Masking the Lightweight Authenticated Ciphers ACORN and Ascon in Software

Alexandre Adomnicai[1,2], Jacques J.A. Fournier[3], and Laurent Masson[1]

[1] Trusted Objects, Aix-en-Provence, France,
{a.adomnicai, l.masson}@trusted-objects.com
[2] Mines Saint-Étienne, CEA-Tech, Centre CMP, Gardanne, France,
[3] Univ. Grenoble Alpes, CEA-LETI, DSYS, Grenoble, France,
jacques.fournier@cea.fr

Abstract. The ongoing CAESAR competition aims at finding authenticated encryption schemes that offer advantages over AES-GCM for several use-cases, including lightweight applications. ACORN and Ascon are the two finalists for this profile. Our paper compares these two candidates according to their resilience against differential power analysis and their ability to integrate countermeasures against such attacks. Especially, we focus on software implementations and provide benchmarks for several security levels on an ARM Cortex-M3 embedded microprocessor.

Keywords: ACORN, Ascon, DPA, Masking, Side-Channel Attacks

1 Introduction

Authenticated encryption (AE) schemes provide confidentiality, integrity and authenticity at once. The classical way to achieve such a construction is to combine several cryptographic primitives, typically an encryption algorithm for confidentiality and a message authentication code (MAC) for integrity and authenticity. Besides of being non optimal in terms of performance, the generic combination may lead to implementation errors. Therefore, several mode of operations for block ciphers have been designed to provide efficient and secure AE constructions such as Counter-with-CBC-MAC (CCM) [29], Galois/Counter Mode (GCM) [10] and Offset Codebook Mode (OCB) [22]. Moreover, other algorithms than operation modes have been investigated to design dedicated AE ciphers. Despite the variety of available designs, because of a lack of widespread support and patenting issues, AES-GCM is currently the most widely used cipher for AE. However, numerous vulnerabilities regarding AES-GCM have been pointed out by the cryptographic community [20,6]. It is within this context that, in January 2013, the CAESAR (Competition for Authenticated Encryption: Security, Applicability, and Robustness) contest has arisen with the objective to come out with AE schemes that offer advantages over AES-GCM and are suitable for widespread adoption. In March 2014, 57 heterogeneous submissions have been received. In July 2016, the CAESAR committee specified three use cases in order to classify the candidates. In March 2018, the finalists for each type of use case were announced: ACORN [30] and Ascon [9] for lightweight applications, AEGIS [32], MORUS [31] and OCB [18] for high-performance applications, and COLM [2] and Deoxys-II [15] for defense in depth in misuse scenarios.

Our contribution. In this paper, we focus on ACORN and Ascon, both finalists for the lightweight application profile. We compare them in terms of resilience against side-channel analysis but also in terms of performance when implemented with countermeasures against such attacks. While numerous benchmarks are available for the CAESAR candidates, most of them do not take countermeasures against physical attacks into account. Nevertheless, this is

a significant criteria to consider since it can definitely be a game-changer in terms of performance. For instance, ARX designs are very efficient in software but lose their advantage once protected against power analysis [1]. A paper that benchmarks some CAESAR candidates, including ACORN and Ascon, when implemented with countermeasures against side-channel attacks has been recently published [8]. However, it only concerns hardware implementations, leaving embedded software implementations high and dry. Moreover, although the authors proposed a threshold implementation of ACORN , they focused on leakage detection rather than identifying potential attack paths. Therefore, the vulnerability of ACORN to differential power analysis (DPA) has not been explicitly established yet. In this paper, we clarify this point by introducing the first DPA against this cryptographic primitive. This result allows us to justify the need of countermeasures and to propose a dedicated masking scheme. Regarding Ascon, we briefly recall the previous security evaluations that have been published in order to integrate suitable countermeasures. Especially, we highlight that for both candidates, it is sufficient to apply countermeasures to specific phases of the algorithm rather than to its entirety. All the implementations discussed in this paper are implemented in assembly language and executed on an ARM Cortex-M3 embedded microprocessor in order to compare both candidates with side-channel attacks and software implementations in mind.

Outline. First, we briefly recall the two finalists ACORN and Ascon before discussing their vulnerabilities against differential power analysis. Especially, we introduce a DPA attack against ACORN and highlight that only a partial knowledge of the initialization vector is necessary to recover the encryption key. These results allow us to propose a secure implementation for each finalist by taking inspiration from the optimized code proposed by the designers and masking schemes from the litterature. Finally, we present a performance benchmark between ACORN and Ascon, with and without first-order masking.

2 ACORN

ACORN is a stream cipher based authenticated encryption with associated data (AEAD) algorithm. AEAD schemes allow to also include information that does not need to be encrypted but of which the integrity and authenticity needs to still be guaranteed. ACORN uses a 128-bit key, a 128-bit initialization vector (IV) and produces a 128-bit authentication tag. Its internal state is 293-bit long and consists in the concatenation of six LFSRs as shown in Fig.1.

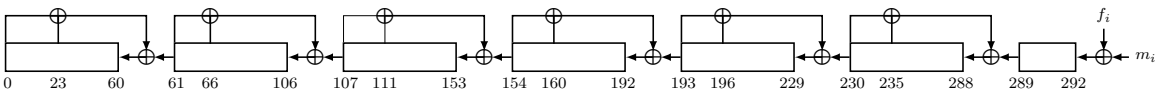

Fig. 1: The concatenation of 6 LFSRs in ACORN . f_i and m_i indicate the overall feedback bit and the message bit for the i^{th} step, respectively.

ACORN relies on three main functions: an output keystream generation function, a nonlinear feedback function, and a state update function. The keystream generation function is defined by

$$\kappa(S) = S_{12} \oplus S_{154} \oplus Maj(S_{235}, S_{61}, S_{193}) \oplus Ch(S_{230}, S_{111}, S_{66}) \tag{1}$$

where $Maj(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$ and $Ch(x, y) = (x \wedge y) \oplus (\neg x \wedge z)$. The nonlinear feedback function is defined by

$$\varphi(S, k, ca, cb) = S_0 \oplus \neg S_{107} \oplus Maj(S_{244}, S_{23}, S_{160}) \oplus (ca \wedge S_{196}) \oplus (cb \wedge k)\,. \tag{2}$$

The variables ca and cb allow to define different variants of the feedback function for the four phases of the cipher: initialization, additional data processing, encryption and tag generation. All of them use the state update function,the core of ACORN, which is defined in Alg.1.

Algorithm 1 $\texttt{StateUpdate}(S, m, ca, cb)$

$S_{289} \leftarrow S_{289} \oplus S_{235} \oplus S_{230}$ ▷ Update using six LFSRs
$S_{230} \leftarrow S_{230} \oplus S_{196} \oplus S_{193}$
$S_{193} \leftarrow S_{193} \oplus S_{160} \oplus S_{154}$
$S_{154} \leftarrow S_{154} \oplus S_{111} \oplus S_{107}$
$S_{107} \leftarrow S_{107} \oplus S_{66} \oplus S_{61}$
$S_{61} \leftarrow S_{61} \oplus S_{23} \oplus S_0$
$k \leftarrow \kappa(S)$ ▷ Keystream bit generation
$f \leftarrow \varphi(S, k, ca, cb)$ ▷ Nonlinear feedback bit generation
for i from 0 to 291 **do**
  $S_i \leftarrow S_{i+1}$ ▷ Shift the state
end for
$S_{292} \leftarrow f \oplus m$

Initialization. The initialization phase takes as input the encryption key and the IV. First, the entire state is initialized to zero. Then the state is updated for 1792 steps as described in Alg.2.

Algorithm 2 $\texttt{AcornInit}(S, K, IV)$

$(S_0, ..., S_{292}) \leftarrow (0, ..., 0)$ ▷ Initialize the state to zero
for i from 0 to 127 **do**
  $S \leftarrow \texttt{StateUpdate}(S, K_i, 1, 1)$
end for
for i from 0 to 127 **do**
  $S \leftarrow \texttt{StateUpdate}(S, IV_i, 1, 1)$
end for
$S \leftarrow \texttt{StateUpdate}(S, K_0 \oplus 1, 1, 1)$
for i from 1 to 1535 **do**
  $S \leftarrow \texttt{StateUpdate}(S, K_{i \bmod 128}, 1, 1)$
end for

Additional Data Processing. After the initialization step, the associated data is used to update the state. The state is updated for at least 256 steps, even if there is no associated data to process.

Encryption. At each step of the encryption, one bit from the plaintext is encrypted. The state is updated for at least 256 steps, even if there is no plaintext to encrypt.

Finalization. At the end, an n-bit authentication tag is computed. The state is updated 768 times and the tag consists of the last n keystream bits generated.

3 Ascon

Ascon is an AEAD algorithm based on duplex sponge modes [4]. It uses a 128-bit key, a 128-bit IV and produces a 128-bit authentication tag. The internal state is 320-bit long and is represented by five 64-bit registers, noted x_0 to x_4. In the same way as ACORN, Ascon processing consists of four phases as shown in Fig.4.

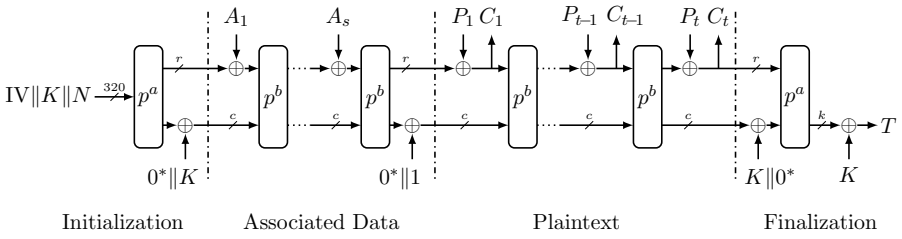

Fig. 2: ASCON cipher

Initilization The initial state is built from the encryption key and the initial vector before applying $a = 12$ rounds of the transformation p. Finally, the secret key is XORed to the 128 last bits of the state.

Additional Data Process Each additional data block is absorbed into the state before applying b rounds of the transformation p. Then, even if there is no additional data to process, the last bit of the state is swapped to set up a domain separation in order to prevent attacks that change the role of plaintext and associated data blocks.

Encryption Each plaintext block is duplexed into the state, producing a ciphertext block. Except for the last block, the state is then updated by b rounds of the transformation p.

Finalization The encryption key is XORed to the internal state before applying $a = 12$ rounds of the transformation p. Finally, the n-bit tag consists of the last n bits of the state XORed with the key.

The main component of Ascon is a 320-bit substitution-permutation network (SPN) based transformation, noted p. It consists of three functions: a constant adder on top of nonlinear and linear layers. The nonlinear layer consists in 64 parallel applications of a 5-bit S-box to each bit-slice of the five registers x_0,...,x_4. Therefore, the S-box is intended to be implemented in its bitsliced form as shown in Fig.3. Regarding the linear layer, it provides diffusion within each register separately.

```
x0 ^= x4;       x4 ^= x3;       x2 ^= x1;
t0 = ~x0;       t1 = ~x1;       t2 = ~x2;       t3 = ~x3;       t4 = ~x4;
t0 &= x1;       t1 &= x2;       t2 &= x3;       t3 &= x4;       t4 &= x0;
x0 ^= t1;       x1 ^= t2;       x2 ^= t3;       x3 ^= t4;       x4 ^= t0;
x1 ^= x0;       x0 ^= x4;       x3 ^= x2;       x2 = ~x2;
```

Fig. 3: Instructions to implement the Ascon S-box in a bitsliced manner

There are two recommended parameters for Ascon. The only two differences are the block size and the number of rounds b of the transformation p when applied to the state after absorbing the plaintext blocks. Ascon-128 works on 64-bit data blocks with $b = 6$ while

Ascon-128a works on 128-bit data blocks with $b = 8$. Because Ascon-128 is the primary recommendation by the designers, we focus on this version and Ascon refers to Ascon-128 throughout the rest of this paper.

4 Differential Power Analysis

Side-channel attacks do not target the encryption algorithm itself, but side effects that occur during the execution of an implementation such as computation time, electromagnetic (EM) fields and power consumption. ACORN and Ascon do not need any conditional branches depending on the internal state, nor do they employ look-up tables. Although they are generally not vulnerable to timing attacks, they remain vulnerable to power or EM related attacks such as DPA [17].

4.1 ACORN

DPA against stream ciphers can be more challenging compared to block ciphers because the keystream is computed independently from the plaintext, which interferes in the relationship between known values and the secret key. As a result, side-channel attacks against streamciphers usually focus on the initialization phase or resynchronization mechanisms rather than the encryption step itself [26,12]. While several works have been undertaken to show how ACORN can be defeated using fault attack [33,24], to the best of our knowledge, there is no similar study with regards to side-channel attacks.

In the case of ACORN where the initialization is repeated each time a packet needs to be encrypted, an adversary who has knowledge of IVs could then exploit their interactions with the encryption key during this phase. An interesting feature of ACORN regarding DPA is the way the initialization is processed. Instead of simply loading the key and the IV into the state and then updating it for a certain number of steps, ACORN initalizes the state at zero and updates it using the key and the IV as inputs. As the keystream and the 196^{th} bits are used to update the state (*i.e.* $ca = cb = 1$), carrying a DPA during the initialization does not lead to a direct key recovery as only terms composed of several key bits can be retrieved.

Actually, before updating the state using the IV, the state is as follows

$$\begin{aligned}
(S_0, ..., S_{164}) &= (0, ..., 0) \\
(S_{165}, ..., S_{198}) &= (\neg K_0, ..., \neg K_{33}) \\
(S_{199}, ..., S_{201}) &= (K_{34} \oplus K_0, ..., K_{36} \oplus K_2) \\
(S_{202}, ..., S_{218}) &= (\neg K_{37} \oplus K_3 \oplus K_0, ..., \neg K_{53} \oplus K_{19} \oplus K_{16}) \\
(S_{219}, ..., S_{223}) &= (K_{54} \oplus K_{20} \oplus K_{17} \oplus K_0, ..., K_{58} \oplus K_{24} \oplus K_{21} \oplus K_4) \\
(S_{224}, ..., S_{229}) &= (\neg K_{59} \oplus K_{25} \oplus K_{22} \oplus K_5 \oplus K_0, ..., \neg K_{64} \oplus K_{30} \oplus K_{27} \oplus K_{10} \oplus K_5) \\
(S_{230}, ..., S_{261}) &= (\neg K_{65} \oplus K_{11} \oplus K_6, ..., \neg K_{96} \oplus K_{42} \oplus K_{37}) \\
(S_{262}, ..., S_{272}) &= (K_{97} \oplus K_{43} \oplus K_{38} \oplus f_{97}, ..., K_{107} \oplus K_{53} \oplus K_{48} \oplus f_{107}) \\
(S_{273}, ..., S_{288}) &= (\neg K_{108} \oplus K_{54} \oplus K_{49} \oplus K_0 \oplus f_{108}, ..., \neg K_{123} \oplus K_{69} \oplus K_{64} \oplus K_{15} \oplus f_{123}) \\
(S_{289}, ..., S_{292}) &= (\neg K_{124} \oplus f_{124}, ..., \neg K_{127} \oplus f_{127})
\end{aligned} \tag{3}$$

where f_i defines the nonlinear feedback bit.

$$f_i = \begin{cases} 1 & \text{if } 0 \leq i \leq 96 \\ K_{i-97} & \text{if } 97 \leq i \leq 99 \\ (\neg K_{i-58}) \wedge (\neg K_{i-100}) \oplus K_{i-97} & \text{if } 100 \leq i \leq 111 \\ (K_{i-58} \oplus K_{i-112}) \wedge (\neg K_{i-100}) \oplus K_{i-97} & \text{if } 112 \leq i \leq 116 \\ \neg(K_{i-58} \oplus K_{i-112} \oplus K_{i-117}) \wedge (\neg K_{i-100}) \oplus K_{i-97} & \text{if } 117 \leq i \leq 127 \end{cases} \tag{4}$$

From step 128 to step 255, each bit of the IV will be XORed with the nonlinear feedback bit. Running a DPA against this operation would not directly return the key bits but combinations of them. Moreover, this approach only works for the first 49 bits as the corresponding nonlinear feedback bits only depend on key bits. That is no longer the case for the next bits because the values f_{i+128} for $i \geq 49$ are also IV-dependent and therefore, not constant anymore. However, it is still possible to isolate the IV bit in order to express f_i in terms of unknown constants (*i.e.* key-dependent only) bits. As an example, Eq. 5 details how it can be done for f_{177}.

$$\begin{aligned} f_{177} &= Maj(S_{244}, S_{23}, S_{160}) \oplus S_{196} \oplus S_{154} \oplus Maj(S_{235}, S_{61}, S_{193}) \oplus 1 \\ &= S_{244} \wedge S_{160} \oplus S_{196} \oplus S_{154} \oplus Maj(S_{235}, S_{61}, S_{193}) \oplus 1 \\ &= (f_{128} \oplus K_{69} \oplus K_{10} \oplus K_{15} \oplus K_{74} \oplus K_{15} \oplus K_{20} \oplus IV_0) \wedge S_{160} \oplus S_{196} \oplus S_{154} \oplus Maj(S_{235}, S_{61}, S_{193}) \oplus 1 \\ &= f'_{177} \oplus IV_0 \wedge S_{160} \end{aligned} \tag{5}$$

Therefore, a DPA against $f_{177} \oplus IV_{49}$ require to make hypotheses on two variables f'_{177} and S_{160} instead of f_{177}. Note that it increases exponentially the attack complexity as two unknown variables have to be considered. Moreover, the attack complexity increases throughout the state update as more and more IV-dependent bits are involved. Anyway, one can still run an attack against several components as shown above. The number of hypothetical variables to consider in order to successfully carry a DPA depends on the nonlinear feedback bit index as summarised in Table 1.

Table 1: Number of additional variables x to guess when running a DPA against $f_{128+i} \oplus IV_i$ for $i \in \mathcal{I}$

$\mathcal{I}$	$[0, 48]$	$[49, 57]$	$[58, 62]$	$[63, 96]$	$\cdots$	$[126, 127]$
x	0	1	2	3	$\cdots$	33

Consequently, unless the attacker is able to target each bit independently (*e.g.* hardware implementation or enough traces to compute a partial correlation [28]), a DPA becomes too computationally expensive for the last bits to be practical. However, it is not necessary to attack all the 128 XOR operations to recover the entire key. Indeed, all key bits K_0 to K_{127} are involved in the first 49 combinations (*i.e.* from f_{128} to f_{176}). These latter can be seen as a nonlinear system of Boolean equations as defined in Eq.6.

$$\mathscr{F} = \begin{cases} \neg(K_{70} \oplus K_{11} \oplus K_{16}) \wedge \neg K_{28} \oplus K_{31} & = f_{128} \\ \neg(K_{71} \oplus K_{12} \oplus K_{17}) \wedge \neg K_{29} \oplus K_{32} & = f_{129} \\ \neg(K_{72} \oplus K_{13} \oplus K_{18}) \wedge \neg K_{30} \oplus K_{33} & = f_{130} \\ \neg(K_{73} \oplus K_{14} \oplus K_{19}) \wedge \neg K_{31} \oplus \neg(K_{34} \oplus K_0) & = f_{131} \\ \neg(K_{74} \oplus K_{15} \oplus K_{20}) \wedge \neg K_{32} \oplus \neg(K_{35} \oplus K_1) & = f_{132} \\ \neg(K_{75} \oplus K_{16} \oplus K_{21}) \wedge (K_{33} \oplus K_0) \oplus \neg(K_{36} \oplus K_2) \oplus \neg(K_{84} \oplus K_{25} \oplus K_{30}) \wedge \neg K_0 & = f_{133} \\ \neg(K_{76} \oplus K_{17} \oplus K_{22}) \wedge \neg(K_{34} \oplus K_1 \oplus K_0) \oplus K_{37} \oplus K_3 \oplus K_0 \oplus \neg(K_{85} \oplus K_{26} \oplus K_{31}) \wedge \neg K_1 & = f_{134} \\ \vdots & \\ (\neg(K_{69} \oplus K_{10} \oplus K_{15}) \wedge \neg K_{27} \oplus K_{30} \oplus K_{127} \oplus K_{68} \oplus K_9 \oplus K_{73} \oplus K_{19}) \wedge \cdots & = f_{176} \end{cases} \tag{6}$$

However, because $\mathscr{F}$ defines a system of 49 nonlinear equations with 128 unknowns, there are too many solutions. By taking advantage of leakages related to further nonlinear feedback bits, one can extend $\mathscr{F}$ by increasing the number of equations in order to reduce the set of solutions. In the rest of this section, $\mathscr{F}_i$ denotes the system of equations resulting from a DPA against $f_{128+j} \oplus IV_j$ for $0 \leq j \leq i$. Because additional hypotheses have to be considered

in order to isolate IV bits within nonlinear feedback bits, $\mathscr{F}_i$ reaches 128 equations from $i = 74$ according to Table 1.

The task of recovering the initial key bits from $\mathscr{F}_i$ can be reduced to a variant of the Boolean satisfiability (SAT) problem, which decides whether a given propositional formula in conjunctive normal form (CNF) is satisfiable. In our case, we know that the CNF derived from $\mathscr{F}_i$ is satisfiable, at least by the encryption key K. As a result, we are interested in obtaining all of the truth assignments of SAT, which is another problem called All-SAT. In order to evaluate the practical difficulty of solving $\mathscr{F}_{74}$, we computed the values it defines for an arbtirary random key K. Then, mainly two tools were used to solve the related All-SAT problem. First, the `bc2cnf` tool [16] allowed to translate the Boolean system of equations into CNF formulae, before giving it as input of the SAT solver `CryptoMiniSat5` [25]. An interesting feature of `CryptoMiniSat5` is the option `--maxsol n` which returns up to n solutions, allowing us to turn it into an All-SAT solver by providing a sufficiently large upper bound. Regarding $\mathscr{F}_{74}$, 329 values for K verify the system. By iteratively increasing the parameter i, we come to the conclusion that $\mathscr{F}_i$ has a unique solution if and only if $i \geq 81$. Indeed, $\mathscr{F}_{81}$ which defines a system of 158 equations and 128 unknowns has a single solution, which equals the random key K used for our experiments.

As a result, it is theoretically possible to extract the encryption key during the initialization of ACORN by means of DPA. Moreover, the attack introduced above does not require the knowledge of the entire IV since only the first 82 bits are necessary to return a single key. In cases where less IV bits are known, one can still run the attack to reduce the key search space in order to make a brute-force attack practical. Note that our approach makes the assumption that every nonlinear feedback bit is recovered correctly and thus does not take erroneous results into account. Such errors would make the related SAT problem unsatisfiable by the encryption key and thus the attack unsuccessful. To overcome this difficulty, the usage of pseudo-Boolean optimizers instead of SAT solvers in the presence of errors should be preferred [19]. Another error-tolerant technique that deals with inaccurate side-channel measurements was introduced in [34].

This is the only attack path we identified to mount a DPA against ACORN. However, note that the key is used again to update the state for the last 1536 steps of the initialization phase. Exploiting the leakages related to these computations would result in a way more complex Boolean system to solve. Still, we suggest to protect the entire initialization against DPA. Integrating countermeasures during the encryption process should not bring any benefit since it is assumed that the secret key of ACORN cannot be recovered from the state after initialization.

4.2 Ascon

The side-channel vulnerablities of Ascon have already been evaluated in two papers [13,23]. Both of them consist in focusing on the initialization step as explained below. At the beginning, the register x_0 is initialized with a constant, x_1 and x_2 are initialized with the key, and x_3 and x_4 are initialized with the IV. Because the variables x_3 and x_4 vary at each run and are public, it is possible exploit leakages related to the interaction between these variables and x_1 and x_2 during the first round of the permutation, in order to recover the encryption key.

Moreover, another trivial attack path against Ascon occurs during the finalization phase. Indeed, because the key is XOR-ed to the last permutation output to produce the tag, and because the tag a is public value, an attacker could take advantage of its knowledge to

perform a DPA using the XOR as selection function. Therefore, the intermediate value that is targeted consists of the last 128 bits of the state, just after the last permutation. Note that the number of key bits that can be recovered equals the tag length. Therefore, it could benefit to lightweight applications that truncate the tag to minimize the size of the packet. However even if the tag is half truncated, a brute force attack against the 64 remaining bits might be carried in practice.

These two attack paths are the only ones we identified to mount a DPA against Ascon. Indeed, a state recovery after the initialization or before the finalization neither leads to the recovery of the key nor allows universal forgeries. Therefore, DPAs against either the additional data processing or the encryption step could be applied but would only allow to recover n bits of the state because of the sponge construction, where n refers to the block size, without compromising the security of the algorithm.

5 The Masking Countermeasure

5.1 Overview

A common approach to thwart side-channel attacks is the use of *masking*. The principle of masking is to apply secret sharing at the implementation level. More precisely, it consists in blinding the processed values by means of random masks, so that intermediate variables are impossible to predict. Thus, an attacker has to analyze multiple point distributions, which exponentially increases the attack complexity with the number of shares. This kind of attack is called higher-order DPA and at least $n + 1$ shares have to be used to overcome an attack of order n.

Computing a linear function λ on the shares of a variable is straightforward. If we represent a native variable x by its Boolean shares x_i ($x = \bigoplus_i x_i$) we can compute the shares y_i of $y = \lambda(x)$ by simply applying λ on the individual shares ($y_i = \lambda(x_i)$). A function that consists of the addition of a constant (*e.g.* bitwise NOT) can be applied to a single share. On the other hand, computing a nonlinear function on the shares is a more challenging task. If all separate operations are performed on $n+1$ shares that are independent of sensitive variables, then it provides protection against n-order DPA. Regarding ACORN and Ascon, they can be implemented in software by relying exclusively on the following five bitwise operations: AND, XOR, NOT, shift and rotation. Therefore, the only nonlinear operation that have to be considered is the bitwise AND.

5.2 Secure AND Computation

Several methods have been proposed to compute secure AND gates with Boolean shares [27,7]. All these techniques require additional randomness during the computation to refresh the mask, even for first-order masking. Recently, a new masking scheme for secure AND gates that do not require any intermediate random has been published [5]. On top of that, it is faster than previous solutions since it only costs 7 basic operations, versus 8 for the previous most efficient known methods. We recall the sequence of operations in Alg. 3. Protection against high-order implementations for bitwise AND can be achieved thanks to the Ishai-Sahai-Wagner (ISW) scheme [14] which has a complexity of $\mathcal{O}(n^2)$ and requires to generate $n \times (n + 1)/2$ random numbers of the same size of the share, at each execution.

Algorithm 3 First-Order Secure AND [5]

Require: $x_1 = x \oplus x_2$; $y_1 = y \oplus y_2$; x_2 ; y_2
Ensure: (z_1, z_2) s.t. $z_1 \oplus z_2 = x \wedge y$
$z_1 \leftarrow (x_1 \wedge y_1) \oplus (x_1 \vee \neg y_2)$
$z_2 \leftarrow (x_2 \wedge y_1) \oplus (x_2 \vee \neg y_2)$

5.3 Masking Scheme for ACORN

As discussed in the previous section, ACORN only requires protection against DPA during the initialization phase. Our secure implementation of ACORN consists in generating enough randomness at the beginning of the initialization in order to mask the entire state. All the subsequent operations are performed using the shares in such a way that the processed variables are independent from the key. Note that even if the IV and the key are not directly masked, they will be rearranged in random shares once integrated into the internal state through the XOR with the nonlinear feedback bit. Finally, the shares are recombined at the end of the initialization, and the remaining computations are processed without masking.

5.4 Masking Scheme for Ascon

Contrary to ACORN, protection of the initialization phase is not sufficient for Ascon since its finalization phase is also vulnerable to DPA. Our proposal for a secure implementation of Ascon is as follows. First, the entire state is masked before applying transformation p. All the subsequent operations when applying the permutation are performed masked and the shares are recombined at the end of the initialization, after the addition of the key. Then, the processing of associated data and plaintext blocks are performed without masking. At the beginning of the finalization phase, the internal state is masked again. Note that reusing the random shares used during the initialization should not compromise the security of the masking scheme, saving the generation of additional random numbers. Finally, the recombination of the shares should be done only after the final key addition, since the DPA from the authentication tag targets the output of the last transformation p.

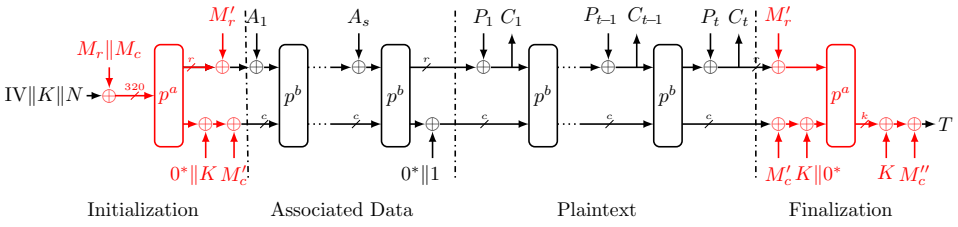

Fig. 4: Our proposed masking scheme for Ascon

6 Embedded Software Implementations

6.1 Methodology

All CAESAR submissions had to be accompanied by a software implementation in order to support public understanding of the cipher. For consistency purposes, all implementations

are provided in C language programming and are compliant with a specific API. Especially, the function to encrypt a message is defined below.

```
int crypto_aead_encrypt(unsigned char *c, unsigned long long *clen,
                        unsigned char *m, unsigned long long *mlen,
                        const unsigned char *ad, unsigned long long adlen,
                        const unsigned char *nsec, const unsigned char *npub, const unsigned char *k);
```

In order to integrate the implementations of ACORN and Ascon into our embedded platform, the core routine of each finalist (*i.e.* the state update and the transformation p, respectively) is implemented in assembly language. This choice is mainly motivated by performance and security purposes. Indeed, masking should be applied at the lowest level in order to avoid side effects from the compiler, such as instruction reordering or removal, and ensure that each register always has a distribution independent from any sensitive variable. Moreover, all auxiliary functions called from `crypto_aead_encrypt` are declared with the keyword `inline` to save function-call overheads and therefore provide a fair comparison of both finalists in terms of running time.

6.2 Core Functions Implementations

ACORN Although ACORN is designed to process one bit per step, up to 32 steps can be processed in parallel. In this way, each function defined in Sect. 2 should be seen as operating on 32-bit words W_i instead of bits S_i. An optimized implementation that stores each LFSR into a 64-bit word, including the last 4 bits $S_{289}, ..., S_{292}$ has been proposed by the designer. As a result, the 293-bit internal state actually requires $7 \times 64 = 448$ bits in RAM. Although more compact implementations can be achieved, dedicating each LFSR to a register allow to save instructions in order to build 32-bit working variables, as simple bit masks can be applied on 64-bit words, resulting in a timememory trade-off. Nervertheless, operands W_i for $i \in \{12, , 24, 66, 111, 160, 196, 235, 244\}$ still require some bit manipulation to be built. We followed this approach for our implementation as it offers the best performance. Because our platform uses a 32-bit processor, the 293-bit internal state fits into thirteen 32-bit words instead of seven 64-bit ones, resulting in $13 \times 32 = 416$ bits. The assembly code is provided in Appendix A.

Ascon Because its state consists of five 64-bit registers and it can be implemented only using bitwise operations on 64-bit words, Ascon achieves very good performance on high-end CPUs. However, it remains efficient on 32-bit platforms since it consists in splitting 64-bit words into 32-bit ones and applying operations on each half. The only drawback comes from the bitwise rotation used in the linear layer, which is defined on 64-bit words and thus requires 4 instructions instead of 2 for the other operations. The 320-bit internal state is represented by ten 32-bit words. Our implementation of the transformation p is straightforward and follows the bitsliced approach as described in its specification. The assembly code is provided in Appendix B.

6.3 Practical Results

All results presented below were obtained using the STM32F100RBT6B development board which comes with 128KB of flash memory and 8KB of RAM, embedding an ARM Cortex-M3 core running at 24 MHz. The C code was compiled using the GNU ARM C compiler 5.06 (update 2) with the `-O2` optimization.

The two finalists are compared in terms of performances with and without masking. For our benchmark, we consider plaintexts from 8 to 256 bytes. We justify this choice by the fact that while some IoT networks only support short payloads (*e.g.* 12 bytes for Sigfox), other technologies allow to transfer larger packets (*e.g.* up to 243 bytes for LoRa [21]). All computations take as input an additional data of 16 bytes, in order to take this processing phase into account. Note that the results presented in Fig. 5 do not take the generation of random numbers into account, since our microcontroller does not support this feature. The code size in ROM of each implementation is reported in Fig. 6. It follows from Fig. 5 that without any protection against DPA, Ascon outperforms ACORN for short messages only. Regarding protected implementations, the overhead introduced by the masking countermeasure is constant and does not depend on the input length. Especially, we measure an extra running time of around 13 000 clock cycles for ACORN and 10 000 for Ascon. Therefore, the integration of masking does not fundamentally change the previous results and Ascon remains more efficient than ACORN for messages less than 128 bytes. As a result, high-order masked implementations should lead to pretty similar analyses, depending on randomness requirements.

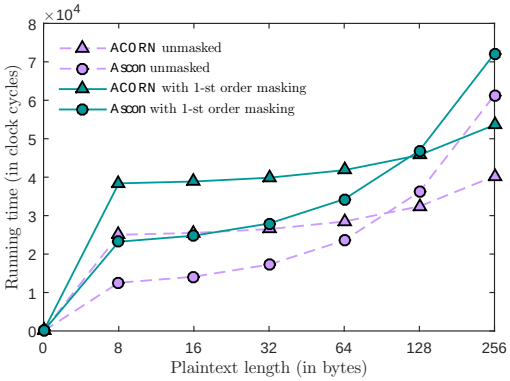

Fig. 5: Execution times comparison

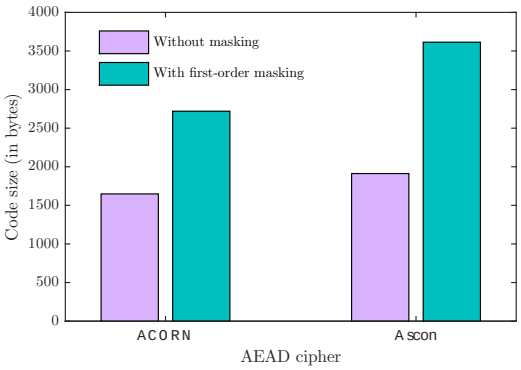

Fig. 6: Code sizes (encryption + decryption)

6.4 Randomness Complexity

Another aspect to consider regarding masked implementations is the randomness complexity, which does matter for ACORN and Ascon since they are intended to run on embedded platforms where the generation of random numbers might be troublesome. Because our first-order masked implementations rely on Alg. 3 that do not require additional random numbers to compute AND gates, the total amount of randomness that has to be generated equals the size of the state. Although the implementation of ACORN discussed in this paper uses additional memory in order to align several words to save some instructions, only the 293 effective bits have to be masked. Therefore, the randomness requirements for first-order masking are very similar for ACORN and Ascon, consisting of 293 and 320 random bits, respectively.

However, high-order masked implementations lead to complications since each AND gate requires additional randomness. Referring to 32-bit implementations, it would precisely require $(n^2 + n)/2$ 32-bit random numbers at each secure AND computation. Because such requirements can be impossible to achieve in practice, reducing the amount of randomness

for masking schemes is an active research field. Regarding secure AND calculaltions, it has been shown in [3] that the randomness complexity at order n can be reduced from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$. Another recent work investigates how to recycle the randomn numbers previously generated in order to re-use them for other operations [11]. Anyway, the fewer secure AND there will be to compute, the fewer random numbers there will be to generate.

In the case of ACORN, 10 AND computations are required for a state update. However because two of them are performed with the control bits *ca* and *cb* as other operands, either equal to `0x00000000` or `0xffffffff`, they can be easily computed without requiring secure AND calculations. As the initialization phase consists of 1792 steps, parallel 32-bit implementations require $1792/32 = 56$ state updates, resulting in a total $56 \times (10-2) = 448$ secure AND calculations for a masked initialization of ACORN. Regarding Ascon, there are only 5 AND calculations within the transformation p, which results in 10 operations for 32-bit implementations. As p is iterated for 12 rounds at both initialization and finalization phases, a total of $10 \times 12 \times 2 = 240$ secure AND calculations are required for a masked implementation of Ascon, according to our masking scheme defined in Sect. 5.4. Therefore, Ascon seems more appropriate to higher-order masking since it only requires almost half the number of AND to secure as ACORN.

7 Conclusion

In this paper we analysed the side-channel resistance of both CAESAR finalists for the lightweight applications profile, ACORN and Ascon. While numerous security evaluations have already been published regarding Ascon, we introduced the first DPA attack against ACORN, justifying the need of countermeasures at the implementation level. Our attack does not return the encryption key itself but a system of boolean equations to solve, introducing an All-SAT problem. Our practical investigations using the `CryptoMiniSat5` tool led to the conclusion that at least the first 82 updates with the IV as input have to be considered in order to get a system with only one solution. We also recalled the previous results regarding Ascon and underlined that the finalization should be protected as well as the initialization.

After recalling the principle of the masking countermeasure, we proposed a masking scheme for each finalist. It results from our study that only subparts of the algorithms can be protected against side-channel attacks in order to minimize the impact on performance. Subsequently, we justified our software implementation strategies on a microcontroller embedding an ARM Cortex-M3 CPU. Finally, we compared the clock cycles required by our implementations to encrypt and authenticate messages of different lengths. It results that, regardless of masking, Ascon outperforms ACORN for messages less than 128 bytes, while ACORN is significantly more efficient for larger inputs. This is mainly explained by the fact that ACORN bears the cost of its heavy initialization phase. Although our benchmark only considers unsecure and first-order masked implementations, we argued that similar results should be observed at higher-order as Ascon requires fewer secure AND computations than ACORN. On the other hand, ACORN offers the best results in terms of code size and allows to achieve more compact implementations.

References

1. Adomnicai, A., Fournier, J.J.A., Masson, L.: Bricklayer Attack: A Side-Channel Analysis on the ChaCha Quarter Round. In: Patra, A., Smart, N.P. (eds.) Progress in Cryptology – INDOCRYPT 2017. pp. 65–84. Springer International Publishing, Cham (2017)

2. Andreeva, E., Bogdanov, A., Datta, N., Luykx, A., Mennink, B., Nandi, M., Tischhauser, E., Yasuda, K.: COLM v1. Submission to the CAESAR competition: https://competitions.cr.yp.to/round3/colmv1.pdf (2016)
3. Belaïd, S., Benhamouda, F., Passelègue, A., Prouff, E., Thillard, A., Vergnaud, D.: Randomness Complexity of Private Circuits for Multiplication. In: Fischlin, M., Coron, J.S. (eds.) Advances in Cryptology – EUROCRYPT 2016. pp. 616–648. Springer Berlin Heidelberg, Berlin, Heidelberg (2016)
4. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: Permutation-based encryption, authentication and authenticated encryption. DIAC - Directions in Authenticated Ciphers (2012)
5. Biryukov, A., Dinu, D., Le Corre, Y., Udovenko, A.: Optimal First-Order Boolean Masking for Embedded IoT Devices. In: Eisenbarth, T., Teglia, Y. (eds.) Smart Card Research and Advanced Applications. pp. 22–41. Springer International Publishing, Cham (2018)
6. Böck, H., Zauner, A., Devlin, S., Somorovsky, J., Jovanovic, P.: Nonce-disrespecting Adversaries: Practical Forgery Attacks on GCM in TLS. In: Proceedings of the 10th USENIX Conference on Offensive Technologies. pp. 15–25. WOOT'16, USENIX Association, Berkeley, CA, USA (2016), http://dl.acm.org/citation.cfm?id=3027019.3027021
7. Coron, J.S., Großschädl, J., Tibouchi, M., Vadnala, P.K.: Conversion from Arithmetic to Boolean Masking with Logarithmic Complexity, pp. 130–149. Springer Berlin Heidelberg, Berlin, Heidelberg (2015), https://doi.org/10.1007/978-3-662-48116-5_7
8. Diehl, W., Abdulgadir, A., Farahmand, F., Kaps, J.P., Gaj, K.: Comparison of cost of protection against differential power analysis of selected authenticated ciphers. In: 2018 IEEE International Symposium on Hardware Oriented Security and Trust (HOST). pp. 147–152 (April 2018)
9. Dobraunig, C., Eichlseder, M., Mendel, F., Schläffer, M.: Ascon v1.2. Submission to the CAESAR competition: http://competitions.cr.yp.to/round3/asconv12.pdf (2016), http://ascon.iaik.tugraz.at
10. Dworkin, M.J.: SP 800-38D. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. Tech. rep., Gaithersburg, MD, United States (2007)
11. Faust, S., Paglialonga, C., Schneider, T.: Amortizing randomness complexity in private circuits. Cryptology ePrint Archive, Report 2017/869 (2017), https://eprint.iacr.org/2017/869
12. Gierlichs, B., Batina, L., Clavier, C., Eisenbarth, T., Gouget, A., Handschuh, H., Kasper, T., Lemke-Rust, K., Mangard, S., Moradi, A., Oswald, E.: Susceptibility of eSTREAM candidates towards side channel analysis pp. 123–150 (07 2018)
13. Gross, H., Wenger, E., Dobraunig, C., Ehrenhfer, C.: Ascon Hardware Implementations and Side-channel Evaluation. Microprocess. Microsyst. 52(C), 470–479 (Jul 2017), https://doi.org/10.1016/j.micpro.2016.10.006
14. Ishai, Y., Sahai, A., Wagner, D.: Private circuits: Securing hardware against probing attacks. In: Boneh, D. (ed.) Advances in Cryptology - CRYPTO 2003. pp. 463–481. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)
15. Jean, J., Nikolić, I., Peyrin, T., Seurin, Y.: Deoxys v1.41. Submission to the CAESAR competition: https://competitions.cr.yp.to/round3/deoxysv141.pdf (2016), http://www1.spms.ntu.edu.sg/~syllab/m/index.php/Deoxys
16. Junttila, T.A., Niemelä, I.: Towards an Efficient Tableau Method for Boolean Circuit Satisfiability Checking. In: Lloyd, J., Dahl, V., Furbach, U., Kerber, M., Lau, K.K., Palamidessi, C., Pereira, L.M., Sagiv, Y., Stuckey, P.J. (eds.) Computational Logic — CL 2000. pp. 553–567. Springer Berlin Heidelberg, Berlin, Heidelberg (2000)
17. Kocher, P.C., Jaffe, J., Jun, B.: Differential power analysis. In: Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology. pp. 388–397. CRYPTO '99, Springer-Verlag, London, UK, UK (1999), http://dl.acm.org/citation.cfm?id=646764.703989
18. Krovetz, T., Rogaway, P.: OCB (v1.1). Submission to the CAESAR competition: https://competitions.cr.yp.to/round3/ocbv11.pdf (2016)
19. Oren, Y., Kirschbaum, M., Popp, T., Wool, A.: Algebraic Side-Channel Analysis in the Presence of Errors. In: Mangard, S., Standaert, F.X. (eds.) Cryptographic Hardware and Embedded Systems, CHES 2010. pp. 428–442. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
20. Procter, G., Cid, C.: On Weak Keys and Forgery Attacks Against Polynomial-Based MAC Schemes. Journal of Cryptology 28, 769–795 (2013)
21. Raza, U., Kulkarni, P., Sooriyabandara, M.: Low Power Wide Area Networks: A Survey. CoRR abs/1606.07360 (2016), http://arxiv.org/abs/1606.07360
22. Rogaway, P., Bellare, M., Black, J., Krovetz, T.: OCB: A Block-cipher Mode of Operation for Efficient Authenticated Encryption. In: Proceedings of the 8th ACM Conference on Computer and Communications Security. pp. 196–205. CCS '01, ACM, New York, NY, USA (2001), http://doi.acm.org/10.1145/501983.502011

23. Samwel, N., Daemen, J.: DPA on Hardware Implementations of Ascon and Keyak. In: Proceedings of the Computing Frontiers Conference. pp. 415–424. CF'17, ACM, New York, NY, USA (2017), http://doi.acm.org/10.1145/3075564.3079067
24. Siddhanti, A., Sarkar, S., Maitra, S., Chattopadhyay, A.: Differential Fault Attack on Grain v1, ACORN v3 and Lizard. In: SPACE (2017)
25. Soos, M., Nohl, K., Castelluccia, C.: Extending SAT Solvers to Cryptographic Problems. In: Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings. pp. 244–257 (2009), https://doi.org/10.1007/978-3-642-02777-2_24
26. Strobel, D.: Side Channel Analysis Attacks on Stream Ciphers. Master's thesis, Ruhr-Universitt Bochum (2010), https://www.emsec.rub.de/media/crypto/attachments/files/2010/04/mt_strobel.pdf
27. Trichina, E.: Combinational Logic Design for AES SubByte Transformation on Masked Data. Cryptology ePrint Archive, Report 2003/236 (2003), https://eprint.iacr.org/2003/236
28. Tunstall, M., Hanley, N., McEvoy, R., Whelan, C., Murphy, C., Marnane, W.: Correlation Power Analysis of Large Word Sizes (2007), http://www.geocities.ws/mike.tunstall/papers/THMWMM.pdf
29. Whiting, D., Housley, R., Ferguson, N.: Counter with CBC-MAC (CCM). RFC 3610, RFC Editor (September 2003), http://www.rfc-editor.org/rfc/rfc3610.txt, http://www.rfc-editor.org/rfc/rfc3610.txt
30. Wu, H.: ACORN: A Lightweight Authenticated Cipher (v3). Submission to the CAESAR competition: https://competitions.cr.yp.to/round3/acornv3.pdf (2016)
31. Wu, H., Huang, T.: The Authenticated Cipher MORUS (v2). Submission to the CAESAR competition: https://competitions.cr.yp.to/round3/morusv2.pdf (2016)
32. Wu, H., Preneel, B.: AEGIS: A Fast Authenticated Encryption Algorithm (v1.1). Submission to the CAESAR competition: https://competitions.cr.yp.to/round3/aegisv11.pdf (2016)
33. Zhang, X., Feng, X., Lin, D.: Fault Attack on ACORN v3. The Computer Journal p. bxy044 (2018), http://dx.doi.org/10.1093/comjnl/bxy044
34. Zhao, X., Zhang, F., Guo, S., Wang, T., Shi, Z., Liu, H., Ji, K.: MDASCA: An Enhanced Algebraic Side-Channel Attack for Error Tolerance and New Leakage Model Exploitation. In: Schindler, W., Huss, S.A. (eds.) Constructive Side-Channel Analysis and Secure Design. pp. 231–248. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)

A ARM Assembly Code for the **ACORN** State Update

```
;state[0]:  S_31...S_0          (32 bits)
;state[1]:  S_60...S_32         (29 bits)
;state[2]:  S_92...S_61         (32 bits)
;state[3]:  S_106...S_93        (14 bits)
;state[4]:  S_138...S_107       (32 bits)
;state[5]:  S_153...S_139       (15 bits)
;state[6]:  S_185...S_154       (32 bits)
;state[7]:  S_192...S_186       (7  bits)
;state[8]:  S_224...S_193       (32 bits)
;state[9]:  S_229...S_225       (5  bits)
;state[10]: S_261...S_230       (32 bits)
;state[11]: S_288...S_262       (27 bits)
;state[12]: S_292...S_289       (4  bits)
acorn_asm
        ; r0 points to the internal state
        PUSH    {r4-r11, lr}
        SUB     sp, sp, #0x1c
        ; --- computes temp data ---
        ; computes S_275...S_244
        LDRD    r4, r5, [r0,#0x28]
        LSR     r4, r4, #14
        ORR     r4, r4, r5, LSL #18
        STR     r4, [sp, #0x18]
        ; computes S_266...S_235
        LDR     r4, [r0, #0x28]
        LSR     r4, r4, #5
        ORR     r4, r4, r5, LSL #27
        STR     r4, [sp, #0x14]
        ; computes S_227...S_196
        LDRD    r4, r5, [r0, #0x20]
        LSR     r4, r4, #3
        ORR     r4, r4, r5, LSL #29
        STR     r4, [sp, #0x10]
        ; computes S_191...S_160
        LDRD    r4, r5, [r0, #0x18]
        LSR     r4, r4, #6
        ORR     r4, r4, r5, LSL #26
        STR     r4, [sp, #0x0c]
        ; computes S_142...S_111
        LDRD    r4, r5, [r0, #0x10]
        LSR     r4, r4, #4
        ORR     r4, r4, r5, LSL #28
        STR     r4, [sp, #0x08]
        ; computes S_97...S_66
        LDRD    r4, r5, [r0, #0x08]
        LSR     r4, r4, #5
        ORR     r4, r4, r5, LSL #27
        STR     r4, [sp, #0x04]
        ; computes S_54...S_23
        LDRD    r4, r5, [r0]
        LSR     r4, r4, #23
        ORR     r4, r4, r5, LSL #9
        STR     r4, [sp]

        ; --- updates LFSRs ---
        ; updates LFSR 5
        LDR     r4, [r0, #0x28]
        LDR     r5, [r0, #0x30]
        LDR     r6, [sp, #0x14]
        EOR     r6, r4, r6
        EOR     r5, r5, r6
        STR     r5, [r0, #0x30]
        ; updates LFSR 4
        LDR     r5, [r0, #0x20]
        LDR     r6, [sp, #0x10]
        EOR     r6, r5, r6
        EOR     r4, r4, r6
        STR     r4, [r0, #0x28]
        ; updates LFSR 3
        LDR     r4, [r0, #0x18]
        LDR     r6, [sp, #0x0c]
        EOR     r6, r4, r6
        EOR     r5, r5, r6
        STR     r5, [r0, #0x20]
        ; updates LFSR 2
        LDR     r5, [r0, #0x10]
        LDR     r6, [sp, #0x08]
        EOR     r6, r5, r6
        EOR     r4, r4, r6
        STR     r4, [r0, #0x18]
        ; updates LFSR 1
        LDR     r4, [r0, #0x08]
        LDR     r6, [sp, #0x04]
        EOR     r6, r4, r6
        EOR     r5, r5, r6
        STR     r5, [r0, #0x10]
        ; updates LFSR 0
        LDR     r5, [r0]
        LDR     r6, [sp]
        EOR     r6, r5, r6
        EOR     r4, r4, r6
        STR     r4, [r0, #0x08]

        ; --- generates the keystream ---
        ; computes maj(W_235, W_61, W_193)
        LDR     r5, [r0, #0x20]
        LDR     r6, [sp, #0x14]
        AND     r7, r4, r5
        AND     r4, r4, r6
        AND     r5, r5, r6
        EOR     r4, r4, r5
        EOR     r4, r4, r7
        ; computes ch(W_230, W_111, W_66)
        LDRD    r5, r6, [sp, #0x04]
        LDR     r7, [r0, #0x28]
        AND     r6, r6, r7
        MVN     r7, r7
        AND     r7, r5, r7
        EOR     r6, r6, r7
        ; finalizes keystream computation
        EOR     r4, r4, r6
        LDR     r5, [r0, #0x18] ; W_154
        EOR     r4, r4, r5
        ; computes S_43...S_12
        LDRD    r5, r6, [r0]
        LSR     r5, r5, #12
        ORR     r5, r5, r6, LSL #20
        EOR     r4, r4, r5

        ; --- computes the ciphertext ---
        EOR     r5, r1, r4
        STR     r5, [r2]

        ; --- generates the feedback word ---
        ; computes (c_a & W_196) ^ (c_b & ks)
        LDRD    r5, r6, [r3]
        AND     r4, r4, r6
        LDR     r6, [sp, #0x10]
        AND     r5, r5, r6
        EOR     r4, r4, r5
        EOR     r1, r1, r4
        ; computes maj(W_244, W_23, W_160)
        LDR     r4, [sp, #0x18]
        LDR     r5, [sp]
        LDR     r6, [sp, #0x0c]
        AND     r7, r4, r5
        AND     r4, r4, r6
        AND     r5, r5, r6
        EOR     r4, r4, r5
        EOR     r4, r4, r7
        EOR     r1, r1, r4
        ; computes state[0] ^ (~state[2])
        LDR     r4, [r0, #0x10]
        MVN     r4, r4
        EOR     r1, r1, r4
        LDR     r4, [r0]
        EOR     r1, r1, r4

        ; --- state[6] ^= feedback << 4 ---
        LDR     r4, [r0, #0x30]
        EOR     r4, r4, r1, LSL #4
        STR     r4, [r0, #0x30]

        ; --- shifts all LFSRs by 32 bits ---
        ; 1st word
        LDRD    r4, r5, [r0, #0x04]
        ORR     r4, r4, r5, LSL #29
        STR     r4, [r0]
        ; 2nd word
        LSR     r5, r5, #3
        STR     r5, [r0, #0x04]
        ; 3rd word
        LDRD    r4, r5, [r0, #0x0c]
        ORR     r4, r4, r5, LSL #14
        STR     r4, [r0, #0x08]
        ; 4th word
        LSR     r5, r5, #18
        STR     r5, [r0, #0x0c]
        ; 5th word
        LDRD    r4, r5, [r0, #0x14]
        ORR     r4, r4, r5, LSL #15
        STR     r4, [r0, #0x10]
        ; 6th word
        LSR     r5, r5, #17
        STR     r5, [r0, #0x14]
        ; 7th word
        LDRD    r4, r5, [r0, #0x1c]
        ORR     r4, r4, r5, LSL #7
        STR     r4, [r0, #0x18]
        ; 8th word
        LSR     r5, r5, #25
        STR     r5, [r0, #0x1c]
        ; 9th word
        LDRD    r4, r5, [r0, #0x24]
        ORR     r4, r4, r5, LSL #5
        STR     r4, [r0, #0x20]
        ; 10th word
        LSR     r5, r5, #27
        STR     r5, [r0, #0x24]
        ; 11th word
        LDRD    r4, r5, [r0, #0x2c]
        ORR     r4, r4, r5, LSL #27
        STR     r4, [r0, #0x28]
        ; 12th word
        LSR     r5, r5, #5
        STR     r5, [r0, #0x2c]
        ; 13th word
        LSR     r4, r1, #28
        STR     r4, [r0, #0x30]

        ADD     sp, sp, #0x1c
        POP     {r4-r11, lr}
        BX      lr
```

B ARM Assembly Code for the transformation p of **Ascon**

```
permutation_ascon
        ; r0 points to the internal state
        PUSH    {r2-r11, lr}
        SUB             sp, sp, #0x08

        ; for each round
loop
        ; --- loads the entire state ---
        LDRD    r2, r3, [r0]
        LDRD    r4, r5, [r0, #0x08]
        LDRD    r6, r7, [r0, #0x10]
        LDRD    r8, r9, [r0, #0x18]
        LDRD    r10, r11, [r0, #0x20]

        ; --- adds round constant ---
        EOR     r7, r7, r1

        ; --- applies the nonlinear layer ---
        EOR     r2, r2, r10     ; x0 ^= x4
        EOR     r3, r3, r11     ; x0 ^= x4
        STRD    r2, r3, [r0]
        EOR     r10, r10, r8    ; x4 ^= x3
        EOR     r11, r11, r9    ; x4 ^= x3
        STRD    r10, r11, [r0, #0x20]
        EOR     r6, r6, r4      ; x2 ^= x1
        EOR     r7, r7, r5      ; x2 ^= x1
        STRD    r6, r7, [r0, #0x10]
        MVN     r2, r2  ; ~x0
        MVN     r3, r3  ; ~x0
        AND     r2, r2, r4      ; ~x0 & x1
        AND     r3, r3, r5      ; ~x0 & x1
        STRD    r2, r3, [sp]
        MVN     r4, r4  ; ~x1
        MVN     r5, r5  ; ~x1
        AND     r4, r4, r6      ; ~x1 & x2
        AND     r5, r5, r7      ; ~x1 & x2
        MVN     r6, r6  ; ~x2
        MVN     r7, r7  ; ~x2
        AND     r6, r6, r8      ; ~x2 & x3
        AND     r7, r7, r9      ; ~x2 & x3
        MVN     r8, r8  ; ~x3
        MVN     r9, r9  ; ~x3
        AND     r8, r8, r10     ; ~x3 & x4
        AND     r9, r9, r11     ; ~x3 & x4
        MVN     r10, r10        ; ~x4
        MVN     r11, r11        ; ~x4
        LDRD    r2, r3, [r0]
        AND     r10, r10, r2    ; ~x4 & x0
        AND     r11, r11, r3    ; ~x4 & x0
        EOR     r2, r2, r4      ; x0 ^= ~x1 & x2
        EOR     r3, r3, r5      ; x0 ^= ~x1 & x2
        STRD    r2, r3, [r0]
        LDRD    r2, r3, [r0, #0x08]
        EOR     r2, r2, r6      ; x1 ^= ~x2 & x3
        EOR     r3, r3, r7      ; x1 ^= ~x2 & x3
        STRD    r2, r3, [r0, #0x08]
        LDRD    r2, r3, [r0, #0x10]
        EOR     r2, r2, r8      ; x2 ^= ~x3 & x4
        EOR     r3, r3, r9      ; x2 ^= ~x3 & x4
        STRD    r2, r3, [r0, #0x10]
        LDRD    r2, r3, [r0, #0x18]
        EOR     r2, r2, r10     ; x3 ^= ~x4 & x0
        EOR     r3, r3, r11     ; x3 ^= ~x4 & x0
        STRD    r2, r3, [r0, #0x18]
        LDRD    r10, r11, [r0, #0x20]
        LDRD    r2, r3, [sp]
        EOR     r10, r10, r2    ; x4 ^= ~x0 & x1
        EOR     r11, r11, r3    ; x4 ^= ~x0 & x1
        LDRD    r2, r3, [r0]
        LDRD    r4, r5, [r0, #0x08]
        LDRD    r6, r7, [r0, #0x10]
        LDRD    r8, r9, [r0, #0x18]
        EOR     r4, r4, r2      ; x1 ^= x0
        EOR     r5, r5, r3      ; x1 ^= x0
        EOR     r2, r2, r10     ; x0 ^= x4
        EOR     r3, r3, r11     ; x0 ^= x4
        EOR     r8, r8, r6      ; x3 ^= x2
        EOR     r9, r9, r7      ; x3 ^= x2
        MVN     r6, r6          ; ~x2
        MVN     r7, r7          ; ~x2
        STRD    r8, r9, [r0, #0x18]
        STRD    r10, r11, [r0, #0x20]

        ; --- applies the linear layer ---
        ; linear diffusion on x0
        LSR     r10, r2, #19
        ORR     r10, r10, r3, LSL #13
        LSR     r11, r3, #19
        ORR     r11, r11, r2, LSL #13
        LSR     r8, r2, #28
        ORR     r8, r8, r3, LSL #4
        LSR     r9, r3, #28
        ORR     r9, r9, r2, LSL #4
        EOR     r2, r2, r10
        EOR     r3, r3, r11
        EOR     r2, r2, r8
        EOR     r3, r3, r9
        STRD    r2, r3, [r0]
        ; linear diffusion on x1
        LSR     r10, r5, #29
        ORR     r10, r10, r4, LSL #3
        LSR     r11, r4, #29
        ORR     r11, r11, r5, LSL #3
        LSR     r8, r5, #7
        ORR     r8, r8, r4, LSL #25
        LSR     r9, r4, #7
        ORR     r9, r9, r5, LSL #25
        EOR     r4, r4, r10
        EOR     r5, r5, r11
        EOR     r4, r4, r8
        EOR     r5, r5, r9
        STRD    r4, r5, [r0, #0x08]
        ; linear diffusion on x2
        LSR     r10, r6, #1
        ORR     r10, r10, r7, LSL #31
        LSR     r11, r7, #1
        ORR     r11, r11, r6, LSL #31
        LSR     r8, r6, #6
        ORR     r8, r8, r7, LSL #26
        LSR     r9, r7, #6
        ORR     r9, r9, r6, LSL #26
        EOR     r6, r6, r10
        EOR     r7, r7, r11
        EOR     r6, r6, r8
        EOR     r7, r7, r9
        STRD    r6, r7, [r0, #0x10]
        ; linear diffusion on x3
        LDRD    r6, r7, [r0, #0x18]
        LSR     r10, r6, #10
        ORR     r10, r10, r7, LSL #22
        LSR     r11, r7, #10
        ORR     r11, r11, r6, LSL #22
        LSR     r8, r6, #17
        ORR     r8, r8, r7, LSL #15
        LSR     r9, r7, #17
        ORR     r9, r9, r6, LSL #15
        EOR     r6, r6, r10
        EOR     r7, r7, r11
        EOR     r6, r6, r8
        EOR     r7, r7, r9
        STRD    r6, r7, [r0, #0x18]
        ; linear diffusion on x4
        LDRD    r6, r7, [r0, #0x20]
        LSR     r10, r6, #7
        ORR     r10, r10, r7, LSL #25
        LSR     r11, r7, #7
        ORR     r11, r11, r6, LSL #25
        LSR     r8, r7, #9
        ORR     r8, r8, r6, LSL #23
        LSR     r9, r6, #9
        ORR     r9, r9, r7, LSL #23
        EOR     r6, r6, r10
        EOR     r7, r7, r11
        EOR     r6, r6, r8
        EOR     r7, r7, r9
        STRD    r6, r7, [r0, #0x20]

        ; --- loop iteration check ---
        SUB     r1, r1, #0x0f
        CMP     r1, #0x4b
        BGE     loop

        ADD     sp, sp, #0x08
        POP     {r2-r11, lr}
        BX      lr
```