'99*, volume 1636 of *LNCS*, pages 156–170. Springer, Berlin, Heidelberg, March 1999.

[WFW05] Peng Wang, Dengguo Feng, and Wenling Wu. HCTR: A variable-input-length enciphering mode. In Dengguo Feng, Dongdai Lin, and Moti Yung, editors, *Information Security and Cryptology, First SKLOIS Conference, CISC 2005, Beijing, China, December 15-17, 2005, Proceedings*, volume 3822 of *Lecture Notes in Computer Science*, pages 175–188. Springer, 2005.

[WMEJ25] Dachao Wang, Alexander Maximov, Patrik Ekdahl, and Thomas Johansson. A new stand-alone mac construct called smac. *IACR Transactions on Symmetric Cryptology*, 2025(1):5–43, Mar. 2025.

[WT86] A. F. Webster and Stafford E. Tavares. On the design of S-boxes (impromptu talk). In Hugh C. Williams, editor, *CRYPTO'85*, volume 218 of *LNCS*, pages 523–534. Springer, Berlin, Heidelberg, August 1986.

A Engine mixing

The design of ALF-n-t has a useful property that enables running multiple instances in parallel while sharing the same state registers. As mentioned before, ALF-n-t occupies only m bytes in the 16-byte state X. The core idea of parallelisation is to reuse the remaining $16 - m$ bytes for other encryption engines. As a simple example, four ALF-4-0 can run in parallel in the same register by taking one column each, while the cost is roughly the same as running only one engine. This becomes very useful when many IPv4 addresses need to be encrypted. Furthermore, the master keys to these four parallel engines can be independent. It is also possible to combine engines with different (n, t) configurations if the 16-byte state allows. Given 512-bit registers in AVX-512, even more independent engines can run in parallel and achieve a high throughput.

Combining two engines:

$$m = \hat{m} + \tilde{m} \tag{12}$$
$$\alpha = \hat{\alpha}\ \texttt{AND}\ \texttt{shadd}(\tilde{\alpha}, \hat{m}) \tag{13}$$
$$\beta = \hat{\beta}\ \texttt{AND}\ \texttt{shadd}(\tilde{\beta}, \hat{m}) \tag{14}$$
$$\alpha' = \hat{\alpha'}\ \texttt{AND}\ \texttt{shadd}(\tilde{\alpha'}, \hat{m}) \tag{15}$$
$$\beta' = \hat{\beta'}\ \texttt{AND}\ \texttt{shadd}(\tilde{\beta'}, \hat{m}) \tag{16}$$
$$\pi = \hat{\pi}\ \texttt{AND}\ \texttt{shadd}(\tilde{\pi}, \hat{m}) \tag{17}$$
$$\rho = \hat{\rho}\ \texttt{AND}\ \texttt{shadd}(\tilde{\rho}, \hat{m}) \tag{18}$$
$$M = M^{(\hat{t})} \oplus (M^{(\tilde{t})} \ll 8\hat{m}) \tag{19}$$
$$RK[i] = \hat{RK}[i] \oplus (\tilde{RK}[i] \ll 8\hat{m}) \tag{20}$$
$$A = \hat{A} \oplus (\tilde{A} \ll 8\hat{m}) \tag{21}$$
$$B = (\hat{B} \oplus (\tilde{B} \ll 8\hat{m}))\ \texttt{AND}\ M \tag{22}$$

Let us have two engines with parameters $(\hat{n}, \hat{t})$ and $(\tilde{n}, \tilde{t})$, each having their own sets of shuffles and round keys. We want the first engine $(\hat{n}, \hat{t})$ to execute on the first $\hat{m}$ bytes of the 16-byte state, while the second engine $(\tilde{n}, \tilde{t})$ would occupy the next $\tilde{m}$ bytes right after the first $\hat{m}$ bytes In total, they would occupy $\hat{m} + \tilde{m}$ bytes of the state. Obviously, the total occupied bytes must be at most 16 bytes. The encryption running on the combined state is the same as Algorithm 1 but takes the parameters as derived by Eq. (12)..Eq. (19).

`AND` is a bitwise AND operation on two shuffles. $\texttt{shadd}(\alpha, m)$ is a transformation applying to a shuffle α and integer m, which increases all indices from the range $[0, 15]$ in α by m and shifts the whole shuffle to the left by m bytes while shifting in (-1)s. Formally, it is defined as

$$\gamma = \texttt{shadd}(\alpha, m), \text{ where } \gamma_i = \begin{cases} -1 & i \in [0, m-1] \text{ or } \alpha_{i-m} = -1 \\ m + \alpha_{i-m} & \alpha_{i-m} \in [0, 15] \end{cases}$$

A similar technique applies to the unaligned bits. Recall that the t bits are stored in a different register than n bytes. Let E be the register containing $\hat{t}$ bits in the first byte of E_0. We reuse the same register and put the $\tilde{t}$ bits in the first byte of $E_{\hat{m}}$. The new ρ shuffle is then derived from $\hat{\rho}$ and $\tilde{\rho}$ by Eq. (18). To obtain a combined parity function, we leverage Eq. (3). Since it computes four parities at the same time, only a combined mask M is needed Eq. (19). Thus the t-bit exchange in Algorithm 1 still works for two engines but takes combined shuffles and a combined mask value. Round keys RKs are combined as well in a similar fashion by Eq. (20). The constants A and B for the derivation of the decryption round keys are given by Eq. (21) and Eq. (22).

When two engines have distinct number of rounds, the combined implementation could be more challenging though still possible. The algorithm has to pause in the loop when reaching the lowest number of rounds, and extract the first ciphertext. Then, it resumes the loop until it gets the next ciphertext, and so on.

Note that the combined parameters can be computed only once for a static use case. We also emphasise that the above methods work in the same way for more than two engines, maximum four parallel engines can be combined in the same 16-byte state registers since there are 4 columns. However, with AVX-512 one can combine up to 16 parallel engines using 512-bit registers.

B Decryptability of ALF-n-t and constraints on (α, β)

In order to prove that the encryption of ALF-n-0 can (theoretically) be decrypted we need to show that each step of the ALF round is invertible. The permutations π and SR, as well as SB and RK are invertible, and thus we only need to show that the linear transformation $\alpha \to MC \to \beta \to XOR$, as depicted on Figure 2 (left), followed by the n-byte truncation produces a linear matrix $L_E^{(n,0)}$ that has a full rank for every n. Below are these matrices where each cell is an element of the Rijndael field $GF(2^8)$.

```
n=2  -n=3-  --n=4--  ---n=5---  ----n=6----  ---n=7-------  ---n=8---------
3 2  1 2 0  2 3 1 1  2 1 1 1 2  3 2 3 2 0 0  3 1 2 0 0 0 0  2 3 1 1 0 0 0 0
2 3  2 3 2  1 2 3 1  1 0 3 1 2  2 3 2 3 0 0  1 2 3 1 3 1 1  1 2 3 1 0 0 0 0
     2 0 3  1 1 2 3  1 3 2 3 2  1 1 2 3 1 1  0 3 1 2 0 0 0  1 1 2 3 0 0 0 0
            3 1 1 2  3 3 1 2 2  3 1 1 2 3 1  2 3 2 3 0 0 0  3 1 1 2 0 0 0 0
                     0 1 0 0 1  1 1 2 3 2 3  1 2 3 1 2 3 1  0 0 0 0 2 3 1 1
                                3 1 1 2 1 2  1 2 3 1 1 2 3  0 0 0 0 1 2 3 1
                                             1 2 3 1 1 1 2  0 0 0 0 1 1 2 3
                                                            0 0 0 0 3 1 1 2

---n=9-----------  ---n=10------------  ---n=11--------------  ---n=12----------------
2 1 1 1 0 0 0 0 2  3 2 3 2 0 0 0 0 0 0  3 1 2 0 0 0 0 0 0 0 0  2 3 1 1 0 0 0 0 0 0 0 0
1 0 3 1 0 0 0 0 2  2 3 2 3 0 0 0 0 0 0  1 2 3 1 0 0 0 0 3 1 1  1 2 3 1 0 0 0 0 0 0 0 0
1 3 2 3 0 0 0 0 2  1 1 2 3 0 0 0 0 0 1  0 3 1 2 0 0 0 0 0 0 0  1 1 2 3 0 0 0 0 0 0 0 0
3 3 1 2 0 0 0 0 2  3 1 1 2 0 0 0 0 3 1  2 3 2 3 0 0 0 0 0 0 0  3 1 1 2 0 0 0 0 0 0 0 0
0 0 0 0 2 3 1 1 0  0 0 0 0 2 3 1 1 0 0  0 0 0 0 2 3 1 1 0 0 0  0 0 0 0 2 3 1 1 0 0 0 0
0 0 0 0 1 2 3 1 0  0 0 0 0 1 2 3 1 0 0  0 0 0 0 1 2 3 1 0 0 0  0 0 0 0 1 2 3 1 0 0 0 0
0 0 0 0 1 1 2 3 0  0 0 0 0 1 1 2 3 0 0  0 0 0 0 1 1 2 3 0 0 0  0 0 0 0 1 1 2 3 0 0 0 0
0 0 0 0 3 1 1 2 0  0 0 0 0 3 1 1 2 0 0  0 0 0 0 3 1 1 2 0 0 0  0 0 0 0 3 1 1 2 0 0 0 0
0 1 0 0 0 0 0 0 1  1 1 2 3 0 0 0 0 2 3  1 2 3 1 0 0 0 0 2 3 1  0 0 0 0 0 0 0 0 2 3 1 1
                   3 1 1 2 0 0 0 0 1 2  1 2 3 1 0 0 0 0 1 2 3  0 0 0 0 0 0 0 0 1 2 3 1
                                        1 2 3 1 0 0 0 0 1 1 2  0 0 0 0 0 0 0 0 1 1 2 3
                                                               0 0 0 0 0 0 0 0 3 1 1 2

---n=13------------------  ---n=14--------------------  ---n=15----------------------
2 1 1 1 0 0 0 0 0 0 0 0 2  3 2 3 2 0 0 0 0 0 0 0 0 0 0  3 1 2 0 0 0 0 0 0 0 0 0 0 0
1 0 3 1 0 0 0 0 0 0 0 0 2  2 3 2 3 0 0 0 0 0 0 0 0 0 0  1 2 3 1 0 0 0 0 0 0 0 0 3 1 1
1 3 2 3 0 0 0 0 0 0 0 0 2  1 1 2 3 0 0 0 0 0 0 0 0 1 1  0 3 1 2 0 0 0 0 0 0 0 0 0 0 0
3 3 1 2 0 0 0 0 0 0 0 0 2  3 1 1 2 0 0 0 0 0 0 0 0 3 1  2 3 2 3 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 2 3 1 1 2 3 1 1 0  0 0 0 0 2 3 1 1 0 0 0 0 0 0  0 0 0 0 2 3 1 1 0 0 0 0 0 0 0
0 0 0 0 1 2 3 1 1 2 3 1 0  0 0 0 0 1 2 3 1 0 0 0 0 0 0  0 0 0 0 1 2 3 1 0 0 0 0 0 0 0
0 0 0 0 1 1 2 3 1 1 2 3 0  0 0 0 0 1 1 2 3 0 0 0 0 0 0  0 0 0 0 1 1 2 3 0 0 0 0 0 0 0
0 0 0 0 3 1 1 2 3 1 1 2 0  0 0 0 0 3 1 1 2 0 0 0 0 0 0  0 0 0 0 3 1 1 2 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 2 3 1 1 0  0 0 0 0 0 0 0 0 2 3 1 1 0 0  0 0 0 0 0 0 0 0 2 3 1 1 0 0 0
0 0 0 0 0 0 0 0 1 2 3 1 0  0 0 0 0 0 0 0 0 1 2 3 1 0 0  0 0 0 0 0 0 0 0 1 2 3 1 0 0 0
0 0 0 0 0 0 0 0 1 1 2 3 0  0 0 0 0 0 0 0 0 1 1 2 3 0 0  0 0 0 0 0 0 0 0 1 1 2 3 0 0 0
0 0 0 0 0 0 0 0 3 1 1 2 0  0 0 0 0 0 0 0 0 3 1 1 2 0 0  0 0 0 0 0 0 0 0 3 1 1 2 0 0 0
0 1 0 0 0 0 0 0 0 0 0 0 1  1 1 2 3 0 0 0 0 0 0 0 0 2 3  1 2 3 1 0 0 0 0 0 0 0 0 2 3 1
                           3 1 1 2 0 0 0 0 0 0 0 0 1 2  1 2 3 1 0 0 0 0 0 0 0 0 1 2 3
                                                        1 2 3 1 0 0 0 0 0 0 0 0 1 1 2
```

We have checked that the above matrices have a full rank, thus resulting in a 1-to-1 mapping of the whole encryption process of ALF-n-0. In order to prove the correctness of the decryption process, we have derived similar matrices $L_D^{(n,0)}$ for the transformation $\alpha' \to MC' \to \beta' \to XOR$, and verified that $L_D^{(n,0)} \cdot L_E^{(n,0)} = \mathbf{I}$. The matrices $L_D^{(n,0)}$ are not given here but they can be easily derived by either constructing from the linear transformation sequence, as we did for the verification, or simply computing by $L_D^{(n,0)} = (L_E^{(n,0)})^{-1}$.

When $t \neq 0$, we can similarly derive a bit-truncated *binary* matrices represented over $GF(2)$, then a similar verification procedure applies. The only difference between $L_E^{(n,t)}$ and $L_E^{(n,0)}$ is the addition of t binary columns (ρ) and t rows (`parity`) that are in fact the

result of the t-bit exchange procedure followed by the t-bit truncation. As an example, we give $L_E^{(n,t)}$ and $L_D^{(n,t)}$ for ALF-3-5:

```
=== matrix L_E^(3,5) ===                    === matrix L_D^(3,5) ===
10000000 | 00000001 | 00000000 | 10000      00000010 | 10000011 | 10000010 | 10000
01000000 | 10000001 | 00000000 | 01000      00000011 | 11000010 | 01000011 | 01000
00100000 | 01000000 | 00000000 | 00100      10000001 | 11100001 | 10100001 | 00100
00010000 | 00100001 | 00000000 | 00010      01000010 | 01110011 | 01010010 | 00010
00001000 | 00010001 | 00000000 | 00001      00100011 | 00111010 | 00101011 | 00001
00000100 | 00001000 | 00000000 | 00000      00010101 | 00011001 | 00010001 | 00000
00000010 | 00000100 | 00000000 | 00000      00001010 | 00001100 | 00001000 | 00000
00000001 | 00000010 | 00000000 | 00000      00000101 | 00000110 | 00000100 | 00000
-----------------------------------         -----------------------------------
00000001 | 10000001 | 00000001 | 00000      00000001 | 10000001 | 00000001 | 00000
10000001 | 11000001 | 10000001 | 00000      10000001 | 11000001 | 10000001 | 00000
01000000 | 01100000 | 01000000 | 00000      01000000 | 01100000 | 01000000 | 00000
00100001 | 00110001 | 00100001 | 00000      00100001 | 00110001 | 00100001 | 00000
00010001 | 00011001 | 00010001 | 00000      00010001 | 00011001 | 00010001 | 00000
00001000 | 00001100 | 00001000 | 00000      00001000 | 00001100 | 00001000 | 00000
00000100 | 00000110 | 00000100 | 00000      00000100 | 00000110 | 00000100 | 00000
00000010 | 00000011 | 00000010 | 00000      00000010 | 00000011 | 00000010 | 00000
-----------------------------------         -----------------------------------
00000001 | 00000000 | 10000001 | 10000      10000011 | 10000010 | 00000011 | 10000
10000001 | 00000000 | 11000001 | 01000      11000010 | 01000011 | 10000010 | 01000
01000000 | 00000000 | 01100000 | 00100      11100001 | 10100001 | 11000001 | 00100
00100001 | 00000000 | 00110001 | 00010      01110011 | 01010010 | 01100011 | 00010
00010001 | 00000000 | 00011001 | 00001      00111010 | 00101011 | 00110010 | 00001
00001000 | 00000000 | 00001100 | 00000      00011001 | 00010001 | 00011101 | 00000
00000100 | 00000000 | 00000110 | 00000      00001100 | 00001000 | 00001110 | 00000
00000010 | 00000000 | 00000011 | 00000      00000110 | 00000100 | 00000111 | 00000
-----------------------------------         -----------------------------------
10000000 | 10000000 | 10000000 | 10000      10000000 | 10000000 | 10000000 | 10000
01000000 | 01000000 | 01000000 | 01000      01000000 | 01000000 | 01000000 | 01000
00100000 | 00100000 | 00100000 | 00100      00100000 | 00100000 | 00100000 | 00100
00010000 | 00010000 | 00010000 | 00010      00010000 | 00010000 | 00010000 | 00010
00001000 | 00001000 | 00001000 | 00001      00001000 | 00001000 | 00001000 | 00001
```

We have verified the decryptability and correctness of ALF-n-t for all combinations of (n, t). Moreover, our reference implementation with an encryption-decryption cycle gives the correct decrypted result under an extensive set of tests, which serves as additional evidence of the algorithm's decryptability and correctness.

The effect of the parity function applied after MC' does not always result in addition of four first bytes from X to E, when $n \geq 4$. Here is one such example for ALF-6-8 (although not allowed combination since here $t = 8$ and ALF-7-0 should be used instead), given over $GF(2^8)$. The last row of the decryption matrix reveals that the effect of the `parity` function after MC' effectively adds only two bytes to E:

```
== matrix L_E^(6,8) ==          == matrix L_D^(6,8) ==
 3  2  3  2  0  0  1             f  a  e  b  d  9  1
 2  3  2  3  0  0  1             8  f  9  e  b  d  1
 1  1  2  3  1  1  0             d  9  e  b  d  9  0
 3  1  1  2  3  1  0             b  d  9  e  b  d  0
 1  1  2  3  2  3  1             1  1  3  2  3  2  1
 3  1  1  2  1  2  1             1  1  2  3  2  3  1
 1  1  1  1  0  0  1             1  1  0  0  0  0  1
```

Note that the last column represents the ρ shuffle, which is identical for both the encryption and decryption directions – this was one of the search strategy on purpose to reduce the number of parameters, while not all valid combinations $(\alpha, \beta, \alpha', \beta')$ have this property.

In a search for transformation shuffles (α, β) it is not enough to verify that $L_E^{(n,0)}$ is of full rank, and one of the constraints is that there must exist a corresponding pair (α', β') such that $L_D^{(n,0)} \cdot L_E^{(n,0)} = \mathtt{I}$. Here is one of such examples where (α, β) results in a full-rank linear transformation but there is no corresponding (α', β') for decryption:

```
n=3, t=0
alpha = (0, 1, 2, -1, -1, ...)
beta  = (-1, -1, ...)
L_E:=  2  3  1
       1  2  3
       1  1  2
rank(L_E) = 24 (bits)
```

Another constraint was the full diffusion round count that we targeted to be at most 2 for all ns; this example explains why the search for MC-bypass methods was done up to $n \leq 8$ rather than $n \leq 4$. Let us take a simple solution for $n = 3$ as $\alpha = (0, 1, 2, -1, ...)$ and $\beta = (3, 3, 3, -1, ...)$, the corresponding decryption pair exists: $(\alpha', \beta') = (\alpha, \beta)$, i.e. it is a symmetric solution. The encryption linear matrix is

```
L_E := 1  2  0
       2  3  2
       2  0  3
```

with the full diffusion in 2 rounds (although, there exist a solution $(\alpha, \beta, \alpha', \beta')$ where one round reaches the full diffusion). Now if we try to extend this transformation to $n = 12 + 3$ we get the following situation:

```
alpha = ( 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,10,11,12,13,14,-1)
beta  = (-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,15,15,15,-1)
L_E :=
        2  3  1  1  0  0  0  0  0  0  0  0  0  0  0
        1  2  3  1  0  0  0  0  0  0  0  0  0  0  0
        1  1  2  3  0  0  0  0  0  0  0  0  0  0  0
        3  1  1  2  0  0  0  0  0  0  0  0  0  0  0
        0  0  0  0  2  3  1  1  0  0  0  0  0  0  0
        0  0  0  0  1  2  3  1  0  0  0  0  0  0  0
        0  0  0  0  1  1  2  3  0  0  0  0  0  0  0
        0  0  0  0  3  1  1  2  0  0  0  0  0  0  0
        0  0  0  0  0  0  0  0  2  3  1  1  0  0  0
        0  0  0  0  0  0  0  0  1  2  3  1  0  0  0
        0  0  0  0  0  0  0  0  1  1  2  3  0  0  0
        0  0  0  0  0  0  0  0  3  1  1  2  0  0  0
        0  0  0  0  0  0  0  0  0  0  0  0  1  2  0
        0  0  0  0  0  0  0  0  0  0  0  0  2  3  2
        0  0  0  0  0  0  0  0  0  0  0  0  2  0  3
```

It is easy to see that the last 3 bytes of the input will be mixed into the last 3 bytes of the output after $\alpha \to MC \to \beta \to XOR$, the next such transformation is applied after π, SR, SB operations. However, whatever permutation $SR \circ \pi$ is applied to the previous output, the last three bytes cannot be distributed over the four columns of the state, thus the second round of $\alpha \to MC \to \beta \to XOR$ cannot succeed in reaching a full diffusion as well, and thus at least 3 rounds are needed. The full diffusion in 2 rounds for $n = 15$ can only be achieved when we would use a suitable MC-bypass method derived from an appropriate two-mixed-columns case $n = 7$. A similar constraint appears for other ns as well.

There is a special case $n = 13$ where we found none of appropriate $n = 5$ cases that can be extended to $n = 8 + 5$ with the full diffusion in 2 rounds; therefore, an additional two-columns-XOR was required to reach a 2-round full diffusion for $n = 13$. However, that XOR is eventually "for free" as it becomes a part of $\beta^{(13)}$.

C Precomputed vectors for ALF-n-t

This section contains precomputed vectors that may be needed for an efficient implementation of ALF-n-t and study a concrete instance, although most of these vectors could also be computed at runtime following the description given in Section 2.

For encryption we need $\sigma = SR' \circ \alpha \circ \pi$ and β (`encSigma` and `encBeta`), and for decryption we need $\sigma' = SR \circ \alpha' \circ \pi', \beta', \tau' = SR \circ \pi'$ (`decSigma`, `decBeta` and `decTau`). When $t \neq 0$ an additional vector ρ is also needed (`Rho`). For the conversion of encryption round keys to the decryption keys we need β, α', A (`encBeta`, `decAlpha` and `constA`). The truncation vector M and the compensation constant B can be computed at runtime as per Eq. (4) and Eq. (11), respectively.

```
n=2: Rounds when t=0: 20, when t>0: 28
    encAlpha = ( 0, 1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1)
    encBeta  = ( 2, 3,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1)
    encSigma = ( 0,-1,-1,-1,-1, 1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1)
    decAlpha = ( 0, 1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1)
    decBeta  = ( 2, 3,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1)
    decSigma = ( 0,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, 1,-1,-1)
    decTau   = ( 0,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, 1,-1,-1)
    Rho      = ( 0, 0,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1)
    constA   = (99,99,82,82, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
n=3: Rounds when t=0: 16, when t>0: 24
    encAlpha = ( 0, 1, 2,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1)
    encBeta  = ( 3, 3, 3,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1)
    encSigma = ( 0,-1,-1,-1,-1, 2,-1,-1,-1,-1, 1,-1,-1,-1,-1,-1)
    decAlpha = ( 0, 1, 2,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1)
    decBeta  = ( 3, 3, 3,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1)
    decSigma = ( 0,-1,-1,-1,-1,-1,-1,-1,-1,-1, 1,-1,-1, 2,-1,-1)
    decTau   = ( 0,-1,-1,-1,-1,-1,-1,-1,-1,-1, 1,-1,-1, 2,-1,-1)
    Rho      = ( 0,-1, 0,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1)
    constA   = (-91,-91,99,82, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
n=4: Rounds when t=0: 14, when t>0: 18
    encAlpha = ( 0, 1, 2, 3,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1)
    encBeta  = (-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1)
    encSigma = ( 0,-1,-1,-1,-1, 1,-1,-1,-1,-1, 2,-1,-1,-1,-1, 3)
    decAlpha = ( 0, 1, 2, 3,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1)
    decBeta  = (-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1)
    decSigma = ( 0,-1,-1,-1,-1,-1,-1, 3,-1,-1, 2,-1,-1, 1,-1,-1)
    decTau   = ( 0,-1,-1,-1,-1,-1,-1, 3,-1,-1, 2,-1,-1, 1,-1,-1)
    Rho      = ( 0, 0, 0, 0,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1)
    constA   = ( 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
n=5: Rounds when t=0: 14, when t>0: 18
    encAlpha = ( 0, 1, 2, 3, 4, 1,-1, 1,-1,-1,-1,-1,-1,-1,-1,-1)
    encBeta  = ( 4, 4, 4, 4, 7,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1)
    encSigma = ( 1,-1,-1, 0, 3, 0,-1,-1,-1, 0, 4,-1,-1,-1,-1, 2)
    decAlpha = ( 0, 1, 2, 3, 4,-1, 4,-1,-1,-1,-1,-1,-1,-1,-1,-1)
    decBeta  = ( 5, 5, 7, 5, 1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1)
    decSigma = ( 1,-1,-1,-1, 2,-1,-1, 4,-1,-1, 3,-1,-1, 0, 2,-1)
    decTau   = ( 1,-1,-1,-1, 2,-1,-1, 4,-1,-1, 3,-1,-1, 0,-1,-1)
    Rho      = ( 0,-1, 0,-1, 0,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1)
    constA   = (99,99,99,99, 0,82, 0,82, 0, 0, 0, 0, 0, 0, 0, 0)
n=6: Rounds when t=0: 14, when t>0: 16
    encAlpha = ( 0, 1, 2, 3, 4, 5,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1)
    encBeta  = ( 2, 3, 6, 7, 2, 3,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1)
    encSigma = ( 2,-1,-1,-1, 4, 1,-1,-1,-1, 3, 0,-1,-1,-1,-1, 5)
    decAlpha = ( 0, 1,-1,-1, 2, 3, 4, 5,-1,-1,-1,-1,-1,-1,-1,-1)
    decBeta  = ( 4, 5, 4, 5, 6, 7,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1)
    decSigma = ( 2, 5,-1,-1, 0,-1,-1,-1,-1,-1,-1, 3,-1, 1, 4,-1)
    decTau   = ( 2, 3,-1,-1, 4,-1,-1, 5,-1,-1, 0,-1,-1, 1,-1,-1)
    Rho      = ( 0, 0,-1,-1, 0, 0,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1)
    constA   = ( 0, 0,82,82,99,-91, 0,-58, 0, 0, 0, 0, 0, 0, 0, 0)
n=7: Rounds when t=0: 14, when t>0: 16
    encAlpha = ( 0, 1, 2, 3, 4, 5, 6,-1,-1,-1,-1,-1,-1,-1,-1,-1)
    encBeta  = ( 1, 7, 1, 1, 1, 1, 1,-1,-1,-1,-1,-1,-1,-1,-1,-1)
    encSigma = ( 0,-1,-1,-1, 2, 4,-1,-1,-1, 5, 3,-1,-1,-1, 1, 6)
    decAlpha = ( 0,-1, 2, 3, 4, 5, 6, 1,-1,-1,-1,-1,-1,-1,-1,-1)
    decBeta  = ( 7, 7, 7, 7, 7, 7, 7,-1,-1,-1,-1,-1,-1,-1,-1,-1)
    decSigma = ( 0, 5,-1,-1, 1,-1,-1, 2,-1,-1, 4, 6,-1,-1, 3,-1)
    decTau   = ( 0, 5,-1,-1, 1,-1,-1, 2,-1,-1, 4,-1,-1, 6, 3,-1)
    Rho      = ( 0,-1, 0,-1, 0,-1, 0,-1,-1,-1,-1,-1,-1,-1,-1,-1)
    constA   = ( 0,82, 0, 0,99,99,-91,-58, 0, 0, 0, 0, 0, 0, 0, 0)
n=8: Rounds when t=0: 12, when t>0: 16
    encAlpha = ( 0, 1, 2, 3, 4, 5, 6, 7,-1,-1,-1,-1,-1,-1,-1,-1)
```

```
    encBeta  = (-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1)
    encSigma = ( 0,-1,-1, 7, 2, 1,-1,-1,-1, 3, 4,-1,-1,-1, 6, 5)
    decAlpha = ( 0, 1, 2, 3, 4, 5, 6, 7,-1,-1,-1,-1,-1,-1,-1,-1)
    decBeta  = (-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1)
    decSigma = ( 0, 3,-1,-1, 2,-1,-1, 5,-1,-1, 4, 7,-1, 1, 6,-1)
    decTau   = ( 0, 3,-1,-1, 2,-1,-1, 5,-1,-1, 4, 7,-1, 1, 6,-1)
    Rho      = ( 0, 0, 0, 0,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1)
    constA   = ( 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
n=9: Rounds when t=0: 12, when t>0: 16
    encAlpha = ( 0, 1, 2, 3, 4, 5, 6, 7, 8, 1,-1, 1,-1,-1,-1,-1)
    encBeta  = ( 8, 8, 8, 8,-1,-1,-1,-1,11,-1,-1,-1,-1,-1,-1,-1)
    encSigma = ( 2,-1,-1, 8, 3, 4,-1, 4, 0, 6, 1,-1,-1, 4, 7, 5)
    decAlpha = ( 0, 1, 2, 3, 4, 5, 6, 7, 8,-1, 8,-1,-1,-1,-1,-1)
    decBeta  = ( 9, 9,11, 9,-1,-1,-1,-1, 1,-1,-1,-1,-1,-1,-1,-1)
    decSigma = ( 8, 3, 7,-1, 1,-1,-1, 4, 7,-1, 0, 6,-1, 2, 5,-1)
    decTau   = ( 8, 3,-1,-1, 1,-1,-1, 4, 7,-1, 0, 6,-1, 2, 5,-1)
    Rho      = ( 0,-1, 0,-1,-1,-1,-1,-1, 0,-1,-1,-1,-1,-1,-1,-1)
    constA   = (99,99,99,99, 0, 0, 0, 0, 0,82, 0,82, 0, 0, 0, 0)
n=10: Rounds when t=0: 12, when t>0: 14
    encAlpha = ( 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,-1,-1,-1,-1,-1,-1)
    encBeta  = ( 2, 3,10,11,-1,-1,-1,-1, 2, 3,-1,-1,-1,-1,-1,-1)
    encSigma = ( 7,-1,-1, 9, 0, 4,-1,-1, 6, 2, 1,-1,-1, 8, 5, 3)
    decAlpha = ( 0, 1,-1,-1, 4, 5, 6, 7, 2, 3, 8, 9,-1,-1,-1,-1)
    decBeta  = ( 8, 9, 8, 9,-1,-1,-1,-1,10,11,-1,-1,-1,-1,-1,-1)
    decSigma = ( 4, 6, 9,-1, 1, 3,-1,-1, 5,-1,-1, 0,-1, 2, 8, 7)
    decTau   = ( 4, 6,-1,-1, 1, 7,-1, 3, 9,-1, 5, 0,-1, 2, 8,-1)
    Rho      = ( 0, 0,-1,-1,-1,-1,-1,-1, 0, 0,-1,-1,-1,-1,-1,-1)
    constA   = ( 0, 0,82,82, 0, 0, 0, 0,99,-91, 0,-58, 0, 0, 0, 0)
n=11: Rounds when t=0: 12, when t>0: 14
    encAlpha = ( 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,10,-1,-1,-1,-1,-1)
    encBeta  = ( 1,11, 1, 1,-1,-1,-1,-1, 1, 1, 1,-1,-1,-1,-1,-1)
    encSigma = ( 5,-1,10, 4, 1, 7,-1,-1, 6, 0, 9,-1,-1, 3, 2, 8)
    decAlpha = ( 0,-1, 2, 3, 4, 5, 6, 7, 8, 9,10, 1,-1,-1,-1,-1)
    decBeta  = (11,11,11,11,-1,-1,-1,-1,11,11,11,-1,-1,-1,-1,-1)
    decSigma = ( 5, 0,10,-1, 7, 2,-1, 9, 3,-1, 6, 1,-1,-1, 8, 4)
    decTau   = ( 5, 0,10,-1, 7, 2,-1, 9, 3,-1, 6, 1,-1, 4, 8,-1)
    Rho      = ( 0,-1, 0,-1,-1,-1,-1,-1, 0,-1, 0,-1,-1,-1,-1,-1)
    constA   = ( 0,82, 0, 0, 0, 0, 0, 0,99,99,-91,-58, 0, 0, 0, 0)
n=12: Rounds when t=0: 12, when t>0: 14
    encAlpha = ( 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,10,11,-1,-1,-1,-1)
    encBeta  = (-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1)
    encSigma = ( 0,-1, 3,11, 4, 5,-1, 7, 8, 9, 6,-1,-1, 1, 2,10)
    decAlpha = ( 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,10,11,-1,-1,-1,-1)
    decBeta  = (-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1)
    decSigma = ( 0, 1, 3,-1, 4, 5,-1,10, 8,-1, 6,11,-1, 9, 2, 7)
    decTau   = ( 0, 1, 3,-1, 4, 5,-1,10, 8,-1, 6,11,-1, 9, 2, 7)
    Rho      = ( 0, 0, 0, 0,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1)
    constA   = ( 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
n=13: Rounds when t=0: 12, when t>0: 14
    encAlpha = ( 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,10,11,12, 1,-1, 1)
    encBeta  = (12,12,12,12, 8, 9,10,11,-1,-1,-1,-1,15,-1,-1,-1)
    encSigma = ( 5, 7, 6,10, 2, 7,-1,12,11, 9, 8, 7, 3, 0, 4, 1)
    decAlpha = ( 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,10,11,12,-1,12,-1)
    decBeta  = (13,13,15,13, 8, 9,10,11,-1,-1,-1,-1, 1,-1,-1,-1)
    decSigma = ( 9, 0, 7,-1, 6, 5,11,12, 2,-1, 4, 1,11, 3,10, 8)
    decTau   = ( 9, 0, 7,-1, 6, 5,-1,12, 2,-1, 4, 1,11, 3,10, 8)
    Rho      = ( 0,-1, 0,-1,-1,-1,-1,-1,-1,-1,-1,-1, 0,-1,-1,-1)
    constA   = (99,99,99,99, 0, 0, 0, 0, 0, 0, 0, 0, 0,82, 0,82)
n=14: Rounds when t=0: 12, when t>0: 14
    encAlpha = ( 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,10,11,12,13,-1,-1)
    encBeta  = ( 2, 3,14,15,-1,-1,-1,-1,-1,-1,-1,-1, 2, 3,-1,-1)
    encSigma = ( 4,12,10, 2, 5,11,-1,13, 6, 1, 8,-1, 7, 0, 9, 3)
    decAlpha = ( 0, 1,-1,-1, 4, 5, 6, 7, 8, 9,10,11, 2, 3,12,13)
    decBeta  = (12,13,12,13,-1,-1,-1,-1,-1,-1,-1,-1,14,15,-1,-1)
    decSigma = ( 9, 4,10,11, 0, 6,13,-1, 2, 3,-1,12, 7, 5, 8, 1)
    decTau   = ( 9, 4,10,-1, 0, 6,-1, 3, 2,11, 7,12,13, 5, 8, 1)
    Rho      = ( 0, 0,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, 0, 0,-1,-1)
    constA   = ( 0, 0,82,82, 0, 0, 0, 0, 0, 0, 0, 0,99,-91, 0,-58)
n=15: Rounds when t=0: 12, when t>0: 12
    encAlpha = ( 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,10,11,12,13,14,-1)
    encBeta  = ( 1,15, 1, 1,-1,-1,-1,-1,-1,-1,-1,-1, 1, 1, 1,-1)
    encSigma = ( 4, 0, 1, 2, 5, 8,11, 3, 6, 9,12,-1,14,10,13, 7)
    decAlpha = ( 0,-1, 2, 3, 4, 5, 6, 7, 8, 9,10,11,12,13,14, 1)
    decBeta  = (15,15,15,15,-1,-1,-1,-1,-1,-1,-1,-1,15,15,15,-1)
    decSigma = (13, 4, 9,10, 0, 5,12,11, 1, 6, 7, 3, 2,-1, 8,14)
    decTau   = (13, 4, 9,-1, 0, 5,12,11, 1, 6, 7, 3, 2,10, 8,14)
    Rho      = ( 0,-1, 0,-1,-1,-1,-1,-1,-1,-1,-1,-1, 0,-1, 0,-1)
    constA   = ( 0,82, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,99,99,-91,-58)
```

Below is an exampled (non-optimised) implementation of the 93-bit wide block cipher ALF-11-5 in C/C++, excluding the function to generate round keys.

```
#include <intrin.h> // for gcc use <x86intrin.h> instead
#include <stdint.h>
#include <memory.h>
#include <stdio.h>

const int n = 11, t = 5, Rounds = 14;   // Example for ALF-11-5
__m128i RK[Rounds]; // Round keys
// __m128i encAlpha = _mm_setr_epi8( 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,10,-1,-1,-1,-1,-1);
__m128i encBeta  = _mm_setr_epi8( 1,11, 1, 1,-1,-1,-1,-1, 1, 1, 1,-1,-1,-1,-1,-1);
__m128i encSigma = _mm_setr_epi8( 5,-1,10, 4, 1, 7,-1,-1, 6, 0, 9,-1,-1, 3, 2, 8);
__m128i decAlpha = _mm_setr_epi8( 0,-1, 2, 3, 4, 5, 6, 7, 8, 9,10, 1,-1,-1,-1,-1);
__m128i decBeta  = _mm_setr_epi8(11,11,11,11,-1,-1,-1,-1,11,11,11,-1,-1,-1,-1,-1);
__m128i decSigma = _mm_setr_epi8( 5, 0,10,-1, 7, 2,-1, 9, 3,-1, 6, 1,-1,-1, 8, 4);
__m128i decTau   = _mm_setr_epi8( 5, 0,10,-1, 7, 2,-1, 9, 3,-1, 6, 1,-1, 4, 8,-1);
__m128i Rho      = _mm_setr_epi8( 0,-1, 0,-1,-1,-1,-1,-1, 0,-1, 0,-1,-1,-1,-1,-1);
__m128i A        = _mm_setr_epi8( 0,82, 0, 0, 0, 0, 0, 0,99,99,-91,-58, 0, 0, 0, 0);
__m128i B = ((n % 4) == 3) ? _mm_set1_epi8(0x52) : _mm_setzero_si128();
__m128i M = _mm_setr_epi32((1 << t) - 1, 0, 0, 0);

#define xor2(a, b) _mm_xor_si128(a, b)
#define xor3(a, b, c) xor2(xor2(a,b),c)
#define c_00 _mm_setzero_si128()
#define load128(ptr) _mm_loadu_si128((__m128i*)(ptr))
#define shuffle(x, sh) _mm_shuffle_epi8(x, sh)

inline __m128i _UpdateE0(__m128i E, __m128i U, __m128i M)
{   U = _mm_clmulepi64_si128(U, _mm_set1_epi64x(0x01010101), 0x00);
    U = _mm_srli_epi32(U, 24); // can be opt-out if E_3 would hold the t bits, instead
    return _mm_and_si128(xor2(U, E), M);
}

void Encrypt(uint8_t* out /* [16] */, uint8_t* in /* [16] */)
{   __m128i U, X = load128(in), E = load128(in + n);
    for (int r = 0; r < Rounds; r++)
    {   U = _mm_aesenc_si128(shuffle(X, encSigma), RK[r]);
        X = xor3(U, shuffle(U, encBeta), shuffle(E, Rho));
        E = _UpdateE0(E, U, M);   // when t>0
    }
    memcpy(out, &X, n);
    out[n] = _mm_extract_epi8(E, 0);
}

void Decrypt(uint8_t* out /* [16] */, uint8_t* in /* [16] */)
{   _m128i X = load128(in), E = load128(in + n);
    X = _mm_aesenclast_si128(shuffle(X, encSigma), c_00); // Auxiliary step
    for (int r = Rounds - 1; r >= 0; r--)
    {   X = _mm_aesdec_si128(shuffle(X, decSigma), RK[r]);
        E = _UpdateE0(xor2(E, B), xor2(X, RK[r]), M);    // when t>0
        X = xor3(X, shuffle(X, decBeta), shuffle(E, Rho));
    }
    X = _mm_aesdeclast_si128(shuffle(X, decTau), c_00); // SRF
    memcpy(out, &X, n);
    out[n] = _mm_extract_epi8(E, 0);
}

void PrepareDecryption(void)
{   for (int r = 0; r < Rounds; r++)
    {   __m128i U = xor2(RK[r], shuffle(RK[r], encBeta));
        RK[r] = _mm_aesimc_si128(xor2(A, shuffle(U, decAlpha)));
    }
}

int main()
{   uint8_t out[16], in[16] = {1,2,3,4,5,6,7,8,9,10,11,0x1a};  // 11 full bytes plus 5 bits
    for (int r = 0; r < Rounds; r++)   // generate pseudo-random round keys
    {   RK[r] = c_00;
        for (int i = 0; i < n; i++) ((uint8_t*)(RK + r))[i] = 16 * r + i + 1;
    }
    Encrypt(out, in);
    // out = Enc(in) = {0xc4,0x12,0x73,0x4f,0x64,0x7a,0x7d,0xaa,0x3f,0x73,0x60,0x13}
    PrepareDecryption();
    Decrypt(out, out);
    printf("Compare In vs. Dec(Enc(In)): %s\n", memcmp(in, out, n + 1) ? "FAIL" : "PASS");
}
```

D More details on security analysis results

Since the first round of ALF-n-t can be seen as a simple XOR with the first round key, that round can be "virtually" removed. Therefore, the tables start from round 2.

Table 9: Full table of linear and differential analysis results for $n \in [2, 8]$.

r	$n=2$	$n=3$	$n=4$	$n=5$	$n=6$	$n=7$	$n=8$
The number of active Sboxes "$a, b/c, d$" for each r and n, according to four scenario: a–differential with $t=0$, b–linear with $t=0$, c–differential with $t>0$, d–linear with $t>0$.							
2	1,1/0,0	1,1/0,0	1,1/0,0	1,1/0,0	1,1/0,0	1,1/0,0	1,1/0,0
3	3,3/2,2	3,3/2,2	5,5/3,3	5,3/3,3	5,4/4,2	5,4/4,3	5,5/3,3
4	4,4/2,2	5,5/3,3	6,6/4,4	7,6/4,5	8,8/5,6	8,7/7,5	8,8/6,6
5	6,6/4,4	7,7/4,5	10,10/7,7	10,9/7,7	10,10/8,7	10,10/10,9	15,15/9,9
6	7,7/4,4	9,9/5,6	11,11/8,8	12,11/8,9	13,13/10,9	14,13/11,11	16,16/13,13
7	9,9/6,6	11,11/7,8	15,15/11,11	15,14/11,11	16,16/13,12	17,17/15,13	20,20/16,16
8	10,10/6,6	13,13/8,9	16,16/12,12	18,17/12,14	18,19/15,15	19,20/18,16	23,23/18,18
9	12,12/8,8	15,15/9,10	20,20/15,15	20,19/15,16	22,22/18,17	23,23/20,19	30,30/21,21
10	13,13/8,8	17,17/10,12	21,21/16,16	23,22/16,18	25,25/20,19	26,26/22,22	31,31/24,24
11	15,15/10,10	19,19/12,13	25,25/19,19	26,25/19,20	27,28/23,22	28,29/26,24	35,35/29,29
12	16,16/10,10	21,21/13,14	26,26/20,20	28,27/20,22	30,31/25,24	32,32/29,27	38,38/31,31
13	18,18/12,12	23,23/14,16	30,30/23,23	31,30/23,24	34,34/28,27	35,35/31,30	45,45/33,33
14	19,19/12,12	25,25/15,17	31,31/24,24	34,33/24,26	36,37/30,29	37,39/33,32	46,46/36,36
15	21,21/14,14	27,27/17,18	35,35/27,27	36,35/27,28	39,40/33,32	41,42/37,35	50,50/39,39
16	22,22/14,14	29,29/18,20	36,36/28,28	39,38/28,30	42,43/35,34	44,45/40,38	53,53/44,44
17	24,24/16,16	31,31/19,21	40,40/31,31	42,41/31,32	45,46/38,37	46,48/42,41	60,60/46,46
18	25,25/16,16	33,33/20,22	41,41/32,32	44,43/32,34	48,49/40,39	50,51/44,43	–
19	27,27/18,18	35,35/22,24	45,45/35,35	47,46/35,36	51,52/43,42	53,54/48,46	–
20	28,28/18,18	37,37/23,25	46,46/36,36	50,49/36,38	54,55/45,44	55,57/51,49	–
21	30,30/20,20	39,39/24,26	50,50/39,39	52,51/39,40	57,58/48,47	59,61/53,51	–
22	31,31/20,20	41,41/25,28	51,51/40,40	55,54/40,42	–	–	–
23	33,33/22,22	43,43/27,29	55,55/43,43	58,57/43,44	–	–	–
24	34,34/22,22	45,45/28,30	–	–	–	–	–
25	36,36/24,24	47,47/29,32	–	–	–	–	–
26	37,37/24,24	49,49/30,33	–	–	–	–	–

Table 10: Full table of linear and differential analysis results for $n \in [9, 15]$.

r	$n=9$	$n=10$	$n=11$	$n=12$	$n=13$	$n=14$	$n=15$
The number of active Sboxes "$a, b/c, d$" for each r and n, according to four scenario: a–differential with $t=0$, b–linear with $t=0$, c–differential with $t>0$, d–linear with $t>0$.							
2	1,1/0,0	1,1/0,0	1,1/0,0	1,1/0,0	1,1/0,0	1,1/0,0	1,1/0,0
3	5,3/3,3	5,4/4,2	5,4/4,3	5,5/3,3	5,3/3,3	5,4/4,2	5,4/4,3
4	8,7/5,6	8,8/7,6	8,8/8,7	8,8/6,6	8,7/6,7	9,9/8,6	8,8/8,7
5	11,11/8,8	12,12/11,11	12,13/12,12	15,15/12,12	15,15/14,14	18,18/15,16	18,18/16,16
6	17,17/11,12	17,17/15,15	18,18/16,16	21,21/17,17	21,21/18,18	23,23/18,18	23,24/20,20
7	20,19/14,17	21,21/19,17	22,21/20,19	25,25/21,21	25,23/21,21	27,26/21,21	27,27/23,23
8	23,22/17,19	24,25/22,21	26,25/24,22	28,28/24,23	28,27/24,24	31,30/26,26	31,30/27,27
9	27,25/20,22	28,29/26,24	29,29/27,26	31,31/27,26	33,33/29,29	36,36/31,32	38,37/33,34
10	30,30/23,25	32,33/30,28	33,34/31,30	38,38/32,32	39,39/35,35	43,42/35,35	44,44/38,38
11	35,35/26,27	37,37/33,31	38,39/34,33	45,45/37,37	43,43/39,38	48,48/40,38	48,49/42,42
12	39,37/29,31	41,41/37,35	43,42/38,37	48,48/41,41	48,47/42,41	52,52/44,43	–
13	42,41/32,35	44,45/41,38	47,46/41,40	51,51/45,43	52,52/46,46	–	–
14	45,44/35,38	48,49/45,42	50,50/45,44	54,54/48,46	–	–	–
15	49,49/38,41	52,53/48,45	54,55/48,47	61,61/52,52	–	–	–
16	54,53/41,43	57,57/52,49	59,59/52,51	–	–	–	–
17	57,56/44,46	61,61/56,52	63,63/55,54	–	–	–	–

Table 11: Full table of integral attacks.

	t=0	t=1	t=2	t=3	t=4	t=5	t=6	t=7
n=2	3	4	4	4	4	4	4	4
n=3	3	4	4	4	4	4	4	4
n=4	3	4	4	4	4	4	4	5
n=5	3	4	4	4	4	5	5	5
n=6	4	4	4	5	5	5	5	5
n=7	4	5	5	5	5	5	5	5
n=8	5	6	6	6	6	6	6	6
n=9	5	6	6	6	6	6	6	6
n=10	5	6	6	6	6	6	6	6
n=11	5	6	6	6	6	6	6	6
n=12	5	6	6	6	6	6	6	6
n=13	5	6	6	6	6	6	6	6
n=14	5	6	6	6	6	6	6	6
n=15	5	6	6	6	6	6	6	6

E Example trails

We provide a few differential trails as examples in Table 12 to help the reader to verify the details. As an attack outline using a differential trail, an attacker can prepare pairs of plaintexts with a certain input difference, then check output differences of the ciphertexts. Once the attacker observes a certain output difference to appear more frequent than expected in a random case, the distinguishing attack succeeds. Table 12 contains examples for three reduces-round ALF instances, the exampled trails and their probabilities, as well as the outlined attack complexities.

The differences are given in hexadecimal form. Note that all trails in Table 12 start from round 2, since the first round of ALF-n-t has no effect in a differential attack and can be viewed as a whitening key, i.e., the attacker can construct any plaintext pairs matching the input difference at the second round by inverting the operations of the first round.

Table 12: Example differential trails.

rounds	ALF-2-7	ALF-8-0	ALF-8-7
2	`00,00,2`	`00,00,ff,00,00,00,ff,ff`	`00,00,00,00,00,ff,00,00,1`
3	`02,02,2`	`00,00,00,00,08,0c,00,00`	`00,00,02,03,00,00,00,00,0`
4	`00,00,2`	`18,38,38,00,00,00,00,00`	`00,00,00,00,40,c0,00,80,0`
5	`02,02,2`	`02,14,06,16,20,10,10,30`	`07,05,08,0d,40,40,c0,80,7`
active Sboxes	2	8	6
probability	2^{-12}	2^{-48}	2^{-36}
complexity	$\sim 2^{12}$	$\sim 2^{48}$	$\sim 2^{36}$

F Complementary ALF instances

In previous sections we covered the most challenging case of ALF-n-t, supporting a modulus $Q \in (2^{15}, 2^{127}]$. In this section we *briefly* introduce a number of complementary ALF variants for completeness:

- ALF-0 provides encryption of a single plaintext with a very small modulus $Q \in [2, 256]$, or a length-preserving encryption with a width of 1 to 8 bits.
- ALF-1-t, $t \in [1, 7]$, provides encryption of a single plaintext modulo $Q \in (2^8, 2^{15}]$, or a length-preserving encryption with width of 9 to 15 bits determined by $(8 \cdot 1 + t)$.
- ALF-16-t, $t \in [0, 16]$, provides encryption of a single plaintext modulo $Q \in (2^{127}, 2^{144}]$, or a length-preserving encryption with width of 128 to 144 bits determined by $(8 \cdot 16 + t)$.
- ALF-L provides encryption of a long plaintext that consists of N 16-bit plaintext symbols $(p_0, \ldots, p_{N-1})$, each having its own modulus q_i in the range $q_i \in [1, 2^{16}]$, or a length-preserving encryption of arbitrary binary width.

F.1 ALF-0

This is the smallest ALF variant and the idea behind is to create a secret Sbox of size $Q \in [2, 256]$, so that an encryption is a table look-up. There exist many ways to generate a secret Sbox of such a small domain, but ALF-0 leverages AES-NI. Let a secret Sbox on Q values be $S_Q(x)$, and the Rijndael Sbox is $S(x)$. We first create a secret Sbox S_{256} on 256 values as follows:

$$S_{256}(x) = S(\ldots S(S(S(x) \oplus RK[0]) \oplus RK[1]) \oplus RK[2]) \ldots) \oplus RK[r-1],$$

where RKs are 1-byte round keys. For a 128-bit security, the minimum number of round keys is $r = 32$, based also on the attack outlined in Section 3.4. The way to generate these round keys will be given in more detail in Appendix F.5.

In practice, we first assign a 128-bit register $X = (0, 1, \ldots, 15) \in \mathbb{B}_{16}$, then apply the AES last round r times: $X = \texttt{aesenclast}(X, K_i)$ where $K_i = \{RK[i]\}^{16}$. The resulting vector of X will implement exactly the formulae for $S_{256}(x)$ for the first 16 values $x \in [0, 15]$, in parallel. Note that the last round of AES excludes MC, and SR has the period of 4 clocks. We repeat the above procedure 16 times to complete $S_{256}(x)$ for all x, and since these steps are independent then one can perform multiple parallel computations within the same loop utilising CPU interleaving effects. A faster computation of $S_{256}(x)$ can be achieved by using AVX-512 and four 512-bit registers.

The last step is to shrink S_{256} down to S_Q. This step can be performed by collecting all values of S_{256} that are less than Q, and moving them sequentially into a new table S_Q while preserving the order. This step can be done linearly, while using `_pext_u64()` can partially speed up this step up to 8 times.

F.2 ALF-1-t

ALF-1-t implements a small domain block cipher with a width between 9 and 15 bits. The state consists of a full byte $X \in \mathbb{N}_8$ and $t \in [1, 7]$ unaligned bits $E \in \mathbb{N}_t$. The rounds are:

$$\begin{aligned}
\textbf{Enc.}: \quad & E' = (S(X) \oplus RK[i] \oplus E)|_t \\
& X' = S(X) \oplus RK[i] \oplus (E \ll 1) \\
\textbf{Dec.}: \quad & E = \texttt{clmulqdq}(X' \oplus E', \texttt{0xff})|_t \\
& X = S^{-1}(X' \oplus (E \ll 1) \oplus RK[i])
\end{aligned}$$

where $RK[i]$ are round keys for $i \in [0, r)$, $S()$ is the Rijndael Sbox, and $\texttt{clmulqdq}(x, \texttt{0xff})$ is the carryless multiplication of two bytes x and $\texttt{0xff}$, which can be implemented by e.g. a constant table with 256 entries. The number of rounds is $r = 48$ and the exact derivation of round keys will be given in Appendix F.5. Note, it is possible to implement ALF-1-t with SIMD registers and utilise AES-NI in case an application needs to perform up to 64 parallel encryptions sharing the same registers (AVX-512).

Algorithm 3 ALF-1-t Encryption.

function $\texttt{ALF1t-Enc}(Q \in (2^8, 2^{15}], P \in [0, Q), RK[0..47] \in \mathbb{B}_1^{48}) \to C \in [0, Q)$
  Let $t = \lceil \log_2 Q \rceil - 8$
  $X = P|_8 \in \mathbb{B}_1, E = P \gg 8 \in \mathbb{B}_1$
  for $i = 0 \dots 23$ **do**
    while True **do**
      for $j = 0 \dots 1$ **do**
        $U = S(X) \oplus RK[2i + j]$
        $X = U \oplus (E \ll 1)$
        $E = (U \oplus E)|_t$
      if $(E \cdot 2^8 + X) < Q$ then **break**
  return $C = E \cdot 2^8 + X$

When the modulus $Q \in (2^8, 2^{15}]$ is not a power of 2, we then determine t as the minimum number of bits such that the integer $(Q - 1)$ fits in $(8 + t)$ bits, then we repeat each block of two rounds until $(E \cdot 2^8 + X) < Q$ – same approach with cycle-sliding as for ALF-n-t. For all the details, we provide the encryption flow in Algorithm 3, and the decryption flow is trivial while the loop goes in the reversed order. The analysis technique in Section 3 also works for ALF-1-t. In both differential and linear cryptanalysis, 48 rounds of ALF-1-t gives 24 actives Sboxes for $t \neq 0$. In integral attacks, we could only find a 1-round distinguisher.

F.3 ALF-16-t

This variant of ALF has the width from 128 to 128+16 bits, $t \in [0, 16]$, and designed for the purpose to further support ALF-L while providing 128 bits of security.

Algorithm 4 ALF-16-t Encryption.

function $\texttt{ALF16t-Enc}(Q \in (2^{127}, 2^{144}], P \in [0, Q), RK[0..11] \in \mathbb{B}_{16}^{12}) \to C \in [0, Q)$
  Let $t = \lceil \log_2 Q \rceil - 128$
  Split the plaintext $X = P|_{128} \in \mathbb{B}_{16}, E = (P \gg 128) \in \mathbb{B}_2$
  for $i = 0 \dots 5$ **do**
    while True **do** ▷ only for non-binary
      for $j = 0 \dots 1$ **do**
        $U = \texttt{aesenc}(X, RK[2i + j])$
        $p_0 = U_0 \oplus U_1 \oplus U_2 \oplus U_3$ ▷ parity of the first column
        $p_1 = U_4 \oplus U_5 \oplus U_6 \oplus U_7$ ▷ parity of the second column
        $X = U \oplus (E_0, E_0, E_0, E_0, E_1, E_1, E_1, E_1, 0, 0 \dots)$
        $E = (E \oplus (p_1 \cdot 2^8 + p_0))|_t$ ▷ only when $t \neq 0$
      if $(E \cdot 2^{128} + X) < Q$ then break the loop ▷ only for non-binary
  return $C = E \cdot 2^{128} + X$

The largest 128-bit part of the input is encrypted with $r = 12$ standard AES rounds, while the addition to the round is the way how the t-bit exchange of entropy is performed. Here, the unaligned t bits are stored in a register E where the first byte is exchanged with the parity of the first column of X, and the second byte of E is exchanged with the parity of the second column. For FPE use cases, the supported modulus must be in the range $Q \in (2^{127}, 2^{144}]$ and the cycle-sliding technique applies similarly to ALF-n-t. The way how encryption round keys are derived is given in Appendix F.5, and for the decryption direction they are transformed in the same way as in the standard AES: $RK'[i] = MC'(RK[i])$, for

Algorithm 5 ALF-16-t Decryption.

function ALF16t-Dec($Q \in (2^{127}, 2^{144}], C \in [0, Q), RK'[0..11] \in \mathbb{B}_{16}^{12}) \to P \in [0, Q)$
  Let $t = \lceil \log_2 Q \rceil - 128$
  Split the ciphertext $X = C|_{128} \in \mathbb{B}_{16}, E = (C \gg 128) \in \mathbb{B}_2$
  $X = \texttt{aesenclast}(X, \{0\}^{16})$ ▷ auxiliary step
  for $i = 5 \ldots 0$ **do**
    while True **do** ▷ only for non-binary
      for $j = 1 \ldots 0$ **do**
        $X = \texttt{aesdec}(X, \{0\}^{16})$
        $p_0 = X_0 \oplus X_1 \oplus X_2 \oplus X_3$ ▷ parity of the first column
        $p_1 = X_4 \oplus X_5 \oplus X_6 \oplus X_7$ ▷ parity of the second column
        $E = (E \oplus (p_1 \cdot 2^8 + p_0))|_t$ ▷ only when $t \neq 0$
        $X = X \oplus RK'[2i + j] \oplus (E_0, E_0, E_0, E_0, E_1, E_1, E_1, E_1, 0, 0 \ldots)$
      $U = \texttt{aesdeclast}(X, \{0\}^{16})$ ▷ SRF function
      if $(E \cdot 2^{128} + U) < Q$ then break the loop ▷ only for non-binary
  return $P = E \cdot 2^{128} + U$

$i = 0..11$. In a sense, ALF-16-t is very similar to ALF-n-t but without $\sigma, \pi, \alpha, \beta$, etc; for a more detailed description we refer to Algorithm 4 and Algorithm 5.

For the full diffusion round count, it is 2 for $t = 0$ and 3 for $t \in [1, 16]$. The number of active Sboxes in a differential analysis is 75 for $t = 0$ and 51 for $t \neq 0$. Similarly, our tool shows more than 72 and 51 active Sboxes for $t = 0$ and $t \neq 0$ in a linear analysis, respectively. Finally, the best found integral attacks cover 5 and 6 rounds for $t = 0$ and $t \neq 0$.

F.4 ALF-L

The variant ALF-L encrypts a long plaintext of length $N < 2^{48}$, where each symbol is a 16-bit value, i.e. $P = (p_0, p_1, \ldots, p_{N-1})$ where $p_i \in \mathbb{N}_{16}$, and each element may have it's own modulus, i.e., $Q = (q_0, q_1, \ldots, q_{N-1})$ where $q_i \leq 2^{16}$. Note that ALF-L also covers the case of "same moduli" where $q_i = q_j$ for all $0 \leq i, j < N$. The application domain of ALF-L is when $\prod_{i=0}^{N-1} q_i > 2^{144}$, otherwise a smaller variant of ALF is applied.

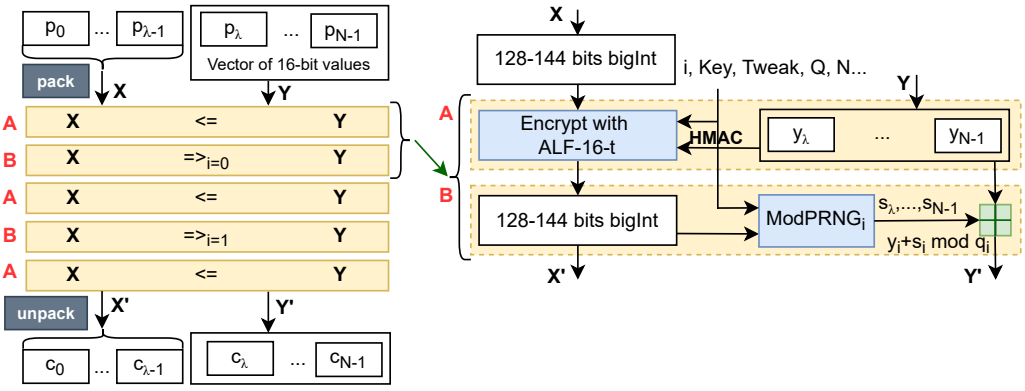

Figure 8: Encryption method for ALF-L.

The general scheme for ALF-L is depicted on Figure 8. In the first step the plaintext vector is split in two parts $P = (X \in \mathbb{N}_{128..144}, Y \in \mathbb{N}_{16}^{N-\lambda})$, where X combines the first λ

16-bit plaintext symbols into a big integer value, such that the combined modulus

$$Q_\lambda = q_0 \cdot q_1 \cdot \ldots \cdot q_{\lambda-1} \to \max, \quad Q_\lambda \le 2^{144}, 1 \le \lambda \le N,$$

is maximised, but at most 2^{144}. The packing procedure is as follows:

$$X = (\ldots(((p_0 \cdot q_1) + p_1) \cdot q_2 + p_2)\ldots) \cdot q_{\lambda-1} + p_{\lambda-1}.$$

What remains from the plaintext is the vector $Y = \{p_\lambda, \ldots, p_{N-1}\} \in \mathbb{N}_{16}^{N-\lambda}$, and it is kept as-is without a need for additional processing.[10] Five steps of encryption are then performed repeating two steps A and B in the order A-B-A-B-A, after which the X-part is unpacked back to the ciphertext symbols. The A/B steps are:

A. The vector Y is HMACed and mixed with the key, tweak, and other parameters. The output is used to generate round keys for ALF-16-t which encrypts X modulo Q_λ.

B. The vector Y is additively encrypted using a keystream $s_\lambda, \ldots, s_{N-1}$ that is in turn produced by a modular pseudo-random number generator (`ModPRNG`) initialised by the secret key, tweak, configuration parameters such as moduli and the length N, **and** the value of X itself.

In many FPE schemes, the whole plaintext needs to be packed first to a very large integer before encryption – this step is usually expensive in CPUs. In case of ALF-L, the longest part of the plaintext Y remains intact, and only the smaller part X needs long arithmetic over integers up to 144 bits. This is a big advantage that helps to achieve a higher performance. Moreover, the packing of X can be implemented by first packing into chunks in 64-bit scalar registers, while only a few long multiplications remained.

The part of HMACing and generation of the round keys for ALF-16-t will be given in Appendix F.5, and a technical/engineering solution for an unbiased modular keystream `ModPRNG`, highly optimised for the use of SIMD, is given in Appendix F.4.1.

F.4.1 ModPRNG – a fast modular pseudo-random number generator

For the purpose of concrete ALF specification, in this section we define a binary and modular *deterministic* PRNG functions, called `BinPRNG` and `ModPRNG`, respectively. Both algorithms are adapted for an efficient implementation using SIMD with 128/256-bit registers and AES-NI/SSE/AVX2 instructions. These algorithms may however be also implemented on a more restricted platform. Finally, we note that `ModPRNG` can be used as a stand-alone FPE stream cipher when a **nonce** is available.

Algorithm 6 Binary PRNG

1: **function** `BinPRNG`($\mathbf{S}$) $\to$ ($\mathbf{S}', Z \in \mathbb{N}_{256}$)
2: $Z0 =$ `aesenc`($\mathbf{S}.S3 \oplus \mathbf{S}.S5, \mathbf{S}.S0$)
3: $Z1 =$ `aesenc`($\mathbf{S}.S4 \oplus \mathbf{S}.S6, \mathbf{S}.S2$)
4: $\mathbf{S}'.S0 = \mathbf{S}.S6 \oplus \mathbf{S}.S1$
5: $\mathbf{S}'.S1 =$ `aesenc`($\mathbf{S}.S0, \{0\}^{16}$)
6: $\mathbf{S}'.S2 =$ `aesenc`($\mathbf{S}.S1, \mathbf{S}.S0$)
7: $\mathbf{S}'.S3 =$ `aesenc`($\mathbf{S}.S2, \mathbf{S}.S6$)
8: $\mathbf{S}'.S4 =$ `aesenc`($\mathbf{S}.S3, \{0\}^{16}$)
9: $\mathbf{S}'.S5 =$ `aesenc`($\mathbf{S}.S4, \mathbf{S}.S3$)
10: $\mathbf{S}'.S6 =$ `aesenc`($\mathbf{S}.S5, \mathbf{S}.S4$)
11: **return** ($\mathbf{S}', Z = Z0||Z1$)

The first Algorithm 6 specifies `BinPRNG` that is based on the AEAD cipher Rocca-S [NFI24] whose keystream is regarded as a pseudo-random binary stream. The state of Rocca-S is seven 128-bit registers $\mathbf{S} = (S0, S1, S2, S3, S4, S5, S6) \in \mathbb{N}_{128}^7$. For the purpose of ALF, we skip Rocca-S' original initialisation procedure and instead provide ALF's alternative initialisation as given in Appendix F.5. I.e., further in this section we assume that the state is already initialised.

[10] When $\lambda = N$ we get an empty Y-part and thus only encrypt the X-part. Note that in this case Q_λ can be any integer between 1 and 2^{144}, and thus *any* of the smaller ALF variants can be used, not only ALF-16-t as in the case of ALF-L.

Given a pseudo-random initial state $\mathbf{S}$, every call to `BinPRNG` produces 256 bits of fresh pseudo-random bits, as well as the state is updated to $\mathbf{S}'$ for the next calls.

The second Algorithm 7 specifies `ModPRNG` that generates a sequence of N samples[11] $(s_0, \ldots, s_{N-1})$ modulo $(q_0, \ldots, q_{N-1})$, respectively, based on `BinPRNG`. In the proposed algorithm, each modulus is at most 2^{16} and each returned sample s_i occupies 16 bits, i.e., $s_i \in \mathbb{N}_{16}$ as well as $s_i \in \mathbb{Z}_{q_i}$.[12]

Algorithm 7 Modular PRNG

1: **function** `ModPRNG`$(\mathbf{S}, N, (q_0, \ldots, q_{N-1}) : q_i \in [1, 2^{16}]) \to (s_0, \ldots, s_{N-1}) : s_i \in \mathbb{Z}_{q_i}$
2: Let `Pool`$[0..7] \in \mathbb{N}_{32}^8$, `Pool`$_{sz} = 8$, and $L_j, H_j, L'_j, H'_j \in \mathbb{N}_{16}$ for $j = 0..15$
3: Extend moduli $q_N = q_{N+1} = \ldots = q_{N'-1} = 2^{16}$, until N' is divisible by 16.
4: **for** $i = 0..N' - 1$ **by** 16 **do**
5: Get $(\mathbf{S}, L_{0..15}) \leftarrow$ `BinPRNG`$(\mathbf{S})$ and $(\mathbf{S}, H_{0..15}) \leftarrow$ `BinPRNG`$(\mathbf{S})$
6: Compute $\forall j = 0..15 :$ $\underbrace{s_{i+j} \cdot 2^{32} + H'_j \cdot 2^{16} + L'_j}_{\text{split the result to } 3 \times \text{16-bit chunks}} \leftarrow \underbrace{(H_j \cdot 2^{16} + L_j) \cdot q_{i+j}}_{\text{multiply 32- by 16-bit integers}}$
7: **for** $\forall j = 0..15 : (H'_j \cdot 2^{16} + L'_j) < q_{i+j}$ **do** ▷ with probability $\lessapprox 2^{-16}$
8: Set $l = L'_j$ and $t = (2^{32} - q_{i+j}) \mod q_{i+j}$
9: **while** $l < t$ **do**
10: **if** `Pool`$_{sz} = 8$ **then** $(\mathbf{S}, $`Pool`$) \leftarrow$ `BinPRNG`$(\mathbf{S})$ and `Pool`$_{sz} = 0$
11: Compute $m = q_{i+j} \cdot$ `Pool`$[$`Pool`$_{sz}]$ and `Pool`$_{sz} =$ `Pool`$_{sz} + 1$
12: Split $s_{i+j} = m \gg 32$ and $l = m \mod q_{i+j}$
13: **return** $(s_0, \ldots, s_{N-1})$

`ModPRNG` implements the idea of Algorithm 5 from [Lem19], but performs the computation of 16 samples in parallel. As the source of unbiased binary sampling, `ModPRNG` uses `BinPRNG` – each call returns 256-bit binary block that is then viewed as either an 8×32-bit or 16×16-bit vector. For a single 16-bit sample s_i, `ModPRNG` uses 2×16 bits from the binary source, in most cases. This way, to generate 16 16-bit samples `ModPRNG` calls `BinPRNG` twice (and more times in rare cases). The parallel 16-samples computation can be efficiently implemented without leaving SIMD registers, however, in some very rare cases (with probability less than 2^{-16}), `ModPRNG` enters a more expensive procedure (the loop in step 7 of Algorithm 7) where certain samples need to be fixed using scalar registers, in order to ensure all samples are unbiased – there, a pool of 32-bit binary samples is used.

F.4.2 Initial security arguments

Figure 9 outlines the five encryption rounds of ALF-L; the secret key is inserted into the block cipher E, HMAC and PRNG functions, but not shown on the figure. What is important in the scheme is that X has at least 128 bits of entropy and Y is a vector of other data, that can be small or large.

Obviously, two rounds are not enough for a 128-bit security. For example, let the plaintext be $P = (X_1, Y_1)$ and the ciphertext be $C = (X_2, Y_2)$, then any differential inserted in Y_2 would lead to the original Y_1 with exactly the same differential as the one inserted in Y_2. The same issue appears if we assume $(X_2, Y_1) - (X_3, Y_2)$ as the plaintext-ciphertext pair.

[11] Each s_i would be an unbiased sample in case the underlaying binary PRNG is also unbiased, which is not true in reality.

[12] In our actual implementation of Algorithm 7 each q_i fits into a 16-bit type `_uint16_t`, where the modulus 2^{16} is replaced by zero. As for the API, the caller provides the values $q_i - 1$, instead, so that each modulus can fit in a 16-bit data type – these are only technical considerations that do not impact the actual algorithms.

Consider now three rounds such that $P = (X_2, Y_1)$ and $C = (X_3, Y_3)$, as highlighted by the dashed line on Figure 9, i.e., we encrypt Y two times while X is encrypted once. Here it seems that introducing a differential $Y_3' = Y_3 + \Delta Y_3$ would result in a totally different Y_1' when decrypted. If X_3 is kept the same for two decryption queries, PRNG_2 would produce the same keystream and therefore $Y_2' = Y_2 + \Delta Y_3$. A new Y_2' would result in a totally pseudo-random X_2' that would in turn decrypt Y_2' to Y_1' in an unpredictable way. However, the attacker can collect around 2^{64} decryption calls and introduce 2^{64} differentials $\Delta Y_3'$ such that two of them would collide in the value $X_2' = X_2''$ due to the birthday paradox (assuming the width of X is the smallest 128 bits). This collision would lead to the same keystream from PRNG_1 and this would result in $\Delta Y_1' + \Delta Y_1'' = \Delta Y_3' + \Delta Y_3''$ with probability 1 for that pair. This leads to a distinguishing attack of complexity 2^{64} even though the total length of P can be large. A similar attack can be attached by introducing a differential on the plaintext and then observing the ciphertext.

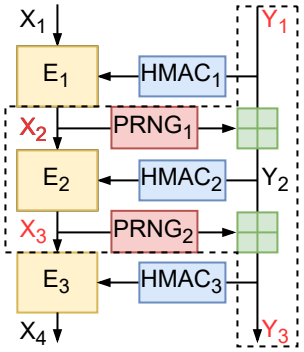

Figure 9: ALF-L outline.

The previous attack on three rounds worked since the attacker keeps the initial (or final) X side as a constant, while introducing a differential into the Y-part. In order to prevent that vector we introduce one round of encryption of X from both sides of the scheme, resulting in 5 rounds in total – this protection is the least needed to resist CPA and CCA. Here, a collision at points X_2 and X_3 could still happen leading to both PRNGs to produce the same keystreams, but such a joint collision can happen with probability 2^{-256}, hence needing 2^{128} queries.

F.4.3 Extended security arguments

In this section we analyse further an adversary advantage for ALF-L from a different angle. Let us for simplicity assume a binary construct where X has the size n bits, Y has the size m bits, and HMAC produces a w-bit MAC value, see Figure 10. Only for the purpose of this section we adopt

$$N = 2^{-n}, M = 2^{-m}, W = 2^{-w}.$$

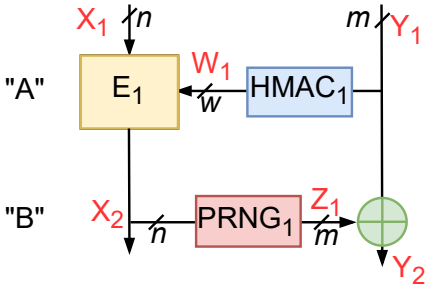

Figure 10: Two rounds ("A" and "B") for a simplified binary model.

Consider a pair of distinct inputs (X_1, Y_1) and (X_1', Y_1'). There are 3 principal cases to study: (a) $X_1 = X_1' \wedge Y_1 \neq Y_1'$; (b) $X_1 \neq X_1' \wedge Y_1 = Y_1'$; and (c) $X_1 \neq X_1' \wedge Y_1 \neq Y_1'$. Each of these cases is propagated to two distinct outputs (X_4, Y_3) and (X_4', Y_3') after 5 rounds of ALF-L (see Figure 9), having similar 3 cases. In an ideal PRP, each of the input case should map to an output case uniformly, such that

$$\begin{aligned} &Pr\{\texttt{case}(X_1, Y_1, X_1', Y_1') \mapsto^{\texttt{ALF-L}} \texttt{case}(X_4, Y_3, X_4', Y_3')\} \\ &= \begin{cases} (1 \pm \epsilon_a)N(1-M), & (a)\ X_4 = X_4' \wedge Y_3 \neq Y_3' \\ (1 \pm \epsilon_b)M(1-N), & (b)\ X_4 \neq X_4' \wedge Y_3 = Y_3' \\ (1 \pm \epsilon_c)(1-N)(1-M), & (c)\ X_4 \neq X_4' \wedge Y_3 \neq Y_3' \end{cases} \end{aligned} \tag{23}$$

where $\epsilon_a, \epsilon_b, \epsilon_c$ are the advantages for each of the input-output cases introduced by ALF-L, and they are zeroes for an ideal PRP. An attack strategy could be to see which input-output

combination gives the largest advantage, then collect q queries and exploit the advantage for e.g. a distinguishing attack.

Let us have a state $\mathbf{S}_{i,j} = (X_i, Y_j, X'_i, Y'_j)$ at every step of the algorithm, with three associated probability formulae $\mathbf{S}^a_{i,j}, \mathbf{S}^b_{i,j}, \mathbf{S}^c_{i,j}$ corresponding to the three cases (a)-(c) described above, with the starting state $\mathbf{S}_{1,1}$ and the final state $\mathbf{S}_{4,3}$. The initial state $\mathbf{S}_{1,1}$ is initialised with probability 1 in the corresponding case (a-c), and 0 in two other cases. We want to derive the probabilities of the final state $\mathbf{S}_{4,3}$ and derive the advantages.

$\mathbf{S}_{i,j} \mapsto \mathbf{S}_{i+1,j}$. Here is the layer "A" where $X_{i+1} = E(X_i; HMAC(Y_j))$. The next state-formulae are computed as follows:

$$\begin{pmatrix} \mathbf{S}^a_{i+1,j} \\ \mathbf{S}^b_{i+1,j} \\ \mathbf{S}^c_{i+1,j} \end{pmatrix} =_{[A=]} \begin{pmatrix} W+(1-W)N & 0 & (1-W)N \\ 0 & 1 & 0 \\ (1-W)(1-N) & 0 & 1-(1-W)N \end{pmatrix} \cdot \begin{pmatrix} \mathbf{S}^a_{i,j} \\ \mathbf{S}^b_{i,j} \\ \mathbf{S}^c_{i,j} \end{pmatrix}$$

For example, case (a) can be transformed to two cases (a) and (c), but not (b) since $Y_j \neq Y'_j$. The probability of the transition (a)-(a) is $W + (1-W)N$, having two cases: either HMAC produces a matching key with probability W, then the output from E would collide with probability 1 since $X_i = X'_i$ in the case (a); otherwise, the output from E can collide by chance with probability N when the keys to E are distinct, which is in turn has the probability $(1-W)$. A similar reasoning is applied to other transition probabilities.

$\mathbf{S}_{i,j} \mapsto \mathbf{S}_{i,j+1}$. Here is the layer "B" where $Y_{j+1} = Y_j \oplus PRNG(X_i)$. The probability transition matrix is as follows:

$$\begin{pmatrix} \mathbf{S}^a_{i,j+1} \\ \mathbf{S}^b_{i,j+1} \\ \mathbf{S}^c_{i,j+1} \end{pmatrix} =_{[B=]} \begin{pmatrix} 1 & 0 & 0 \\ 0 & M & M \\ 0 & 1-M & 1-M \end{pmatrix} \cdot \begin{pmatrix} \mathbf{S}^a_{i,j} \\ \mathbf{S}^b_{i,j} \\ \mathbf{S}^c_{i,j} \end{pmatrix}$$

The overall transition matrix for 5 rounds in ALF-L can be derived by $A \cdot B \cdot A \cdot B \cdot A$.

In our mini-experiments (see below) we found that for a maximum security we must have $w \geq n$. Thus, let $M = 2^{-m}$, $N = 2^{-n}$, and assign the minimum $w = n$, the advantages are then derived as follows:

$$A = \begin{pmatrix} 2N-N^2 & 0 & N-N^2 \\ 0 & 1 & 0 \\ N^2-2N+1 & 0 & N^2-N+1 \end{pmatrix} \qquad B = \begin{pmatrix} 1 & 0 & 0 \\ 0 & M & M \\ 0 & 1-M & 1-M \end{pmatrix}$$

$A \cdot B \cdot A \cdot B \cdot A =$

$$\begin{pmatrix} N(1-M)(1+N^5-6N^4+11N^3+MN^5-4MN^4+6MN^3-4MN^2-7N^2+MN+\frac{-N^5+6N^4-12N^3+8N^2}{1-M}) & N(1-M)(1+MN^3-2MN^2-N^2+MN) & N(1-M)(1+N^5-5N^4+7N^3+MN^5-3MN^4+4MN^3-3MN^2-4N^2+MN+\frac{-N^5+5N^4-8N^3+4N^2}{1-M}) \\ M(1-N)(1+MN^3-2MN^2-N^2+NM) & M(1-N)(1+NM+\frac{N^2}{1-N}) & M(1-N)(1+MN^3-MN^2-N^2+NM+\frac{N^2}{1-N}) \\ (1-N)(1-M)(1+N^5M-3N^4M+4N^3M-3N^2M+N^5-5N^4+7N^3-4N^2+NM+\frac{-N^5+5N^4-8N^3+4N^2}{(1-M)}) & (1-N)(1-M)(1+MN^3-MN^2-N^2+NM+\frac{N^2}{1-N}) & (1-N)(1-M)(1+N^5M-2N^4M+3N^3M-2N^2M+N^5-4N^4+4N^3-3N^2+NM+\frac{N^2}{1-N}+\frac{-N^5+4N^4-5N^3+2N^2}{1-M}) \end{pmatrix}$$

All equations of the resulting matrix are already brought to the shape similar to Eq. (23) (highlighted in red) and therefore the advantages can be seen directly, with dominating probabilities highlighted in blue. In all cases where $m < n$ the advantage of the order $NM = 2^{-(n+m)}$ is dominating, and when $m > n$ the dominating advantage becomes $N^2 = 2^{-2n}$, up to a coefficient. Therefore, for a pair of queries the absolute advantage in any of the (a-c)-transition cases in this model is upper bounded by

$$|\epsilon_{a,b,c}| < 2 \cdot \max\{NM, N^2\}$$

where the coefficient 2 is added to negate the smaller contributions to the advantage, such as $\pm MN^2$, etc. For q queries there are $q(q-1)/2$ pairs of samples, totalling to the sum of advantages over all pairs in q queries by:

$$|\epsilon^{(q)}_{Adv}| < \frac{q^2}{2^{n+\min(n,m)}}.$$

Since $n \geq 128$ bits in ALF-L, the design can be distinguished from random by collecting q queries of the order either the square root of the domain size (X, Y), or at least 2^{128} for larger domains.

Four rounds for ALF-L are less secure. As an example, let us apply the same method but for ALF-L having only 4 rounds, we then receive the following transition matrix:

$B \cdot A \cdot B \cdot A =$

$$\begin{pmatrix} N^4M - 3N^3M + 3N^2M - NM - N^3 + N^2 + N & MN^2 - NM - N^2 + N & N^4M - 2N^3M + 2N^2M - NM - N^3 + N \\ -N^4M^2 + 3N^3M^2 - 3N^2M^2 + NM^2 + N^3M - N^2M - NM + M & NM^2 - N^2M^2 + N^2M + M - NM & -N^4M^2 + 2N^3M^2 - 2N^2M^2 + NM^2 + N^3M - NM + M \\ N^4M^2 - 3N^3M^2 + 3N^2M^2 - NM^2 - N^4M + 2N^3M - 2N^2M + 2NM - M + N^3 - N^2 - N + 1 & N^2M^2 - NM^2 + 2NM - 2N^2M - M + N^2 + 1 - N & N^4M^2 - 2N^3M^2 + 2N^2M^2 - NM^2 - N^4M + N^3M - 2N^2M + 2NM - M + N^3 - N + 1 \end{pmatrix}$$

One can easily see in the coordinate $(0, 1)$ that:

$$MN^2 - NM - N^2 + N = N(1 - M)(1-N)$$

which implies that the advantage is of the order $N = 2^{-n}$ thus $q \approx 2^{n/2}$ can be sufficient to distinguish ALF-L from random even when m is large. A similar result was received for $A \cdot B \cdot A \cdot B$.

Simulations for 5-round ALF-L. We also performed simulations by computing the $\log_2$ of the advantages having the transition matrix, with the following mini-experiments and results:

```
n=10, m=10, w=5 <= the case with insufficient
output from HMAC. The advantage is too strong
for the case (a)-(a) caused by a small w:
[-4.9998 -10.001 -15.091]
[-10.001 -14.955 -19.021]
[-15.091 -19.021 -19.958]

w=n=10, m=5 <= the case where m is small,
thus NM dominates as the advantage:
[-14.949 -15.048 -14.998]
[-15.048 -14.955 -15.001]
[-14.998 -15.001 -14.999]
```

```
n=10, m=10, w=20 <= the case with too
much of output bits from HMAC.
The result is not better than when w=n:
[-19.999 -20.000 -19.999]
[-20.000 -19.997 -20.000]
[-19.999 -20.000 -19.999]

w=n=10, m=40 <= the case where m is large,
thus N^2 dominates as the advantage:
[-20.001 -20.000 -30.000]
[-20.000 -19.998 -29.998]
[-30.000 -29.998 -39.997]
```

Limitations of the model. Finally, we note that the model used in this section is limited, since there may exist other second-order dependencies beyond the (a-c) relations. For example, two nonzero differentials ΔY_1 and ΔY_2 may also be related such as $\Delta Y_1 = \Delta Y_2$ or $\Delta Y_1 \neq \Delta Y_2$, which this model does not cover and considers ΔY_1 and ΔY_2 as independent. However, the model gives a good approximation as the number of layers gets larger.

F.5 Generic key-tweak management (KTM)

In previous sections we introduced ALF variants without giving details on the round keys generation. In this section we introduce the module called the Key-Tweak Management (KTM), the purpose of which is, among others, to generate a set of round keys, support a number of use cases with domain separation, absorbing the secret key, tweak, application ID, and a concrete encryption configuration. The KTM is a generic module covering all ALF instances.

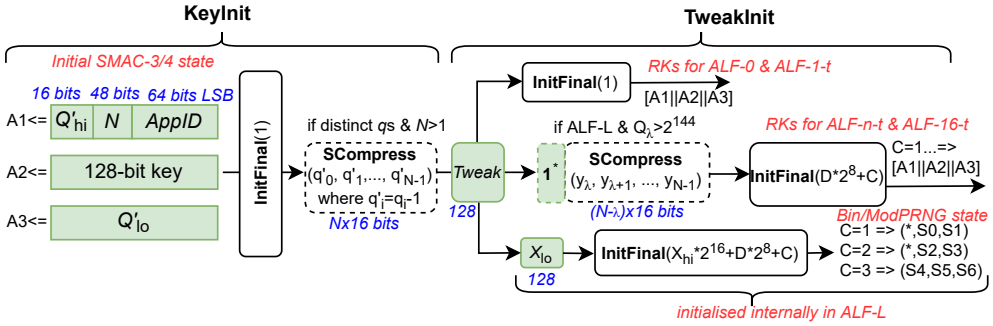

Figure 11: Key-Tweak Management and derivations of round keys and secret states for all ALF variants.

Algorithm 8 KTM components

function $\texttt{SMAC}_R(A1, A2, A3, M) \to (A1', A2', A3')$
  Get SMAC-3/4' shuffle: $\sigma_{42} = (7, 14, 15, 10, 12, 13, 3, 0, 4, 6, 1, 5, 8, 11, 2, 9)$
  return $(A1', A2', A3') \leftarrow (\sigma_{42}(A2 \oplus A3 \oplus M), \texttt{aesenc}(A1, M), \texttt{aesenc}(A2, M))$
function $\texttt{InitFinal}(A1, A2, A3, C \in \mathbb{N}_{32}) \to (A1', A2', A3')$
  return $(A1, A2, A3) \oplus \texttt{SMAC}_R^9(A1, A2, A3, C||\{0\}^{12})$ ▷ 9 rounds
function $\texttt{SCompress}(A1, A2, A3, N, v_0 \in \mathbb{N}_{16}, \ldots, v_{N-1} \in \mathbb{N}_{16}) \to (A1, A2, A3)$
  Derive the number of 128-bit aligned blocks $J = \lceil N/8 \rceil$
  Group each block of 8 v's into a 128-bit value, i.e.: $\forall k \in [0, J) : V_k = v_{8k+0}||v_{8k+1}||\ldots||v_{8k+7}$,; if $(8k + i) \geq N$ then assume $v_{8k+i} = 0$
  for $k = 0..J - 1$ **do**
    $(A1, A2, A3) \leftarrow \texttt{SMAC}_R(A1, A2, A3, V_k)$
    if $(k \mod 3) = 2$ then $(A1, A2, A3) \leftarrow \texttt{SMAC}_R(A1, A2, A3, \mathbf{1}^*)$ ▷ rate 3/4
  if $(b \mod 3) \neq 0$ then $(A1, A2, A3) \leftarrow \texttt{SMAC}_R(A1, A2, A3, \mathbf{1}^*)$

Let $Q'_{hi} = (Q - 1) \gg 128$, $Q'_{lo} = (Q - 1) \mod 2^{128}$, and $q'_i = q_i - 1$. In case when $N = 1$ we assume any modulus $Q \in [2, 2^{144}]$ is allowed. In case $N > 1$ and the plaintext has the same moduli $q_i = q$ we simply set $Q = q$. Finally, in case when $N > 1$ and we have arbitrary moduli, we set $Q = 1$.

Figure 11 depicts the general view of KTM. All ALF instances start with the `KeyInit` function that absorbs the current configuration parameters, such as $(Q - 1) \in \mathbb{N}_{144}, N \in$

$\mathbb{N}_{48}, \{q_i - 1\} \in \mathbb{N}_{16}^N$, a 128-bit secret key and a 64-bit application ID, $AppID$. After that the function `InitFinal`(C) is called with $C = 1$, followed by a 16-bit stream compression function `SCompress` in case $N > 1$ and arbitrary qs.

The next function is `TweakInit` that generates round keys for underlaying ALF variants. As the first step, the 128-bit tweak is compressed. For ALF-0 and ALF-1-t `InitFinal`(C) is called with $C = 1$ and 32/48 bytes for round keys are taken from the registers $(A1, A2, A3)$. For ALF-n-t and ALF-16-t (assuming Q and/or $Q_\lambda \leq 2^{144}$, without the Y-part), there are a few parallel calls to `InitFinal`$(D \cdot 2^8 + C)$ with $D = 0$ and $C = 1, 2, \ldots$ being a counter, until the required number of bytes are collected. Each of these calls gives 48 bytes from the resulting state $(A1, A2, A3)$. When enough bytes are collected, we then consecutively take n-byte blocks as round keys for the relevant ALF instance.

For ALF-L the case is a bit more complex, where `TweakInit` is called several times for each layer. When we encrypt the X-part with ALF-16-t, then the Y-part is compressed after the tweak; we call `InitFinal`$(D \cdot 2^8 + C)$ with $D = 1, 3, 5$ for the corresponding encryption layer ("A" on Figure 8), the counter C starts from 1 as before. When the Y-part is to be encrypted, then the state of `ModPRNG` is initialised by first pushing the lower 128 bits of X, then calling `InitFinal`$((X \gg 128) \cdot 2^{16} + D \cdot 2^8 + C \in \mathbb{N}_{32})$ with a 32-bit integer as the parameter that includes the remaining most significant 16-bits of X, and $D = 2, 4$ for the relevant layer ("B" on Figure 8). The counter goes as $C = 1, 2, 3$ from where we pick the 7 registers as the initial state of `ModPRNG`. The remaining parts are the definitions of `InitFinal` and `SCompress` given in Algorithm 8.

F.6 Software evaluation in more detail

This section complements Section 5 with more details on the performance measurements of ALF-n-t, as well as adding to the table cases with complementary ALF instances, such as ALF-L.

When the number of symbols $N > 1$, we also provide λ – the number of symbols that are (un)packed in the first X-part, see Appendix F.4. When $\lambda < N$ we have the case of ALF-L built on top of ALF-16-t and `ModPRNG`.

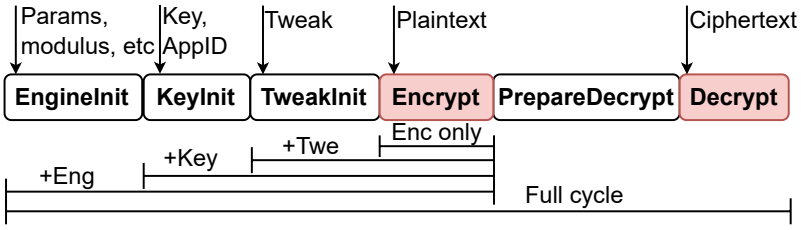

Figure 12: Performance test scenarios for the encryption direction.

Depending on a real application the performance can be even higher if some of the ALF API steps can be done once in the application, or much less frequent than encryption/decryption calls. Different performance test scenarios have been measured where one does only encryption, or adds to that the switch of the tweak ("+Twe"), and so on, these stages are shown on Figure 12 for the encryption, and similar stages can be derived for the decryption. The speed measurements are given on the right side of Table 13 in more detail. Note that the compiler seems to optimise variants differently, e.g., in some cases "Twe+Enc" is slightly faster than only "Enc", but here we cannot control that and present the results as received.

Table 13: Performance measurements. The two values 'max/avr' of each performance point correspond to the maximum and average speeds (in Msps), respectively.

Compare with other FPE algs (Full cycle)			Additional performance details for ALF							
			Encryption cycle				Decryption cycle			
ALF	FF1	FF3	Enc	+Twe	+Key	+Eng	Dec	+Twe	+Key	+Eng
C0. 'Yes/No', $N = 1$, Radix= 2, ALF-0b, $r = 32$										
4.2/3.4	–	–	620/500	5.3/4.1	4.9/3.9	4.9/3.9	746/593	5.1/4.1	4.8/3.9	4.8/3.8
C1. 'Credit card', $N = 16$, Radix= 10, $\lambda = 16$, ALF-6-6n, $r = 16$, $Q_\lambda =$ `0x2386f26fc10000`										
28/25	2.34/1.66	20.3/14.4	70/63	69/61	65/59	57/51	54/49	51/45	50/44	46/41
C2. 'IPv6', $N = 6$, Radix= 2^{16}, $\lambda = 6$, ALF-12-0b, $r = 12$, $Q_\lambda = 2^{96}$										
36/30	1.53/1.06	9.35/7.06	114/99	89/74	79/66	56/48	106/93	73/60	70/59	50/42
C3. 'IMSI', $N = 11$, Radix= 10, $\lambda = 11$, ALF-4-5n, $r = 18$, $Q_\lambda =$ `0x174876e800`										
25/22	2.09/1.48	15.44/11.44	67/61	65/57	61/55	52/47	52/46	47/41	45/40	41/36
C4. 'UTF-16 string', $N = 256$, Radix= 2^{16}, $\lambda = 9$, ALF-16-16b, $r = 12$, $Q_\lambda = 2^{144}$										
212/180	0.92/0.62	–	492/403	494/402	486/397	436/365	422/355	427/351	414/350	385/326
C5. 'English letters', $N = 1000$, Radix= 26, $\lambda = 30$, ALF-16-14n, $r = 12$										
			$Q_\lambda - 1 =$ `0x204b3fa059a3d91315cc9c2e42063fffffff`							
281/241	0.85/0.59	–	592/513	594/515	596/513	581/499	541/468	536/464	538/463	526/455
C6. 'Letters+digits', $N = 10000$, Radix= 36, $\lambda = 27$, ALF-16-12n, $r = 12$										
			$Q_\lambda - 1 =$ `0x0c066be3cd5688407d44a63fffffffffffff`							
427/366	0.10/0.07	–	858/740	859/738	858/743	866/742	854/737	855/736	858/735	852/733