Cryptanalysis of ARMADILLO2

Mohamed Ahmed Abdelraheem, Céline Blondeau, Maria Naya-Plasencia, Marion Videau, Erik Zenner

Cryptanalysis of ARMADILLO2 *

Mohamed Ahmed Abdelraheem[1], Céline Blondeau[2], María Naya-Plasencia[3] †, Marion Videau[4,5] ‡, and Erik Zenner[6] §

[1] Technical University of Denmark, Department of Mathematics, Denmark
[2] INRIA, project-team SECRET, France
[3] FHNW, Windisch, Switzerland and University of Versailles, France
[4] Agence nationale de la scurit des systmes d'information, France
[5] Université Henri Poincaré-Nancy 1 / LORIA, France
[6] University of Applied Sciences Offenburg, Germany

Abstract. ARMADILLO2 is the recommended variant of a multi-purpose cryptographic primitive dedicated to hardware which has been proposed by Badel et al. in [1]. In this paper, we describe a meet-in-the-middle technique relying on the parallel matching algorithm that allows us to invert the ARMADILLO2 function. This makes it possible to perform a key recovery attack when used as a FIL-MAC. A variant of this attack can also be applied to the stream cipher derived from the PRNG mode. Finally we propose a (second) preimage attack when used as a hash function. We have validated our attacks by implementing cryptanalysis on scaled variants. The experimental results match the theoretical complexities.
In addition to these attacks, we present a generalization of the parallel matching algorithm, which can be applied in a broader context than attacking ARMADILLO2.

Keywords: ARMADILLO2, meet-in-the-middle, key recovery attack, preimage attack, parallel matching algorithm.

1 Introduction

ARMADILLO is a multi-purpose cryptographic primitive dedicated to hardware which was proposed by Badel et al. in [1]. Two variants were presented: ARMADILLO and ARMADILLO2, the latter being the recommended version. In the following, the first variant will be denoted ARMADILLO1 to distinguish it from ARMADILLO2. Both variants comprise several versions, each

*This work was partially supported by the European Commission through the ICT programme under contract ICT-2007-216676 ECRYPT II

†Supported by the National Competence Center in Research on Mobile Information and Communication Systems (NCCR-MICS), a center of the Swiss National Science Foundation under grant number 5005-67322 and by the French Agence Nationale de la Recherche through the SAPHIR2 project under Contract ANR-08-VERS-014

‡Partially supported by the French Agence Nationale de la Recherche under Contract ANR-06-SETI-013-RAPIDE

§This work was produced while at the Technical University of Denmark

one associated to a different set of parameters and to a different security level. For both primitives, several applications are proposed: fixed input-length MAC (FIL-MAC), pseudo-random number generator/pseudo-random function (PRNG/PRF), and hash function. In [6], authors present a polynomial attack on ARMADILLO1. Even if the design of ARMADILLO2 is similar to the design of the first version, authors of [6] claim that this attack can not be applied on ARMADILLO2.

The ARMADILLO family uses a parameterized internal permutation as a building block. This internal permutation is based on two bitwise permutations σ_0 and σ_1. In [1], these permutations are not specified, but some of the properties that they must satisfy are given.

In this paper we provide the first cryptanalysis of ARMADILLO2, the recommended variant. As the bitwise permutations σ_0 and σ_1 are not specified, we have performed our analysis under the reasonable assumption that they behave like random permutations. As a consequence, the results of this paper are independent of the choice for σ_0 and σ_1.

To perform our attack, we use a meet-in-the-middle approach and an evolved variant of the parallel matching algorithm introduced in [2] and generalized in [5, 4]. Our method enables us to invert the building block of ARMADILLO2 for a chosen value of the public part of the input, when a part of the output is known. We can use this step to build key recovery attacks faster than exhaustive search on all versions of ARMADILLO2 used in the FIL-MAC application mode. Besides, we propose several trade-offs for the time and memory needed for these attacks. We also adapt the attack to recover the key when ARMADILLO2 is used as a stream cipher in the PRNG application mode. We further show how to build (second) preimage attacks faster than exhaustive search when using the hashing mode, and propose again several time-memory trade-offs. We have implemented the attacks on a scaled version of ARMADILLO2, and the experimental results confirm the theoretical predictions.

Organization of the paper. We briefly describe ARMADILLO2 in Section 2. In Section 3 we detail our technique for inverting its building block and we explain how to extend the parallel matching algorithm to the case of ARMADILLO2. In Section 4, we explain how to apply this technique to build a key recovery attack on the FIL-MAC application mode. We briefly show how to adapt this attack to the stream cipher scenario in Section 4.2. The (second) preimage attack on the hashing mode is presented in Section 5. In Section 6 we present the experimental results of the verification that we have done on a scaled version of the algorithm. Finally, in Section 7, we propose a general form of the parallel matching algorithm derived from our attacks which can hopefully be used in more general contexts.

2 Description of ARMADILLO2

The core of ARMADILLO is based on the so-called *data-dependent bit transpositions* [3]. We recall the description of ARMADILLO2 given in [1] using the same notations.

2.1 Description

Let C be an initial vector of size c and U be a message block of size m. The size of the register $(C\|U)$ is $k = c + m$. The ARMADILLO2 function transforms the vector

(C, U) into (V_c, V_t) as described in Figure 1:

$$\begin{array}{rcl} \text{ARMADILLO2} & : & \mathbb{F}_2^c \times \mathbb{F}_2^m \to \mathbb{F}_2^c \times \mathbb{F}_2^m \\ & & (C, U) \quad \mapsto (V_c, V_t) = \text{ARMADILLO2}(C, U). \end{array}$$

The function ARMADILLO2 relies on an internal bitwise parameterized permutation denoted by Q which is defined by a parameter A of size a and is applied to a vector B of size k:

$$\begin{array}{rcl} Q & : & \mathbb{F}_2^a \times \mathbb{F}_2^k \to \mathbb{F}_2^k \\ & & (A, B) \quad \mapsto Q(A, B) = Q_A(B) \end{array}$$

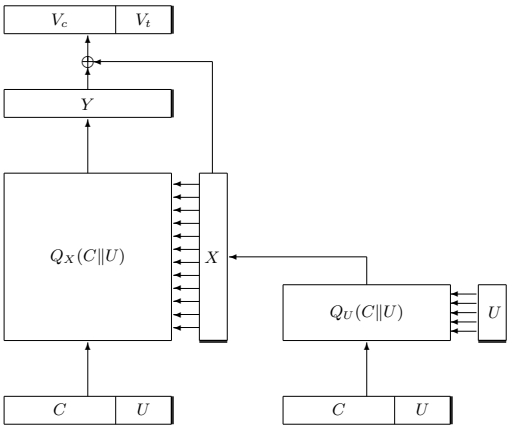

Fig. 1. ARMADILLO2.

Let σ_0 and σ_1 be two fixed bitwise permutations of size k. In [1], the permutations are not defined but some criteria they should fulfil are given. As the attacks presented in this paper are valid for any bitwise permutations, we do not describe these properties. We just stress that in the following, when computing the complexities we assume that these permutations behave like random ones. We denote by γ a constant of size k defined by alternating 0's and 1's: $\gamma = 1010 \cdots 10$.
Using these notations, we can define Q which is used twice in the ARMADILLO2 function. Let A be a parameter and B be the internal state, the parameterized permutation Q (that we denote by Q_A when indicating the parameter is necessary) consists in $a = |A|$ simple steps. The i-th step of Q (reading A from its least significant bit to its most significant one) is defined by:

- an elementary bitwise permutation: $B \leftarrow \sigma_{A_i}(B)$, that is:
  - if the i-th bit of A equals 0 we apply σ_0 to the current state,
  - otherwise (if the i-th bit of A equals 1) we apply σ_1 to the current state,
- a constant addition (bitwise XOR) of γ: $B \leftarrow B \oplus \gamma$.

Using the definition of the permutation Q, we can describe the function ARMADILLO2. Let (C, U) be the input, then ARMADILLO2(C, U) is defined by:

- first compute $X \leftarrow Q_U(C\|U)$
- then compute $Y \leftarrow Q_X(C\|U)$
- finally compute $(V_c\|V_t) \leftarrow Y \oplus X$, the output is (V_c, V_t).

Actually c and m can take different values depending on the required security level. A summary of the sets of parameters for the different versions (A, B, C, D or E) proposed in [1] is given in Table 1.

Version	k	c	m
A	128	80	48
B	192	128	64
C	240	160	80
D	288	192	96
E	384	256	128

Table 1. Sets of parameters for the different versions of ARMADILLO2.

2.2 A Multi-Purpose Cryptographic Primitive

The general-purpose cryptographic function ARMADILLO2 can be used for three types of applications: FIL-MAC, hashing, and PRNG/PRF.

ARMADILLO*2 in FIL-MAC mode.* The secret key is C and the challenge, considered known by the attacker, is U. The response is V_t.

ARMADILLO*2 in hashing mode.* It uses a strengthened Merkle-Damgård construction, where V_c is the chaining value or the hash digest, and U is the message block.

ARMADILLO*2 in PRNG/PRF mode.* The output sequence is obtained by taking the first t bits of (V_c, V_t) after at least r iterations. For ARMADILLO2 the proposed values are $r = 1$ and $t = k$ (see [1, Sec. 6]). When used as a stream cipher, the secret key is C. The keystream is composed of k-bit frames indexed by U which is a public value.

3 Inverting the ARMADILLO2 Function

In [1] a sketch of a meet-in-the-middle (MITM) attack on ARMADILLO1, the first variant of the primitive, is given by the authors to prove *lower bounds* for the complexity and justify the choice of parameters. However, they do not develop further their analysis.

In this section we describe how to invert the ARMADILLO2 function when a part of the output (V_c, V_t) is known and U is chosen in the input $(C\|U)$. Inverting means that we recover C. The method we present can be performed for any arbitrary bitwise permutations σ_0 and σ_1. To conduct our analysis we suppose that they behave like random ones. Indeed, if the permutations σ_0 and σ_1 were not behaving like random ones, one could exploit their distributions to reduce the complexities of the attacks presented in this paper. Therefore, we are considering the worst case scenario for an attacker.

First, we describe the meet-in-the-middle technique we use. It provides two lists of partial states in the middle of the main permutation Q_X. To determine a list of possible values for C, we need to select a subset of the cartesian product of these two lists containing consistent couples of partial states. To build such a subset efficiently, we explain how to use an adaptation of the *parallel matching algorithm* presented in [2, 5]. Then we present and apply the adapted algorithm and compute its time and memory complexities.
All cryptanalysis, we present, on the different applications of ARMADILLO2 relies on the technique for recovering C presented in this section.

3.1 The Meet-in-the-Middle Technique

Whatever mode ARMADILLO2 is embedded in, we use the following facts:

- We can choose the m-bit vector U, in the input vector $(C\|U)$.
- We know part of the output vector $(V_c\|V_t)$: the m-bit vector V_t in the FIL-MAC, the $(c+m)$-bit vector $(V_c\|V_t)$ in the PRNG/PRF and the c-bit vector V_c in the hash function.

We deal with two permutations: the pre-processing Q_U which is known as U is known and the main permutation Q_X which is unknown, and we exploit the three following equations:

- The permutation Q_U used in the pre-processing $X = Q_U(C\|U)$ is known. This implies that all the known bits in the input of the permutation can be traced to their corresponding positions in X. For instance, there are m coordinates of X whose values are determined by choosing U.
- The output of the main permutation $Y = (V_c\|V_t) \oplus X$ implies we know some bits of Y. The amount of known bits of Y is denoted by y and is depending on the mode we are focusing on through $(V_c\|V_t)$.
- In the sequel, we divide X in two parts: $X = (X_{\text{out}}\|X_{\text{in}})$. Then, the main permutation $Y = Q_X(C\|U)$ can be divided in two parts: $Q_{X_{\text{in}}}$ and $Q_{X_{\text{out}}}$ separated by a division line we call the *middle*, hence we perform the meet-in-the-middle technique between $Q_{X_{\text{in}}}$ and $Q^{-1}_{X_{\text{out}}}$.

As $(X_{\text{out}}\|X_{\text{in}}) = Q_U(C\|U)$, we denote by m_{in} (resp. m_{out}) the number of bits of U that are in X_{in} (resp. X_{out}). We have $m_{\text{out}} + m_{\text{in}} = m$. We denote by ℓ_{in} (resp. ℓ_{out}) the number of bits coming from C in X_{in} (resp. X_{out}). We have $\ell_{\text{out}} + \ell_{\text{in}} = c$. The meet-in-the-middle attack is done by guessing the ℓ_{in} unknown bits of X_{in} and the ℓ_{out} unknown bits of X_{out} independently.

First, consider the forward direction. We can trace the ℓ_{in} unknown bits of X_{in} back to C with Q_U^{-1}. Next, for each possible guess of X_{in}, we can trace the corresponding ℓ_{in} bits from C plus the m bits from U to their positions in the middle by computing $Q_{X_{\text{in}}}(C\|U)$. Then consider the backward direction, we can trace the y known bits of Y back to the middle for each possible guess of X_{out}, that is computing $Q^{-1}_{X_{\text{out}}}(Y)$. This way we can obtain two lists $\mathcal{L}_{\text{in}}$ and $\mathcal{L}_{\text{out}}$, of size $2^{\ell_{\text{in}}}$ and $2^{\ell_{\text{out}}}$ respectively, of elements that represent partially known states in the middle of Q_X.

To describe our meet-in-the-middle attack we represent the partial states in the middle of Q_X as ternary vectors with coordinate values from $\{0, 1, -\}$, where $-$ denotes a coordinate (or cell) whose value is unknown. We say that a cell is *active* if it contains 0 or 1 and *inactive* otherwise. The weight of a vector V, denoted by $\text{wt}(V)$, is the number of its active cells. Two partial states are a match if their colliding active cells have the same values.

The list $\mathcal{L}_{\text{in}}$ contains elements $Q_{X_{\text{in}}}(C\|U)$ whose weight is $x = \ell_{\text{in}} + m$. The list $\mathcal{L}_{\text{out}}$ contains elements $Q^{-1}_{X_{\text{out}}}(Y)$ whose weight is y. When taking one element from each list, the probability of finding a match will then depend on the number of collisions of active cells between these two elements.

Consider a vector A in $\{0, 1, -\}^k$ with weight a. We denote by $P_{[k,a,b]}(i)$ the probability over all the vectors $B \in \{0, 1, -\}^k$ with weight b of having i active cells at the same positions in A and B. This event corresponds to the situation where there are i active cells of B among the a active positions in A and the remaining $(b - i)$ active

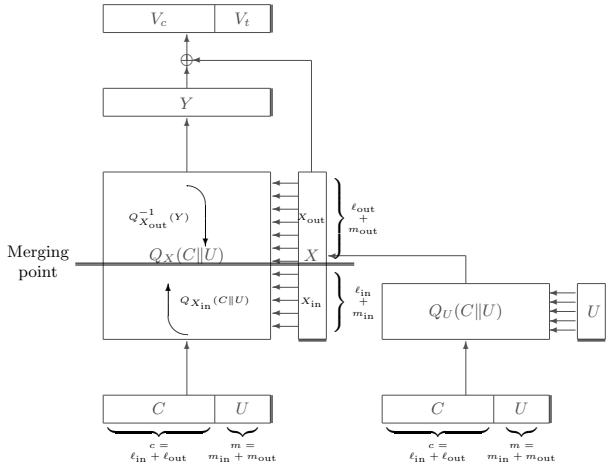

Fig. 2. Overview of the inversion of the ARMADILLO2 core function.

cells of B lie in the $(k-a)$ inactive positions in A. As the number of vectors of length k and weight b is $\binom{k}{b}$, we have:

$$P_{[k,a,b]}(i) = \frac{\binom{a}{i}\binom{k-a}{b-i}}{\binom{k}{b}} = \frac{\binom{b}{i}\binom{k-b}{a-i}}{\binom{k}{a}}.$$

Taking into account the probability of having active cells at the same positions in a pair of elements from $(\mathcal{L}_{\text{in}}, \mathcal{L}_{\text{out}})$ and the probability that these active cells do have the same value, we can compute the *expected probability* of finding a match for a pair of elements, that we will denote $2^{-N_{\text{coll}}}$. We have:

$$2^{-N_{\text{coll}}} = \sum_{i=0}^{y} 2^{-i} P_{[k,x,y]}(i).$$

This means that there will be a possible match with a probability of $2^{-N_{\text{coll}}}$. In total we will find $2^{\ell_{\text{in}}+\ell_{\text{out}}-N_{\text{coll}}}$ pairs of elements that pass this test. Each pair of elements defines a whole C value. Next, we just have to check which of these values is the correct one.

The big question now is that of the cost of checking which elements of the two lists $\mathcal{L}_{\text{in}}$ and $\mathcal{L}_{\text{out}}$ pass the test. The ternary alphabet of the elements and the changing positions of the active cells make it impossible to apply the approach of traditional MITM attacks — having an ordered list $\mathcal{L}_{\text{in}}$ and checking for each element in the list $\mathcal{L}_{\text{out}}$ if a match exists with cost 1 per element. Even more, a priori, for each element in $\mathcal{L}_{\text{in}}$ we would have to try if it matches each of the elements from $\mathcal{L}_{\text{out}}$ independently, which would yield the complexity of exhaustive search.

For solving this problem we adapt the algorithm described in [5, Sec. 2.3] as *parallel matching* to the case of ARMADILLO2. A generalized version of the algorithm is exposed in Section 7 with detailed complexity calculations and the link to our application case.

3.2 ARMADILLO2 Matching Problem: Matching Non-Random Elements

Recently, new algorithms have been proposed in [5] to solve the problem of *merging* several lists of big sizes with respect to a given relation t that can be verified by tuples

of elements. These new algorithms take advantage of the special structures that can be exhibited by t to reduce the complexity of solving this problem. As stated in [5], the problem of merging several lists can be reduced to the problem of merging two lists. Hereafter, we recall the reduced **Problem 1** proposed in [5] that we are interested in.

Problem 1 ([5]). *Let L_1 and L_2 be 2 lists of binary vectors of size 2^{ℓ_1} and 2^{ℓ_2} respectively. We denote by $\boldsymbol{x}$ a vector of L_1 and by $\boldsymbol{y}$ a vector of L_2.*

We assume that vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ can be decomposed into z groups of s bits, i.e. $\boldsymbol{x}, \boldsymbol{y} \in (\{0,1\}^s)^z$ and $\boldsymbol{x} = (x_1, \ldots, x_z)$ (resp. $\boldsymbol{y} = (y_1, \ldots, y_z)$). The vectors in L_1 and L_2 are drawn uniformly and independently at random from $\{0,1\}^{sz}$.

Let t be a Boolean function, $t : \{0,1\}^{sz} \times \{0,1\}^{sz} \to \{0,1\}$ such that there exist some functions $t_j : \{0,1\}^s \times \{0,1\}^s \to \{0,1\}$ which verify:

$$t(\boldsymbol{x}, \boldsymbol{y}) = 1 \quad \Longleftrightarrow \quad \forall j,\ 1 \leq j \leq z, \quad t_j(x_j, y_j) = 1.$$

***Problem 1** consists in computing the set $\mathcal{L}_{sol}$ of all 2-tuples $(\boldsymbol{x}, \boldsymbol{y})$ of $(L_1 \times L_2)$ verifying $t(\boldsymbol{x}, \boldsymbol{y}) = 1$. This operation is called* merging *the lists L_1 and L_2 with respect to t.*

One of the algorithms proposed in [5] to solve **Problem 1** is the *parallel matching* algorithm, which is the one that provides the best time complexity when the number of possible associated elements to one element is bigger than the size of the other list, *i.e.*, when we can associate by t more than $|L_2|$ elements to an element from L_1 as well as more than $|L_1|$ elements to an element from L_2.

In our case, the lists $\mathcal{L}_{\text{in}}$ and $\mathcal{L}_{\text{out}}$ correspond to the lists L_1 and L_2 to merge but the application of this algorithm differs in two aspects. The first one is the alphabet, which is not binary anymore but ternary. The second aspect is the distribution of vectors in the lists. In **Problem 1**, the elements are drawn uniformly and independently at random while in our case the distribution is ruled by the MITM technique we use. For instance, all the elements of $\mathcal{L}_{\text{in}}$ have the same weight x and all the elements of $\mathcal{L}_{\text{out}}$ have the same weight y, which is far from the uniform case.

The function t is the *association rule* we use to select suitable vectors from $\mathcal{L}_{\text{in}}$ and $\mathcal{L}_{\text{out}}$. We say that two elements are *associated* if their colliding active cells have the same values. We can now specify a new **Problem 1** adapted for ARMADILLO2:

ARMADILLO2 Problem 1. *Let $\mathcal{L}_{in}$ and $\mathcal{L}_{out}$ be 2 lists of ternary vectors of size $2^{\ell_{in}}$ and $2^{\ell_{out}}$ respectively. We denote by $\boldsymbol{x}$ a vector of $\mathcal{L}_{in}$ and by $\boldsymbol{y}$ a vector of $\mathcal{L}_{out}$, with $\boldsymbol{x}, \boldsymbol{y} \in \{0,1,-\}^k$*
The lists $\mathcal{L}_{in}$ and $\mathcal{L}_{out}$ are obtained by the MITM technique described in Paragraph 3.1. Let $t : \{0,1,-\}^k \times \{0,1,-\}^k \to \{0,1\}$ be the function defined by $t = t_1 \cdot t_2 \cdots t_{k-1} \cdot t_k$ and:

$$\forall j,\ 1 \leq j \leq k, \quad t_j : \{0,1,-\} \times \{0,1,-\} \to \{0,1\},$$

x_j	0	0	0	1	1	1	−	−	−
y_j	0	1	−	0	1	−	0	1	−
$t_j(x_j, y_j)$	1	0	1	0	1	1	1	1	1

We say that $\boldsymbol{x}$ and $\boldsymbol{y}$ are associated if $t(\boldsymbol{x}, \boldsymbol{y}) = 1$.
***ARMADILLO2 Problem 1** consists in* merging *the lists $\mathcal{L}_{in}$ and $\mathcal{L}_{out}$ with respect to t.*

We can now adapt the parallel matching algorithm to ARMADILLO2 **Problem 1**.

3.3 Applying the Parallel Matching Algorithm to ARMADILLO2

The principle of the parallel matching algorithm is to consider *in parallel* the possible matches for the α first cells and the next β cells in the lists $\mathcal{L}_{\text{in}}$ and $\mathcal{L}_{\text{out}}$. The underlying idea is to improve, when possible, the complexity to find all the elements that are a match for the $(\alpha+\beta)$ first cells. To have a match between a vector in $\mathcal{L}_{\text{in}}$ and a vector in $\mathcal{L}_{\text{out}}$, the vectors should satisfy:

- the vector in $\mathcal{L}_{\text{in}}$ has u of its x active cells among the $(\alpha+\beta)$ first cells;
- the vector in $\mathcal{L}_{\text{out}}$ has v of its y active cells among the $(\alpha+\beta)$ first cells;
- looking at the $(\alpha+\beta)$ first cells, both vectors should have the same value at the same active position.

As x and y are the number of known bits from $(C\|U)$ and from Y resp. (see Fig. 2), the matching probability on the first $(\alpha+\beta)$ cells is:

$$2^{-N_{\text{coll}}^{\alpha+\beta}} = \sum_{u=0}^{x} P_{[k,\alpha+\beta,x]}(u) \cdot \sum_{v=0}^{y} P_{[k,\alpha+\beta,y]}(v) \cdot \sum_{w=0}^{v} 2^{-w} P_{[\alpha+\beta,v,u]}(w).$$

This means that we will find $2^{c-N_{\text{coll}}^{\alpha+\beta}}$ partial solutions. For each pair passing the test we will have to check next if the remaining $k-\alpha-\beta$ cells are verified.

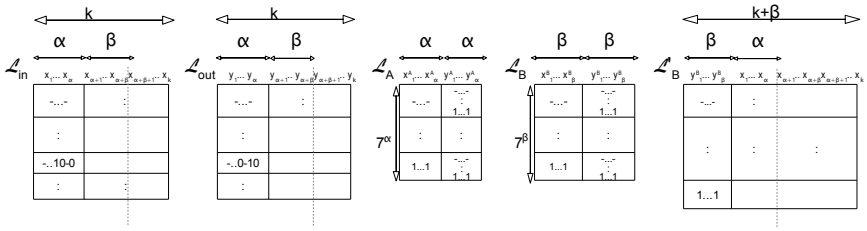

Fig. 3. Lists used in the parallel matching algorithm.

In a pre-processing phase, we first need to build three lists, namely $\mathcal{L}_A$, $\mathcal{L}_B$, $\mathcal{L}'_B$, which are represented in Fig. 3.

List $\mathcal{L}_A$ contains all the elements of the form $(x_1^A \dots x_\alpha^A, y_1^A \dots y_\alpha^A)$ with $(x_1^A \dots x_\alpha^A) \in \{0,1,-\}^\alpha$ and $(y_1^A \dots y_\alpha^A)$ being associated to $(x_1^A \dots x_\alpha^A)$. The size of $\mathcal{L}_A$ is:

$$|\mathcal{L}_A| = \sum_{i=0}^{\alpha} \left(\binom{\alpha}{i} 2^i 3^{\alpha-i} 2^i \right) = 7^\alpha.$$

List $\mathcal{L}_B$ contains all the elements of the form $(x_1^B \dots x_\beta^B, y_1^B \dots y_\beta^B)$ with $(x_1^B \dots x_\beta^B) \in \{0,1,-\}^\beta$ and $(y_1^B, \dots, y_\beta^B)$ being associated to $(x_1^B, \dots, x_\beta^B)$. The size of $\mathcal{L}_B$ is:

$$|\mathcal{L}_B| = \sum_{i=0}^{\beta} \left(\binom{\beta}{i} 2^i 3^{\beta-i} 2^i \right) = 7^\beta.$$

List $\mathcal{L}'_B$ contains for each element $(x_1^B, \dots, x_\beta^B, y_1^B, \dots, y_\beta^B)$ in $\mathcal{L}_B$ all the elements $\boldsymbol{x}$ from $\mathcal{L}_{\text{in}}$ such that $(x_{\alpha+1} \dots, x_{\alpha+\beta}) = (x_1^B, \dots, x_\beta^B)$. Elements in $\mathcal{L}'_B$ are of the form $(y_1^B, \dots, y_\beta^B, x_1, \dots, x_k)$ indexed[7] by $(y_1^B \dots, y_\beta^B, x_1, \dots, x_\alpha)$. The probability

[7] We can use standard hash tables for storage and look up in constant time.

for an element in $\mathcal{L}_{\text{in}}$ to have i active cells in its next β cells is $P_{[k,\beta,x]}(i)$. The size of $\mathcal{L}'_B$ is:

$$|\mathcal{L}'_B| = \sum_{i=0}^{\beta} \binom{\beta}{i} 2^i 3^{\beta-i} 2^i 2^{\ell_{\text{in}}} \frac{P_{[k,\beta,x]}(i)}{2^i \binom{\beta}{i}} = \sum_{i=0}^{\beta} 3^{\beta-i} 2^i 2^{\ell_{\text{in}}} P_{[k,\beta,x]}(i).$$

The cost of building $\mathcal{L}'_B$ is upper bounded by $(|\mathcal{L}'_B| + 3^\beta)$, where 3^β captures the cases where no element in $\mathcal{L}_{\text{in}}$ corresponds to elements in $\mathcal{L}_B$ and is normally negligible.

Next, we do the parallel matching. The probability for an element in $\mathcal{L}_{\text{out}}$ to have i active cells in its α first cells being $P_{[k,\alpha,y]}(i)$, for each element $(x_1^A \dots x_\alpha^A, y_1^A \dots y_\alpha^A)$ in $\mathcal{L}_A$ we consider the $2^{\ell_{\text{out}}} \frac{P_{[k,\alpha,y]}(i)}{2^i \binom{\alpha}{i}}$ elements $\boldsymbol{y}$ from $\mathcal{L}_{\text{out}}$ such that $(y_1, \dots, y_\alpha) = (y_1^A, \dots, y_\alpha^A)$. Then we check in $\mathcal{L}'_B$ if elements indexed by $(y_{\alpha+1} \dots y_{\alpha+\beta}, x_1^A \dots x_\alpha^A)$ exist. If this is the case, we check if each found pair of the form $(\boldsymbol{x}, \boldsymbol{y})$ verifies the remaining $(k - \alpha - \beta)$ cells. As we already noticed, we will find about $2^{c - N_{\text{coll}}^{\alpha+\beta}}$ partial solutions for which we will have to check whether or not they meet the remaining conditions. The time complexity of this algorithm is:

$$\mathcal{O}\left(2^{c-N_{\text{coll}}^{\alpha+\beta}} + 7^\alpha + 7^\beta + \sum_{i=0}^{\beta} 3^{\beta-i} 2^i 2^{\ell_{\text{in}}} P_{[k,\beta,x]}(i) + \sum_{i=0}^{\alpha} 3^{\alpha-i} 2^i 2^{\ell_{\text{out}}} P_{[k,\alpha,y]}(i)\right).$$

The memory complexity is determined by $7^\alpha + 7^\beta + |\mathcal{L}'_B|$. We can notice that if

$$\sum_{i=0}^{\beta} 3^{\beta-i} 2^i 2^{\ell_{\text{in}}} P_{[k,\beta,x]}(i) > \sum_{i=0}^{\alpha} 3^{\alpha-i} 2^i 2^{\ell_{\text{out}}} P_{[k,\alpha,y]}(i),$$

we can exchange the roles of $\mathcal{L}_{\text{in}}$ and $\mathcal{L}_{\text{out}}$, so that the time complexity remains the same but the memory complexity will be reduced. The memory complexity is then:

$$\mathcal{O}\left(7^\alpha + 7^\beta + \min\left\{\sum_{i=0}^{\beta} 3^{\beta-i} 2^i 2^{\ell_{\text{in}}} P_{[k,\beta,x]}(i), \sum_{i=0}^{\alpha} 3^{\alpha-i} 2^i 2^{\ell_{\text{out}}} P_{[k,\alpha,y]}(i)\right\}\right).$$

4 Meet in the Middle Key Recovery attacks

4.1 Key Recovery Attack in the FIL-MAC Setting

In the FIL-MAC usage scenario, C is the secret key and U is the challenge. The response is the m-bit size vector V_t. In order to minimize the complexity of our attack, we want the number of known bits y from Y to be maximal. As $Y = (V_c \| V_t) \oplus X$ and $X = Q_U(C \| U)$ it means that we are interested in having the maximum number of bits from U among the m less significant bits of X.

As we have m bits of freedom in U for choosing the permutation Q_U, we need the probability of having i known bits (from U) among the m first ones (of X), $P_{[k,m,m]}(i)$, to be bigger than 2^{-m}. Then to maximize the number of known bits in Y, we choose y as follows:

$$y = \max_{0 \le i \le m} \left\{ i : P_{[k,m,m]}(i) > 2^{-m} \right\}. \tag{1}$$

For instance for ARMADILLO2-A, we have $y = 38$ with a probability of $2^{-45.19} > 2^{-48}$.

Then, from now on, we assume that we know y among the m bits of the lower part of X and y bits at the same positions of Y.

Now, we can apply our meet-in-the-middle technique which allows us to recover the key. We have computed the optimal parameters for the different versions of

ARMADILLO2, with different trade-offs — the generic attack has a complexity of 2^c. The results appear in Table 2.

For each version of ARMADILLO2 presented in Table 2, the first line corresponds to the ($\log_2$ of the) size of the lists $\mathcal{L}_{\text{in}}$ and $\mathcal{L}_{\text{out}}$ with the smallest time complexity. The second line corresponds to the best parameters when limiting the memory complexity to 2^{45}. In all cases, the complexity is determined by the parallel matching part of the attack. The data complexity of all the attacks is 1, that is, we only need one pair of plaintext/ciphertext to succeed.

Version	c	m	ℓ_{out}	ℓ_{in}	α	β	$\log_2$(Time compl.)	$\log_2$(Mem. compl.)
ARMADILLO2-A	80	48	34	46	24	20	72.54	68.94
			18	62	16	9	75.05	45
ARMADILLO2-B	128	64	58	70	35	35	117.97	108.87
			38	90	2	16	125.15	45
ARMADILLO2-C	160	80	76	84	43	43	148.00	135.90
			35	125	4	16	156.63	45
ARMADILLO2-D	192	96	92	100	50	50	177.98	160.44
			29	163	11	12	187.86	45
ARMADILLO2-E	256	128	125	131	65	65	237.91	209.83
			29	227	11	13	251.55	45

Table 2. Complexities of the meet-in-the-middle key recovery attack on the FIL-MAC application

4.2 Key Recovery Attack in the Stream Cipher Setting

As presented in [1], ARMADILLO2 can be used as a PRNG by taking the t first bits of (V_c, V_t) after at least r iterations. For ARMADILLO2, the authors state in [1, Sc. 6] that $r = 1$ and $t = k$ is a suitable parameter choice. If we want to use it as a stream cipher, the secret key is C. The keystream is composed of k-bit frames indexed by U which is a public value.

In this setting, we can perform an attack which is similar to the one on the FIL-MAC, but with different parameters. As we know more bits of the output of Q_X, $y = m + \ell_{\text{out}}$, complexities of the key recovery attack are lower.

In general, the best time complexity is obtained when $\ell_{\text{in}} = \ell_{\text{out}}$, as the number of known bits at each side is now $x = m + \ell_{\text{in}}$ in the input and $y = m + \ell_{\text{out}}$ in the output. In this context it also appears that the best time complexity occurs when $\alpha = \beta$. There might be a small difference between α and β when the leading term of the time complexity is $2^{c - N_{\text{coll}}^{\alpha+\beta}}$.

We present the best complexities we have computed for this attack in Table 3 — the generic attack has a complexity of 2^c. Other time-memory trade-offs would be possible. As in the previous section, we give as an example the best parameters when limiting the memory complexity to 2^{45}.

5 (Second) Preimage Attack on the Hashing Applications

We recall that the hash function built with ARMADILLO2 as a compression function follows a strengthened Merkle-Damgård construction, where the padding includes the message length. In this case C represents the input chaining value, U the message block

Version	c	m	ℓ_{out}	ℓ_{in}	α	β	$\log_2$(Time compl.)	$\log_2$(Mem. compl.)
ARMADILLO2-A	80	48	40	40	19	19	65.23	62.91
			27	53	11	16	71.62	45
ARMADILLO2-B	128	64	64	64	31	32	104.71	101.75
			29	99	9	16	119.69	45
ARMADILLO2-C	160	80	80	80	39	40	130.53	127.49
			26	134	14	14	151.29	45
ARMADILLO2-D	192	96	96	96	47	48	156.35	153.23
			30	162	8	16	184.37	45
ARMADILLO2-E	256	128	128	128	64	64	207.96	205.93
			30	226	8	16	248.66	45

Table 3. Complexities of the meet-in-the-middle key recovery attack for the stream cipher with various trade-offs.

and V_c the generated new chaining value and the hash digest. In [1] the authors state that (second) preimages are expected with a complexity of 2^c, the one of the generic attack. We show, in this section, how to build (second) preimage attacks with a smaller complexity.

5.1 Meet-in-the-Middle (Second) Preimage Attack

The principle of the attack is represented in Fig. 5.1. We first consider that the ARMADILLO2 function is invertible with a complexity of 2^q, given an output V_c and a message block. In the preimage attack, we choose and fix ℓ, the number of blocks of the preimage. In the second preimage attack, we can consider the length of the given message. Then, given a hash value h:

In the backward direction:

- We invert the insertion of the last block M_{pad} (padding). This step costs 2^q in a preimage scenario and 1 in a second preimage one. We get
$$\text{ARMADILLO2}^{-1}(h, M_{\text{pad}}) = S'.$$
- From state S', we can invert the compression function for 2^b different message blocks M_b with a cost 2^{b+q}, obtaining 2^b different intermediate states: $\text{ARMADILLO2}^{-1}(S', M_b) = S''$.

In the forward direction: From the initial chaining value, we insert 2^a messages of length $(\ell - 2)$ blocks, $\mathcal{M} = M_1 \| M_2 \| \dots \| M_{\ell-2}$, obtaining 2^a intermediate states S. This can be done with a complexity of $\mathcal{O}((\ell - 2)2^a)$.

If we find a collision between one of the 2^a states S and one of the 2^b states S'', we have obtained a (second) preimage that is $\mathcal{M} \| M_b \| M_{\text{pad}}$.

A collision occurs if $a + b \geq c$. The complexity of this attack is $2^a + 2^q + 2^{b+q}$ in time, where the middle term appears only in the case of a preimage attack and is negligible. The memory complexity is about 2^b (plus the memory needed for inverting the compression function). So if $2^q < 2^c$, we can find a and b so that $2^a + 2^{b+q} < 2^c$.

5.2 Inverting the Compression Function

In the previous section we showed that inverting the compression function for a chosen message block and for a given output can be done with a cost of $2^q < 2^c$. In this section we show how this complexity depends on the chosen message block, as the inversion

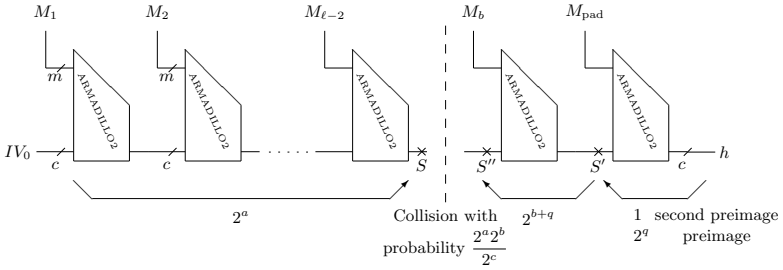

Fig. 4. Representation of the meet-in-the-middle (second) preimage attack.

can be seen as a key recovery similar to the one done in Section 4. In this case we know U (the message block) and V_c, and we want to find C. When inverting the function with the blocks M_b, we choose message blocks (U) that define permutations Q_U which put most of the m bits from U among the c most significant bits of X. This will result in better attacks, as the bits in Y known from U do not cost anything and this gives us more freedom when choosing the parameters ℓ_{in} and ℓ_{out}.

As before, we have 2^m possibilities for Q_U. We denote by n the number of bits of U in the c most significant bits of X. The number of message blocks (U) verifying this condition is:

$$N_{\text{block}}(n) = 2^m P_{[k,c,m]}(n).$$

In fact we are interested in the values of n which are the greatest possible (to lower the complexity) that still leaves enough message blocks to invert in order to obtain S''. It means that these values belong to a set $\{n_i\}$ such that:

$$\sum_{\{n_i\}} N_{\text{block}}(n_i) \geq 2^b.$$

As the output is V_c, the ℓ_{out} bits guessed from X are also known bits from the output of Q_X. The number of known bits of the output of Q_X is then defined by:

$$y = \min(c, \ell_{\text{out}} + n)$$

Compared to the key recovery attack, the number of known bits at the end of the permutation Q_X is significantly bigger, as we may know up to c bits, while in the previous case the maximal number for y was $y = \max_i \left\{i : P_{[k,m,m]}(i) > 2^{-m}\right\}$. To simplify the explanations, we concentrate on the case of ARMADILLO2-A, that can be directly adapted to any of the other versions. For $n = 48$ we have a probability $P_{[128,80,48]} = 2^{-44.171}$. This leaves $2^{48-44.171} = 2^{3.829}$ message blocks to invert which allow us to know $y = \min(80, \ell_{\text{out}} + 48)$ bits from the output of Q_X. As we need to invert 2^b message blocks, if b is bigger than 3.829, we have to consider next the message blocks with $n = 47$, that allow us to know $y = \min(80, \ell_{\text{out}} + 47)$ bits, and so on. For each n considered, the best time complexity (2^{q_n}) for inverting ARMADILLO2 might be different, but in practice, with at most two consecutive values of n we have enough message blocks for building the attack, and the complexity of inverting the compression function for these two different types of messages is very similar.

For instance, in ARMADILLO2-A, we consider $n = 48, 47$, associated each to $2^{3.829}$ and $2^{9.96}$ possible message blocks respectively. The best time complexity for inverting the compression function in both cases is $2^{q_{48}} = 2^{q_{47}} = 2^{65.9}$, as we can see from Table 4. If we want to find the best parameters for a and b in the preimage attack, we can consider that $a+b = c$ and $2^b = 2^{b_{48}} + 2^{b_{47}}$, and we want that $2^a = 2^{b_{48}} 2^{65.9} + 2^{b_{47}} 2^{65.9} = 2^{65.9}(2^{b_{48}} + 2^{b_{47}})$, as the complexity of the attack is $\mathcal{O}(2^a + 2^{65.9}(2^{b_{48}} + 2^{b_{47}}))$. So if we choose the parameters correctly, the best time complexity will be $\mathcal{O}(2^{a+1})$.

Version	c	m	ℓ_{out}	ℓ_{in}	n	$\log_2(N_{\text{block}}(n))$	α	β	$\log_2$(Time compl.)	$\log_2$(Mem. compl.)
ARMADILLO2-A	80	48	35	45	47	9.95	22	16	65.90	63.08
			35	45	48	3.83	22	16	65.90	63.08
			20	60	47	9.95	16	8	71.36	45
			27	53	48	3.83	11	16	71.62	45
ARMADILLO2-B	128	64	62	66	64	15.89	33	30	104.67	102.35
			33	95	64	15.89	6	16	120.41	45
ARMADILLO2-C	160	80	78	82	80	19.82	41	38	130.48	128.08
			26	134	80	19.82	11	16	152.24	45
ARMADILLO2-D	192	96	94	98	96	23.74	49	46	156.31	153.82
			30	162	96	23.74	8	16	184.37	45
ARMADILLO2-E	256	128	126	130	128	31.58	65	62	207.96	205.30
			34	222	128	31.58	5	16	249.47	45

Table 4. Complexities for inverting the compression function.

In this particular case the time complexity for $n = 48$ and for $n = 47$ is the same, so finding the best b and a can be simplified by $b = \frac{c-q}{2}$ and $a = c - b$. We obtain $b = 7.275$, $a = 72.95$. We see that we do not have enough elements with $n = 48$ for inverting 2^b blocks, but we have enough with $n = 47$ alone. As the complexities are the same in both cases, we can just consider $b = b_{47}$. The best time complexity for the preimage attack that we can obtain is then $2^{73.95}$, with a memory complexity of $2^{63.08}$. Other trade-offs are possible by using other parameters for inverting the function, as shown in Table 5.

For the other versions of ARMADILLO2, the number of message blocks associated to $y = m$ is big enough for performing the 2^b inversions, so we do not consider other n's for computing the (second) preimage complexity. Then, $b = b_m = \frac{c-q_{\{n=m\}}}{2}$ and $a = c - b_m$.

Complexities for preimage attacks on the different versions of ARMADILLO2 are given in Table 5, where we can see two different complexities with different trade-offs for each version.

Version	c	m	Best time		Time-memory trade-off	
			$\log_2$(Time compl.)	$\log_2$(Mem. compl.)	$\log_2$(Time compl.)	$\log_2$(Mem. compl.)
ARMADILLO2-A	80	48	73.95	63.08	76.81	45
ARMADILLO2-B	128	64	117.34	102.35	125.21	45
ARMADILLO2-C	160	80	146.24	128.08	157.12	45
ARMADILLO2-D	192	96	175.16	153.82	191.19	45
ARMADILLO2-E	256	128	232.98	205.30	253.74	45

Table 5. Complexities of the (second) preimages attacks.

6 Experimental Verifications

To verify the above theoretical results, we implemented the proposed key recovery attacks in the FIL-MAC and stream cipher settings against a scaled version of ARMADILLO2 that uses a 30-bit key and processes 18-bit messages, i.e. $c = 30$ and $m = 18$. We performed the attack 10 times for both the FIL-MAC and the PRNG settings where at each time we chose random permutations for both σ_0 and σ_1 and random messages U (in the FIL-MAC case U was chosen so that we got y bits from U among the m least significant bits of X).

As for each application the key is a 30-bit key, the generic attack requires a time complexity of 2^{30}. Using the parallel matching algorithm we decrease this complexity. Table 6 shows that the implementation results are very close to the theoretical estimates, confirming our analysis. We can also mention that we exchanged the role of $\mathcal{L}_{\text{in}}$ and $\mathcal{L}_{\text{out}}$ in our implementation of the attacks to minimize the memory needs.

		c	m	ℓ_{out}	ℓ_{in}	α	β	y	$\log_2(\lvert\mathcal{L}'_B\rvert)$	$\log_2(c - N_{\text{coll}}^{\alpha+\beta})$	$\log_2$(Time compl.)	$\log_2$(Mem. compl.)
FIL-MAC	Impl.	30	18	12	18	8	6	14	23.477	27.537	27.874	24.066
	Theory	30	18	12	18	8	6	14	23.475	27.538	27.874	24.064
PRNG	Impl.	30	18	14	16	7	6	32	22.530	24.728	25.396	22.738
	Theory	30	18	14	16	7	6	32	22.530	24.735	25.401	22.738

Table 6. Key recovery attacks against a scaled version of ARMADILLO2 in the FIL-MAC and PRNG modes.

7 Generalization of the Parallel Matching Algorithm

In Section 3, we managed to apply the parallel matching algorithm to invert the ARMADILLO2 function by modifying the merging **Problem 1** of [5].

When the number of possible associated elements to one element is bigger than the other list as it is the case for ARMADILLO2, we cannot apply a basic algorithm like the *instant matching* algorithm proposed in [5]. Instead, we can use either the *gradual matching* or the *parallel matching* algorithms also proposed in [5]. We are going to concentrate on the parallel matching algorithm which allows a significant reduction of the time complexity of solving **Problem 1**, while allowing several time-memory trade-offs.

We can state the generalized problem that also covers our attack on ARMADILLO2 and give the corresponding parallel matching algorithm. We believe that this more general problem will be useful for recognizing situations where the parallel matching can be applied, and solving them in an automatized way.

7.1 The Generalized Problem 1

As stated in [5], **Problem 1** for N lists can be reduced to 2 lists, therefore we will only consider the problem of merging 2 lists in the sequel.

Generalized Problem 1. *We are given 2 lists, L_1 and L_2 of size 2^{ℓ_1} and 2^{ℓ_2} respectively. We denote by $\boldsymbol{x}$ a vector of L_1 and by $\boldsymbol{y}$ a vector of L_2. Coordinates of $\boldsymbol{x}$ and $\boldsymbol{y}$ belong to a general alphabet $\mathcal{A}$.*

We assume that vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ can be decomposed into z groups of s coordinates, i.e. $\boldsymbol{x}, \boldsymbol{y} \in (\mathcal{A}^s)^z$ and $\boldsymbol{x} = (x_1, \ldots, x_z)$ (resp. $\boldsymbol{y} = (y_1, \ldots, y_z)$).

We want to keep pairs of vectors verifying a given relation t: $t(\boldsymbol{x}, \boldsymbol{y}) = 1$. The relation t is group-wise,and is defined by $t : (\mathcal{A}^s)^z \times (\mathcal{A}^s)^z \to \{0,1\}$ such that there exist some functions $t_j : \mathcal{A}^s \times \mathcal{A}^s \to \{0,1\}$, verifying:

$$t(\boldsymbol{x}, \boldsymbol{y}) = 1 \iff \forall j,\ 1 \le j \le z, \quad t_j(x_j, y_j) = 1.$$

Generalized Problem 1 *consists in* merging *these* 2 *lists to obtain the set $\mathcal{L}_{sol}$ of all* 2*-tuples of $(L_1 \times L_2)$ verifying $t(\boldsymbol{x}, \boldsymbol{y}) = 1$. We say that $\boldsymbol{x}$ and $\boldsymbol{y}$ are associated in this case.*

In order to analyze the time and memory complexities of the attack we need to compute the size of $\mathcal{L}_{\text{sol}}$. This quantity depends on the probability that $t(\boldsymbol{x}, \boldsymbol{y}) = 1$. More precisely the complexities of the generalized parallel matching algorithm depends on the conditional probabilities: $\mathbf{Pr}_{y_j}[t_j(x_j, y_j) = 1 | x_j = a]$, $a \in \mathcal{A}^s$. We will denote these probabilities by $p_{j,a}$, $a \in \mathcal{A}^s$.

In [5] the elements of the lists L_1 and L_2 were binary (i.e. $\mathcal{A} = \{0,1\}$) and random, and the probability of each t_j of being verified did not depend on the elements x_j or y_j. Let us consider as an example the case where $s = 1$ and t_j tests the equality of x_j and y_j. We have:

$$\forall j,\ 1 \le j \le z, \quad p_{j,0} = p_{j,1} = \frac{1}{2}.$$

In the case of the ARMADILLO2 cryptanalysis that we present in this paper, the alphabet is ternary (i.e. $\mathcal{A} = \{0, 1, -\}$) and the association rule (*see. ARMADILLO2 Problem 1*) gives:

$$\forall j,\ 1 \le j \le z, \quad p_{j,0} = \frac{2}{3},\ p_{j,1} = \frac{2}{3} \text{ and } p_{j,-} = 1$$

7.2 Generalized Parallel Matching Algorithm

First we need to build the three following lists:

List $\mathcal{L}_A$, of all the elements of the form $(x_1^A, \ldots, x_\alpha^A, y_1^A, \ldots, y_\alpha^A)$ with $(x_1^A, \ldots, x_\alpha^A) \in (\mathcal{A}^s)^\alpha$ and $(y_1^A, \ldots, y_\alpha^A)$ being associated by t to $(x_1^A, \ldots, x_\alpha^A)$. The size of $\mathcal{L}_A$ is:

$$|\mathcal{L}_A| = \sum_{\mathbf{a} \in (\mathcal{A}^s)^\alpha} \prod_{j=1}^{\alpha} |\mathcal{A}|^s \, p_{j,\mathbf{a}_j}, \tag{2}$$

where $\mathbf{a}_j$ is the j-th coordinate of $\mathbf{a} \in (\mathcal{A}^s)^\alpha$.

List $\mathcal{L}_B$, of all the elements of the form $(x_1^B, \ldots, x_\beta^B, y_1^B, \ldots, y_\beta^B)$ with $(x_1^B, \ldots, x_\beta^B) \in (\mathcal{A}^s)^\beta$ and $(y_1^B, \ldots, y_\beta^B)$ being associated by t to $(x_1^B, \ldots, x_\beta^B)$. The size of $\mathcal{L}_B$ is

$$|\mathcal{L}_B| = \sum_{\mathbf{b} \in (\mathcal{A}^s)^\beta} \prod_{j=1}^{\beta} |\mathcal{A}|^s \, p_{j,\mathbf{b}_j},$$

where $\mathbf{b}_j$ is the j-th coordinate of $\mathbf{b} \in (\mathcal{A}^s)^\beta$.

List $\mathcal{L}'_B$, containing for each element $(x_1^B, \ldots, x_\beta^B, y_1^B, \ldots, y_\beta^B)$ in $\mathcal{L}_B$ all the elements $\boldsymbol{x}$ from L_1 such that $(x_{\alpha+1} \ldots, x_{\alpha+\beta}) = (x_1^B, \ldots, x_\beta^B)$. Elements in $\mathcal{L}'_B$ are of the form $(y_1^B, \ldots, y_\beta^B, x_1, \ldots, x_z)$ indexed[8] by $(y_1^B \ldots, y_\beta^B, x_1, \ldots, x_\alpha)$. If we denote by $\mathrm{P}_{\mathbf{b},[\alpha+1,\alpha+\beta],L_1}$ the probability of having an element $\boldsymbol{x}$ from L_1 such that $(x_{\alpha+1}, \ldots, x_{\alpha+\beta}) = \mathbf{b}$, the size of $\mathcal{L}'_B$ is:

$$|\mathcal{L}'_B| = \sum_{\mathbf{b} \in (\mathcal{A}^s)^\beta} \left(\prod_{j=1}^{\beta} |\mathcal{A}|^s \, p_{j,\mathbf{b}_j} \right) 2^{\ell_1} \mathrm{P}_{\mathbf{b},[\alpha+1,\alpha+\beta],L_1}.$$

[8] We can use standard hash tables for storage and look up in constant time.

The cost of building this list is upper-bounded by $(|\mathcal{L}'_B|+(|\mathcal{A}|)^\beta)$, where the second term captures the cases where no element in L_1 corresponds to elements in $\mathcal{L}_B$ and should be negligible.

In the case where

$$\sum_{\mathbf{a}\in(\mathcal{A}^s)^\alpha}\left(\prod_{j=1}^{\alpha}|\mathcal{A}|^s\, p_{j,\mathbf{a}_j}\right)2^{\ell_2}\mathrm{P}_{\mathbf{a},[\beta+1,\alpha+\beta],L_2} < \sum_{\mathbf{b}\in(\mathcal{A}^s)^\beta}\left(\prod_{j=1}^{\beta}|\mathcal{A}|^s\, p_{j,\mathbf{b}_j}\right)2^{\ell_1}\mathrm{P}_{\mathbf{b},[\alpha+1,\alpha+\beta],L_1}$$

we can swap L_1 and L_2, to reduce the memory complexity of the attack.

Next, we do the parallel matching. For each element $(x_1^A,\ldots,x_\alpha^A,y_1^A,\ldots,y_\alpha^A)$ in $\mathcal{L}_A$ we consider the $2^{\ell_2}\mathrm{P}_{(y_1^A,\ldots,y_\alpha^A),[1,\alpha],L_2}$ elements $\boldsymbol{y}$ from L_2 such that $(y_1\ldots y_\alpha)=(y_1^A,\ldots,y_\alpha^A)$ and we check in $\mathcal{L}'_B$ if elements indexed by $(y_{\alpha+1}\ldots y_{\alpha+\beta},x_1^A\ldots x_\alpha^A)$ exist. If this is the case, we check if each found pair of the form $(\boldsymbol{x},\boldsymbol{y})$ verifies the remaining $(k-\alpha-\beta)$ cells. We denote by Ω the number of partial solutions for which we will have to check whether or not they meet the remaining conditions:

$$\Omega = 2^{\ell_1+\ell_2}\sum_{\mathbf{b}\in(\mathcal{A}^s)^{\alpha+\beta}}\left(\prod_{j=1}^{\alpha+\beta}p_{j,\mathbf{b}_j}\right)\mathrm{P}_{\mathbf{b},[1,\alpha+\beta],L_1}$$

The time complexity of this algorithm is:

$$\mathcal{O}\left(\Omega+|\mathcal{L}_A|+|\mathcal{L}_B|+|\mathcal{L}'_B|+\sum_{\mathbf{a}\in(\mathcal{A}^s)^\alpha}\left(\prod_{j=1}^{\alpha}|\mathcal{A}|^s\, p_{j,\mathbf{a}_j}\right)2^{\ell_2}\mathrm{P}_{\mathbf{a},[\beta+1,\alpha+\beta],L_2}\right)$$

The memory complexity is determined by the size of the lists $\mathcal{L}_A$, $\mathcal{L}_B$ and $\mathcal{L}'_B$. Therefore the memory complexity is:

$$\sum_{\mathbf{a}\in(\mathcal{A}^s)^\alpha}\prod_{j=1}^{\alpha}|\mathcal{A}|^s\, p_{j,\mathbf{a}_j}+\sum_{\mathbf{b}\in(\mathcal{A}^s)^\beta}\prod_{j=1}^{\beta}|\mathcal{A}|^s\, p_{j,\mathbf{b}_j}+\sum_{\mathbf{b}\in(\mathcal{A}^s)^\beta}\left(\prod_{j=1}^{\beta}|\mathcal{A}|^s\, p_{j,\mathbf{b}_j}\right)2^{\ell_1}\mathrm{P}_{\mathbf{b},[\alpha+1,\alpha+\beta],L_1}$$

7.3 Link with Formulas in the Case of ARMADILLO

Using the previous formulas for the time and memory complexities, we can rediscover formulas of the time and memory complexities we have computed for ARMADILLO2 (see. Section 3.3). As these formulas depend essentially on the size of the different lists, we simply expose how to find the size of the list $|\mathcal{L}_A|$ using equation (2).

For ARMADILLO2, the probabilities $p_{j,a}$ are independent of the position j and $p_{j,a}=2/3$ if and only if a is an active cell. Moreover, in this case, each cell is composed of one letter of the alphabet which means that $s=1$. And we have:

$$\begin{aligned}|\mathcal{L}_A| &= \sum_{\mathbf{a}\in(\mathcal{A}^s)^\alpha}\prod_{j=1}^{\alpha}|\mathcal{A}|^s\, p_{j,\mathbf{a}_j} = \sum_{\mathbf{a}\in\{0,1,-\}^\alpha}\prod_{j=1}^{\alpha}3\left(\frac{2}{3}\right)^{\mathrm{wt}(\mathbf{a})}\\ &= \sum_{i=0}^{\alpha}\#\left\{\mathbf{a}:\mathrm{wt}(\mathbf{a})=i\right\}3^\alpha\left(\frac{2}{3}\right)^i = \sum_{i=0}^{\alpha}\binom{\alpha}{i}2^i\left(\frac{2}{3}\right)^i 3^\alpha\end{aligned}$$

The same method can be applied to find the size of the list $\mathcal{L}_B$ and $\mathcal{L}'_B$. Here we have $\Omega = 2^{c-N_{\mathrm{coll}}^{\alpha+\beta}}$.

8 Conclusion

In this paper, we have presented the first cryptanalysis of ARMADILLO2, the recommended variant of the ARMADILLO family. We propose a key recovery attack on all its versions for the FIL-MAC and the stream cipher mode, which works for any bitwise permutations σ_0 and σ_1. We give several time-memory trade-offs for its complexity. We also show how to build (second) preimage attacks when using the hashing mode.

Besides the results on ARMADILLO2, we have generalized the parallel matching algorithm presented in [5] for solving a wider **Problem 1** which includes the cases where the lists to merge do not have random elements. We believe that new types of meet-in-the-middle attacks might appear now given this algorithm that is cheaper than exhaustive search.

References

1. Badel, S., Dagtekin, N., Nakahara, J., Ouafi, K., Reffé, N., Sepehrdad, P., Susil, P., Vaudenay, S.: ARMADILLO: A Multi-purpose Cryptographic Primitive Dedicated to Hardware. In: Workshop on Cryptographic Hardware and Embedded Systems, CHES 2010. Lecture Notes in Computer Science, vol. 6225, pp. 398–412 (2010)
2. Khovratovich, D., Naya-Plasencia, M., Röck, A., Schläffer, M.: Cryptanalysis of Luffa v2 components. In: Selected Areas in Cryptography 17th International Workshop, SAC 2010. Lecture Notes in Computer Science, vol. 6544, pp. 388–409 (2010)
3. Moldovyan, A.A., Moldovyan, N.A.: A Cipher Based on Data-Dependent Permutations. Journal of Cryptology 15(1), 61–72 (2002)
4. Naya-Plasencia, M.: How to Improve Rebound Attacks. Tech. Rep. Report 2010/607, Cryptology ePrint Archive (2010), (extended version). http://eprint.iacr.org/2010/607.pdf
5. Naya-Plasencia, M.: How to Improve Rebound Attacks. In: Advances in Crypology: CRYPTO 2011. Lecture Notes in Computer Science, vol. 6841, pp. 188–205 (2011)
6. Sepehrdad, P., Sušil, P., Vaudenay, S.: Fast Key Recovery Attack on ARMADILLO1 and Variants. In: Tenth Smart Card Research and Advanced Application Conference, CARDIS 2011. Lecture Notes in Computer Science (2011), to appear