Full key-recovery on ACORN in nonce-reuse and decryption-misuse settings

Colin Chaigneau, Thomas Fuhr, and Henri Gilbert

ANSSI, France

Abstract. ACORN is an authenticated encryption algorithm submitted by Wu to the CAESAR competition. The security analysis made by the designer relies on the following two assumptions: the nonces used to encrypt plaintexts must never be repeated, and no plaintext should be released after decryption if the tag verification fails. The designer does not make any claim on the security of ACORN if these conditions are not both fulfilled. In this note, we show that if one of these conditions falls, the 128-bit encryption key can be efficiently recovered by solving relatively small linear equation systems.

1 Introduction

ACORN is an AEAD (authenticated encryption with associated data) algorithm that was submitted by Wu to the CAESAR competition [4]. It encrypts plaintexts under a 128-bit key using a 128-bit nonce, and generates a 128-bit tag authentifying the plaintext along with associated data. Its design relies on a dedicated stream cipher, with an additional input bit used to make the internal state plaintext-dependent. It is based on the use of several linear feedback shift registers (LFSR) with a global quadratic feedback. Keystream bits are extracted from the state by applying a quadratic extraction function.

Unlike most stream ciphers, the global feedback function involves one external bit (of associated data or plaintext), making the state dependent on previously encrypted bits. Extra keystream bits generated after the end of encryption can be used to generate the tag. This idea has been used before to design stream ciphers such as Phelix [2]. Phelix has already been shown vulnerable to attacks (including key recovery) when the nonce is reused or if decrypted bits are released before verification of the tag [3]. Such attacks are potentially applicable to most ciphers constructed using this idea.

As a consequence, no security claim is made for ACORN in these settings, the designer even expects the cipher to be vulnerable in such settings. The aim of this note is to strengthen this statement by describing an attack that allows instant key recovery, with a small amount of chosen plaintexts encrypted using a fixed nonce. Our attack consists in building and solving a small system of affine equations, which are much more efficient than what is expected from the potential transposition of the differential attack against Phelix.

The only other published result on ACORN we are aware of has been released by Liu and Lin [1] and reveals the existence of (Key,IV) pairs (with distinct keys

and nonces) that generate the same state up to a clock difference. Such pairs can be found easily in chosen key settings. However, when an unknown key is involved, finding slid pairs requires a data complexity equivalent to a brute force attack on the key.

2 Description of ACORN

ACORN uses a 128-bit key and a 128-bit nonce. It allows to authenticate and encrypt up to 2^{64} bits of plaintext and 2^{64} bits of associated data. It is based on a stream cipher with a 293-bit internal state, to which one bit of external input is added during the update function. This external bit is used to load the key and the nonce, to process the additional data and to make the internal state dependent on the plaintext, to allow for the generation of a tag.

To perform an authenticated encryption the cipher executes the following steps:

1. An initialization step involving the loading of the key K and the nonce IV.
2. An associated data step where the associated data is loaded into the state.
3. An encryption step where the plaintext is combined with the keystream and loaded into the state
4. And finally the tag generation.

The 293-bit ACORN state is initially set to zero. Then, at each of the above steps, it is updated by iterating a state update function, as shown on Figure 1.

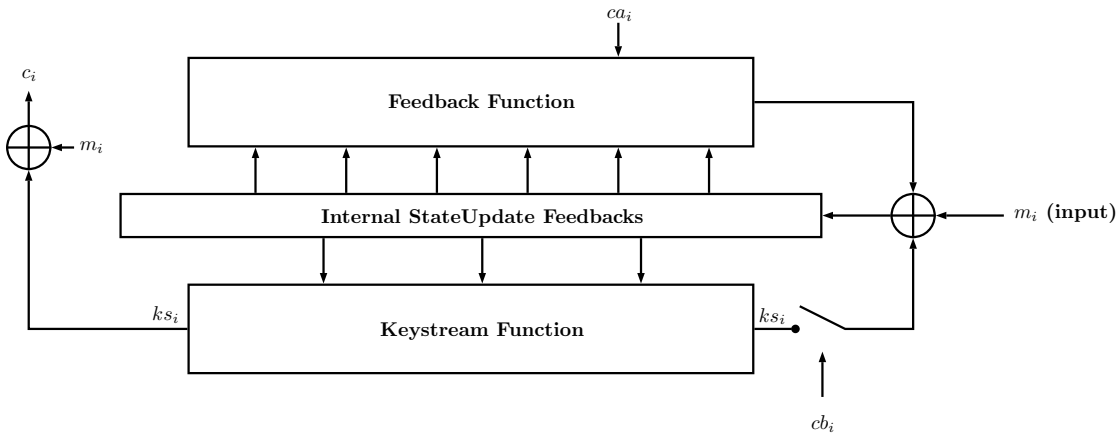

Fig. 1. Generic representation of the StateUpdate function.

The internal state of ACORN consists in a combination of six LFSRs of various sizes and a 4-bit buffer. The output bit of each LFSR is used as input

to the previous LFSR. The structure of the ACORN state is represented on Figure 2.

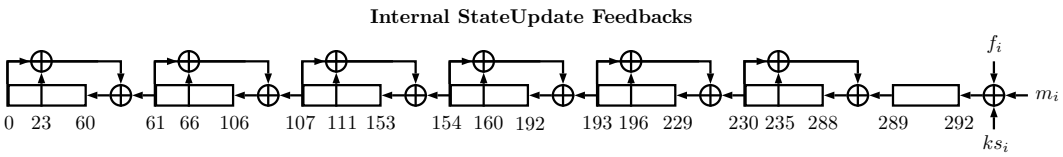

Fig. 2. Detailed representation of the internal state of ACORN, combining six LFSR.

Each invocation of the `StateUpdate` function triggers three operations, in the following order:

1. Computation of a feedback bit for each of the internal LFSRs,
2. Computation of a global nonlinear feedback bit and the keystream bit,
3. Advance of the registers and inclusion of the external (plaintext, key, nonce or AD) input bit.

More explicitely, if the state bits are denoted by x_0 to x_{293} the feedback and keystream functions are given by the following pseudo code where ca_i and cb_i represent two control bits that affect the computation of the global feedback bit f_i, and m_i represents an input bit (e.g. an associated data bit or a plaintext bit)

$x_{i,289} = x_{i,289} \oplus x_{i,235} \oplus x_{i,230}$;
$x_{i,230} = x_{i,230} \oplus x_{i,196} \oplus x_{i,193}$;
$x_{i,193} = x_{i,193} \oplus x_{i,160} \oplus x_{i,154}$;
$x_{i,154} = x_{i,154} \oplus x_{i,111} \oplus x_{i,107}$;
$x_{i,107} = x_{i,107} \oplus x_{i,66} \oplus x_{i,61}$;
$x_{i,61} = x_{i,61} \oplus x_{i,23} \oplus x_{i,0}$;

$ks_i = x_{12} \oplus x_{154} \oplus MAJ(x_{235}, x_{61}, x_{193})$;
$f_i = 1 \oplus x_0 \oplus x_{107} \oplus ca_i x_{196} \oplus MAJ(x_{244}, x_{23}, x_{160}) \oplus cb_i ks_i$;

for j from 0 to 292 do
  $x_{i+1,j} = x_{i,j+1}$;
end for;
$x_{i+1,292} = f_i \oplus m_i$;

MAJ and CH are the two non-linear boolean functions used in ACORN, they are defined by :

$$\begin{aligned} MAJ(x,y,z) &= xy \oplus xz \oplus yz \\ CH(x,y,z) &= xy \oplus \bar{x}z \\ &= xy \oplus (x \oplus 1)z \end{aligned}$$

A more detailed representation of the feedback and keystream function is given on Figure 3.

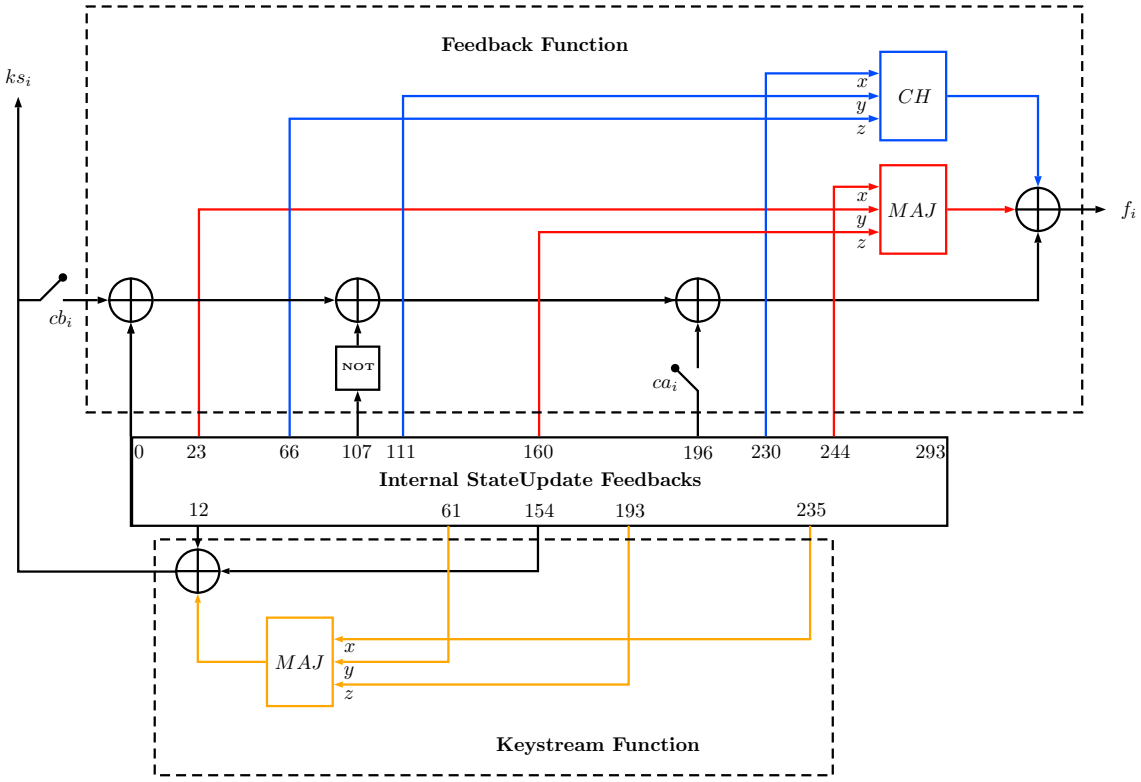

Fig. 3. Detailed representation of the global feedback and keystream generation functions.

The schedule of the four steps of ACORN and the values of m_i, ca_i, cb_i bits that govern each iteration of the update function are summarized in the Table 1.

Each ciphertext bit c_i is computed by the XOR-addition $c_i = p_i \oplus ks_i$ and the tag T is given by the last t bits of keystream (t is the length of the tag between 64 bits and 128 bits).

3 Key recovery attack

In this Section we show that the key can be recovered in the chosen plaintext setting, provided a nonce is repeated at least four times. Then, we show how to adapt our observation to the decryption-misuse setting.

	Initialization			AD			Encryption			Tag generation	
Length	128	128	1280	Up to 2^{64}	256	256	Up to 2^{64}	256	256	512	
m_i	K	IV	$10\ldots0$	AD	$10\ldots0$	$0\ldots0$	P	$10\ldots0$	$0\ldots0$	$0\ldots0$	
ca_i	1			1		0	1		0	1	
cb_i	1			1			0			1	
Output							C				T

Table 1. Summary of the values of the control bits and the external bits through the encryption process

3.1 Summary of our attack

Our attack works as follows. If a given nonce is used to encrypt several messages that share the same associated data field, the internal state of ACORN is identical after the processing of associated data during all the encryptions. Therefore, we aim at gathering information on this value of the internal state.

To achieve it, we exploit the fact that keystream bits depend on the plaintext after a few rounds. This enables us to derive a system of linear equations on the internal state. This system has potentially several solutions, that are then verified exhaustively.

Then, once the full internal state is recovered, we can step the cipher backwards to recover the full key, which enables us to decrypt any ciphertext regardless of the nonce.

3.2 A property of the MAJ function

Our attack relies on an algebraic property of the MAJ function. If we know the values of $a = MAJ(x, y, z)$ and $b = MAJ(\overline{x}, y, z)$, we can recover 2 affine relations on x,y and z. A first relation is trivially obtained by summing these values, as we obtain the derivative in x of the MAJ function, which has degree 1:

$$MAJ(x, y, z) \oplus MAJ(\overline{x}, y, z) = y \oplus z = a \oplus b.$$

Surprisingly, we can also get a second linear relation. Indeed, we can rewrite MAJ as

$$MAJ(x, y, z) = (x \oplus y)(y \oplus z) \oplus y,$$

and replace $(y \oplus z)$ by $(a \oplus b)$, due to the first equation. We get another linear relation:

$$(a \oplus b)(x \oplus y) \oplus y = a.$$

3.3 A key observation

In the remainder of this Section, we focus on the step of the authenticated encryption process in which the plaintext is processed (*i.e.* loaded into the internal state and encrypted by xor-ing the keystream). We denote step i the iteration of the state update function involving plaintext bit p_i, and $S_{j,k}$ the k-th bit of the internal state before step j. When the step number is clear from the context, we might omit it and denote S_k the k-th bit of the state.

Let us consider the encryption of two plaintexts P and P' with a common associated data field under the same nonce. Let us suppose that the first bits of the plaintexts P_0 and P'_0 differ. Due to the definition of the feedback function, the keystream bits ks_j and ks'_j are equal up to step 57. On step 58, the difference introduced in the plaintexts reaches the bit S_{235}, which enters a MAJ function involved in the computation of the keystream bit.

Let us denote x_j the bits of the internal state at that point. The encryption of P naturally gives the following equation:

$$ks_{58} = c_{58} \oplus p_{58} = MAJ(x_{61}, x_{193}, x_{235}) \oplus x_{154} \oplus x_{12}.$$

Similarly, the encryption of P' gives

$$\begin{aligned} ks'_{58} = c'_{58} \oplus p'_{58} &= MAJ(x'_{61}, x'_{193}, x'_{235}) \oplus x'_{154} \oplus x'_{12} \\ &= MAJ(x_{61}, x_{193}, \overline{x_{235}}) \oplus x_{154} \oplus x_{12}, \end{aligned}$$

as only bit 235 of the state differs between both encryptions. We now have two quadratic equations in the state bits. We can use the technique described above to recover 2 affine equations in these bits:

$$\begin{aligned} ks_{58} \oplus ks'_{58} &= MAJ(x_{61}, x_{193}, x_{235}) \oplus MAJ(x_{61}, x_{193}, \overline{x_{235}}) \\ &= x_{61} \oplus x_{193} \\ ks_{58} &= MAJ(x_{61}, x_{193}, x_{235}) \oplus x_{12} \\ &= (ks_{58} \oplus ks'_{58})(x_{193} \oplus x_{235}) \oplus x_{193} \oplus x_{154} \oplus x_{12}. \end{aligned}$$

3.4 Gathering a system of linear equations

This observation can be extended as follows. If the first difference on the plaintext is introduced at step i with $0 \leq i \leq 57$, then this difference can first affect the keystream on bit ks_{i+58}. As shown above, the resulting keystream bits ks_{i+58} and $ks_{i'+58}$ give two affine equations on bits $S_{i+58,12}$, $S_{i+58,61}$, $S_{i+58,154}$, $S_{i+58,193}$ and $S_{i+58,235}$ of the state, which can themselves be expressed as affine combinations of the bits x_j of the state at step 58.

This method enables us to get up to 116 linear equation with 293 unknowns x_j that represent the values of $S_{58,j}$. This is not enough to reduce the complexity of an exhasutive search below 2^{128}. However, one can get more linear equations

by defining extra unknowns y_k. Indeed, after step 116, the bit S_{235} is a nonlinear function of the state at step 58. We can therefore define

$$y_k = S_{k-116,235}.$$

Therefore, each new unknown enables us to express all the bits $S_{u,v}$ of the state (for $u \geq 116$ and $0 \leq v \leq 235$) as an affine function of the x_j's and the y_k's for one more step, leading to 2 extra affine equations. Provided that all these equations are linearly independent, the system has a unique solution if 177 variables y_k are defined.

3.5 Minimizing the number of plaintexts

The attack described above requires the introduction of $58 + 177 = 235$ differences. However, one does not need to encrypt 235 message pairs. First, differences have to occur on distinct bit positions, and therefore one message can be shared by all the pairs. Then, up to 42 differences can be introduced on the same message pair. Indeed, if plaintexts P and P' are such that $P' = P \oplus (0^i|1^{42}|0^j)$, a difference will occur on bit 235 of the state from step $i + 58$ to step $i + 99$, and no difference will occur on bits 12, 61, 154 and 193. If one tries to use a longer string of differences, they will affect bit 193 of the state from step 100.

Putting it together, using $n + 1$ chosen plaintexts of length at least $42n + 58$ bits, one can get $84n$ affine equations with $293 + max(42n - 58, 0) = 235 + 42n$ unknowns (for $n \geq 2$). A rough idea of the complexity of the attack can be obtained by supposing that all these equations are independent and that they involve all the variables. The expected number of solutions to try is then $2^{235-42n}$, which falls below 2^{128} for $n \geq 3$.

Therefore, the expected complexity of the attack is 2^{109} for 4 chosen plaintexts with a common nonce, 2^{67} for 5 plaintexts, and 2^{25} for 6 plaintexts. For 7 plaintexts or more, the dominating part of the attack is the resolution of the affine equation system.

3.6 Recovering the key

Once a value of the internal state is recovered, the full 128-bit key can be computed by running the cipher backwards. Until the inversion of the IV processing, the bit m_i that is involved in the state update function is known. During the inversion of the key processing, the key bit k_i can be obtained using the knowledge that the internal state is initially 0^{293}.

3.7 Adaptation to the decryption-misuse setting

This attack can also be mounted in the decryption-misuse setting, *i.e.* in a chosen ciphertext setting in which the decrypted plaintexts are released even if the tag is not valid. If the adversary submits two ciphertexts with the same nonces and associated data fields, and a difference $0^i|1^{42}|0^j$, she gets 84 affine equation from the resulting plaintexts using the argument above. Indeed, the difference on the plaintexts starts with $0^i|1^{42}$ and the argument described above still works.

4 Experiments

In practice all equations are stored on a matrix where unknowns are represented by the columns. After computation of all linear equations we operate a Gaussian elimination to put the previous matrix under row echelon form giving us the default rank and in this case the number of unknowns we have to guess to recover the state.

With seven plaintexts trials give us an average of between 10 and 13 guesses to do. Since 2^{13} tests can be easily performed this is not a problem.

Furthermore, increasing the number of available plaintexts does not reduce the space of solutions. This suggests that some linear dependencies between equations cannot be avoided. This can be explained heuristically. First, let us notice that after step 58, all internal state bits depend on the variables $x_0, \dots x_{292}$ only through the 235 linear combinations $S_{58,0}, \dots S_{58,234}$. Then, let us focus on the system resulting from equations obtained after step 58. The dimension of the space of its solutions will be at least $293 - 235 = 58$, as it contains all values of $(x_0, \dots, x_{292})$ resulting in the same $(S_{58,0}, \dots S_{58,234})$. Increasing the number of available plaintexts, and thus of equations, cannot give more information on the solution once only the right value $(S_{58,0}, \dots S_{58,234})$ is a solution of the system.

Then, the 116 first equations (obtained from the differences introduced from step 0 to step 57) provide more information on variables $(x_0, \dots, x_{292})$. Nevertheless, we can notice that bits $x_0, \dots, x_{12}$ never enter directly into the function computing the keystream bit. They are therefore unlikely to be recovered. This is consistent with the observations on the dimension of the space of solutions made during our experimentations.

To verify if we did the right guesses we verify that our current value of the state generates the same cipher string that the referenced cipher.

Once we have recovered a fixed ACORN state it is easy to recover the key and the nonce by inverting the `StateUpdate` function from the beginning. It is possible since we know the input m. When we reach the step where the nonce was just included we use the fact that we have zero on the first bit of the state

Testing on a MAGMA implementation the attack takes an average time of 17.68 seconds to recover the key and the nonce.

5 Conclusion

Our results do not contradict the security claims made by the designer. In scenarios allowing for nonce-reuse or release of unverified plaintexts, practical attacks with surprisingly low data and time complexities enable a full key recovery, which demonstrates that this algorithm offers no security in these cases. On the other hand, our attack does not threaten ACORN when operated as advised by the designer.

References

1. Meicheng Liu and Dongdai Lin. Cryptanalysis of Lightweight Authenticated Cipher ACORN, June 2014. Posted on the crypto-competition mailing list.
2. Doug Whiting, Bruce Schneier, Stefan Lucks, and Frédéric Muller. Phelix: Fast Encryption and Authentication in a Single Cryptographic Primitive, 2005. eSTREAM, ECRYPT Stream Cipher Project Report 2005/027.
3. Hongjun Wu and Bart Preneel. Differential-linear attacks against the stream cipher phelix. In Alex Biryukov, editor, *Fast Software Encryption, 14th International Workshop, FSE 2007, Luxembourg, Luxembourg, March 26-28, 2007, Revised Selected Papers*, volume 4593 of *Lecture Notes in Computer Science*, pages 87–100. Springer, 2007.
4. Honjun Wu. ACORN: A Lightweight Authenticated Cipher. In *DIAC 2014: Directions in Authenticated Ciphers, Santa Barbara*, 2014.