Cryptanalysis of ARMADILLO2 *

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

[1] INRIA, project-team SECRET, France
[2] FHNW, Windisch, Switzerland
[3] ANSSI and Université Henri Poincaré-Nancy 1 / LORIA, France
[4] Technical University of Denmark, Department of Mathematics, Denmark

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 propose a meet-in-the-middle technique that allows us to invert the ARMADILLO2 function. Using this technique we are able to perform a key recovery attack on ARMADILLO2 in FIL-MAC application mode. A variant of this attack can also be applied when ARMADILLO2 is used as a stream cipher in the PRNG application mode. Finally we propose a (second) preimage attack on its hashing application mode. We have validated our attacks by implementing cryptanalysis on scaled variants that match the theoretical predicted complexities. All the cryptanalysis presented in this paper can be applied for any arbitrary bitwise permutations σ_0 and σ_1 used in the internal permutation.

Key words: ARMADILLO2, meet-in-the-middle, key recovery attack, preimage attack, parallel matching

1 Introduction

ARMADILLO is a multi-purpose cryptographic primitive dedicated to hardware which has been proposed by Badel et al. in [1]. Two variants have been 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. For both variants, several applications are proposed: FIL-MAC, hash function and pseudo-random generator. Both variants comprise several versions, each one associated to a different set of parameters and to a different security level. In [1] a security analysis of ARMADILLO1 has been given. In order to address security concerns, the authors have defined ARMADILLO2 and stated that it is the design choice to be preferred.

The ARMADILLO family uses a parametrized internal permutation as 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.

*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

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

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 [4]. Our methods enable 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 PNRG 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 validate the theoretical formulas.

We briefly describe ARMADILLO2 in Section 2. In Section 3 our technique for inverting the building block is presented. In Section 4, we explain how to apply this technique for building a key recovery attack on the FIL-MAC application mode. We show briefly 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 briefly the experiments that we have done.

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 value 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 value (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 parametrized permutation denoted by Q which is defined by a parameter A of size a and is applied to a vector B of size k:

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

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 involved in two parts of the ARMADILLO2 function. Let A be a parameter and B be the internal state, the parametrized permutation Q (that we denote by Q_A when showing which is the parameter used is required) consists in $a = |A|$ simple steps. The i-th step of Q (reading A from its less significant bit to its most significant one) is defined by:

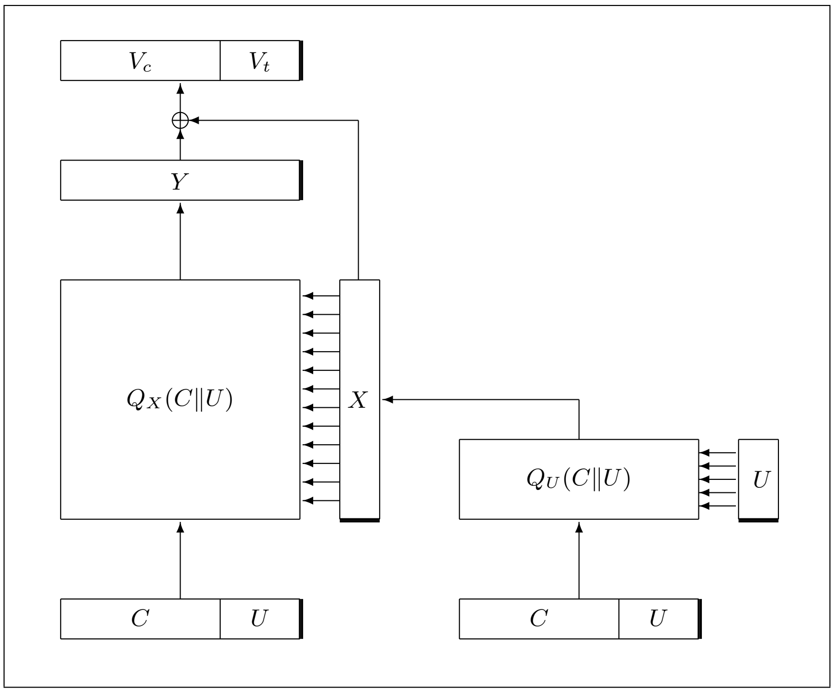

Fig. 1. ARMADILLO2.

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

Actually the variables c and m can take different values depending on the security level aimed at. A summary of the sets of parameters proposed in [1] for the different versions (A, B, C, D or E) is given in Tab. 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 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.* In this application the variable C is the secret and U is a challenge which can be considered known by the attacker. The value V_t corresponds to the response.

ARMADILLO*2 in hashing mode.* The hash mode of ARMADILLO2 uses a strengthened Merkle-DamgArd construction, where V_c is the chaining value or the hash digest, and U is the message block.

ARMADILLO*2 in PRNG and PRF mode.* The PRNG 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 $t = k$ [1, Sec. 6]. If we want to use it as a stream cipher, the secret key is C. The keystream is then composed of k-bit frames and U is the index of the output frame, that is a public value.

3 Inverting the ARMADILLO2 function

In this section we describe how to invert the ARMADILLO2 function when U is chosen and some bits of the output (V_c, V_t) are known. Inverting means that we recover the C part of the input.

For this purpose, we use a meet-in-the-middle approach which can be performed for any arbitrary bit-wise permutations σ_0 and σ_1. To conduct our analysis we suppose that they behave like random ones.

In [1] a sketch of a meet-in-the-middle (MITM) attack on the first variant of the primitive, ARMADILLO1, is given by the authors to prove *lower bounds* for the complexity and justify the choice of parameters.

Here, we first present a meet-in-the-middle technique applied to the recommended version, ARMADILLO2, when only a part of the output is known (V_t, (V_c, V_t) or V_c). With this technique, we obtain two lists of partial states in the middle of the permutation: one computed in the forward direction and one in the backward direction. Next we propose a method to find all the pairs formed by one element from each list that show consistency in the middle of the permutation. Our method has lower complexity than exhaustive search, contrary to the naive way that would cost as much. This second part is indeed the bottleneck of the time complexity of inverting ARMADILLO2.

All the cryptanalyses that we present in further sections on the different applications of ARMADILLO2 rely 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 fact that we can choose the value U. Therefore, as U is known and the permutation Q_U is parametrized by U we know $m = |U|$ bits of $X = Q_U(C\|U)$ and their positions. The number of bits of Y that can also be considered as known depends on the application. We will denote it by y. To summarize, on the one hand we have $Y = Q_X(C\|U)$ and we know m bits of the input $(C\|U)$ of Q_X, and on the other hand, we know y bits of the output $Y = (V_c\|V_t) \oplus X$. Then we can use a meet-in-the middle technique to recover a consistent C.

We divide X into two parts X_{in} and X_{out} such that $X = (X_{\text{out}}\|X_{\text{in}})$. Let us call the division line between $Q_{X_{\text{in}}}$ and $Q_{X_{\text{out}}}$ the middle of Q_X. As previously mentioned, m bits of $X = Q_U(C\|U)$ are already known from U. We denote by m_{in} the number of bits of U that are in X_{in} and by m_{out} the number of bits of U that are in X_{out}. We have $m_{\text{out}} + m_{\text{in}} = m$. Therefore $c = k - m$ bits of X remain unknown. We denote by ℓ_{out}

the number of unknown bits in X_{out} and by ℓ_{in} the number of unknown bits in X_{in} and we have $\ell_{\text{out}} + \ell_{\text{in}} = c$. These bits come from some particular bits of C depending on the permutation Q_U. The meet-in-the-middle attack is done by guessing the ℓ_{in} bits and the ℓ_{out} bits independently.

The lower part X_{in} determines $Q_{X_{\text{in}}}$, the first $\ell_{\text{in}} + m_{\text{in}}$ rounds of Q_X (from the beginning to the middle). We can trace the ℓ_{in} guessed bits of these lower part back to C with Q_U. Next, for each possible guess of the ℓ_{in} bits, we can trace these ℓ_{in} bits plus the m ones from U to their positions in the middle computing forward $Q_{X_{\text{in}}}(C\|U)$. For each one of the $2^{\ell_{\text{in}}}$ possibilities for these bits, we will obtain $x = \ell_{\text{in}} + m$ known bits at some positions in the middle.

The upper part X_{out} determines $Q_{X_{\text{out}}}$, the last $\ell_{\text{out}} + m_{\text{out}}$ rounds of the permutation Q_X (from the middle to the end). We assume that we know y bits of Y. We can trace them back to the middle of Q_X for each possibility of these $2^{\ell_{\text{out}}}$ bits of the upper part, that is computing $Q^{-1}_{X_{\text{out}}}(Y)$.

To describe the meet-in-the-middle attack we represent the vectors that we are trying to match in the middle as ternary words whose cells can contain the values 0, 1 or nothing $(-)$. A cell is said to be *active* if it contains 0 or 1 and *inactive* otherwise. The weight of a vector is the number of active cells it contains.

We deal with 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 (vectors) in the middle of Q_X. The list $\mathcal{L}_{\text{in}}$ contains elements $Q_{X_{\text{in}}}(C\|U)$ coming from the forward direction whose weights are $x = \ell_{\text{in}} + m$. The list $\mathcal{L}_{\text{out}}$ contains elements $Q^{-1}_{X_{\text{out}}}(Y)$ coming from the backwards direction whose weights are y.

The probability that, when considering one element from each list we find a match will depend on the number of collisions on the active cells we will have in this pair of elements.

Considering a binary vector of k bits A, of weight a, we will denote by $P_{[k,a,b]}(i)$ the probability over all the vectors B of weight b of having $\text{wt}(A \cdot B) = i$, where $\cdot$ denotes the bitwise AND. We have:

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

This notation will be used frequently throughout the rest of the paper.

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 first test. Each pair of elements defines a whole key. Next, we just have to test these keys to find the correct one.

The question now is what is the cost of checking which elements of the two lists $\mathcal{L}_{\text{in}}$ and $\mathcal{L}_{\text{out}}$ pass the test. Let us remark that the active cells are at different positions in each element, that an inactive cell can be associated to anything (0, 1, or $(-)$), and that an active cell can be associated to its own value or to an inactive cell. This makes it impossible to apply the approach of having an ordered list $\mathcal{L}_{\text{in}}$ and then checking for each element in the list $\mathcal{L}_{\text{out}}$ if a match exists with a cost 1 per element, as is the case for traditional MITM attacks. 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 are going to use an adaptation of the algorithm described in [4, Sec. 3] as *parallel matching.*

In our case, the application of this algorithm will be more complicated as the elements are not uniformly distributed but have very specific distributions. That is, in previously described parallel matching applications, all the potential elements for the lists $\mathcal{L}_{\text{in}}$ or $\mathcal{L}_{\text{out}}$ had the same probability of occurring, which is not the case here.

3.2 Parallel matching with non-random elements

Translated to our study case, the parallel matching algorithm will consider the possible matches for the α first cells, and in parallel the possible matches for the next β cells in the lists $\mathcal{L}_{\text{in}}$ and $\mathcal{L}_{\text{out}}$. This way we will find the elements that are a match for the $(\alpha+\beta)$ first cells. If x and y are the number of known bits from below and above, resp., then the collision 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 find next if the remaining $k-\alpha-\beta$ cells are verified.

We say that two j-cell elements $(x_1,\ldots,x_j)$ and $(y_1,\ldots,y_j)$ of $\{0,1,-\}^j$ are *associated* if their colliding active cells have the same values. In the following $\boldsymbol{x}=(x_1,\ldots,x_k)$ will represent k-cell vectors in the middle of the permutation obtained in the forward direction and $\boldsymbol{y}=(y_1,\ldots,y_k)$ will represent k-cell vectors in the middle of the permutation obtained backwards. Then $\mathcal{L}_{\text{in}}$ will contain elements of type $\boldsymbol{x}$ and $\mathcal{L}_{\text{out}}$ elements of type $\boldsymbol{y}$. We define the probability for an element in $\mathcal{L}_{\text{out}}$ to have i active cells in its α first cells as $P_{[k,\alpha,y]}(i)$, and the probability for an element in $\mathcal{L}_{\text{in}}$ to have i active cells in its next β cells as $P_{[k,\beta,x]}(i)$.

First we will build the 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 $(x_1^A,\ldots,x_\alpha^A)\in\{0,1,-\}^\alpha$ and $(y_1^A,\ldots,y_\alpha^A)$ being associated to $(x_1^A,\ldots,x_\alpha^A)$. 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 $\mathcal{L}_B$, of all the elements of the form $(x_1^B,\ldots,x_\beta^B,y_1^B,\ldots,y_\beta^B)$ with $(x_1^B,\ldots,x_\beta^B)\in\{0,1,-\}^\beta$ and $(y_1^B,\ldots,y_\beta^B)$ being associated to $(x_1^B,\ldots,x_\beta^B)$. 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 $\mathcal{L}'_B$, 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 $\mathcal{L}_{\text{in}}$ 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 $(y_1^B,\ldots,y_\beta^B,x_1,\ldots,x_k)$ indexed[5] by $(y_1^B\ldots,y_\beta^B,x_1,\ldots,x_\alpha)$. The size of $\mathcal{L}'_B$ is:

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

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

and the cots of building this list is $< (|\mathcal{L}'_B| + 3^\beta)$, where 3^β captures the cases where no element in $\mathcal{L}_{\text{in}}$ is associated to the element in $\mathcal{L}_B$ and is normally negligible.

Next, for each element $(x_1^A, \ldots, x_\alpha^A, y_1^A, \ldots, y_\alpha^A)$ in $\mathcal{L}_A$ we consider the $2^{\ell_{\text{out}}} \frac{P_{[k,\alpha,y]}(i)}{2^i \binom{\alpha}{i}}$ elements $\boldsymbol{y}$ from $\mathcal{L}_{\text{out}}$ such that $(y_1, \ldots, y_\alpha) = (y_1^A, \ldots, y_\alpha^A)$ and we check in $\mathcal{L}'_B$ if elements indexed by $(y_1^B \ldots, y_\beta^B, x_1, \ldots, x_\alpha) = (y_{\alpha+1}, \ldots, y_{\alpha+\beta}, x_1^A, \ldots, x_\alpha^A)$ exist, that is if there is one index $(y_{\alpha+1}, \ldots, y_{\alpha+\beta}, x_1^A, \ldots, x_\alpha^A)$. 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 change 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:

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

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 known part i.e. the challenge. The response to the challenge is 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 $Y = (V_c, V_t) \oplus X$, $X = Q_U(C\|U)$, and V_t is a m-bit size known vector, 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 will suppose that y among the m bits of the lower part of X are known. As represented in Fig. 1, this means that y bits at the same positions of Y are also known (as V_t is known as well).

Now, we can apply the meet-in-the-middle technique described in the previous section which will allow us to recover the key. We have computed the optimal parameters for the different versions of ARMADILLO2, with different trade-offs. The results appear in Table 2.

For each version of ARMADILLO2 presented in Table 2, the first line corresponds to the size of lists $\mathcal{L}_{\text{in}}$ and $\mathcal{L}_{\text{out}}$ for which the time complexity is the smallest. The second line corresponds to the best parameters when the memory complexity is close 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	Time compl.	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. $\log_2$ of the complexities of the meet-in-the-middle key recovery attack for ARMADILLO2 in the FIL-MAC application (different trade-off)

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 stated in [1, Sc. 6] that a possible parameter choice is $r = 1$ and $t = k$. If we want to use it as a stream cipher, the secret key is C. The keystream is composed of k-bit frames and U is the index of the output frame, that is a public value.

In this setting, we can perform a similar attack as the one described in the previous section, but with different parameters. Indeed, we know $y = m + \ell_{\text{out}}$ bits of the output of Q_X. Complexities of the key recovery attack are then lower, as the number of known bits in the output of Q_X is bigger.

In general, the best time complexity will be obtained when $\ell_{\text{in}} = \ell_{\text{out}}$, as the number of known bits in 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 the first one.

We can see the best complexities for this attack in table 3. Other time-memory trade-offs would be possible as in the previous section. In order to have an idea of the different time-memory trade-offs, we have also computed the best parameters for a memory complexity of 2^{45}.

5 (Second) Preimage Attack on the Hashing Applications

We now consider the hashing mode. We recall that the hash function built with ARMADILLO2 as a compression function uses a strengthened Merkle-DamgArd mode, where the padding includes the message length. In this case C represents the input chaining value, U the message block 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.

Version	c	m	ℓ_{out}	ℓ_{in}	α	β	Time compl.	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. $\log_2$ of the best time complexity and the corresponding memory complexity of the key recovery attack for ARMADILLO2 in the PRNG mode.

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 compression function is invertible with a complexity 2^q, given an output V_c and a message block. In the preimage case, we choose and fix a length ℓ (number of blocks) for the preimage (in the second preimage attack, we can just 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}, which corresponds to the padding block including the message length. This step costs 2^q in a preimage scenario and 1 in a second preimage one. We get the state ARMADILLO2$^{-1}(h, M_{\text{pad}}) = S'$.
- From state S', we can invert the compression function for 2^b different blocks of messages M_b, obtaining 2^b different intermediate states ARMADILLO2$^{-1}(S', M_b) = S''$

In the forward direction: From the initial chaining value, we insert 2^a messages $\mathcal{M} = M_1\|M_2\|\ldots\|M_{\ell-2}$ of length $(\ell-2)$ blocks, 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 the 2^a states S and the 2^b states S'', we have obtained a (second) preimage that is $\mathcal{M}\|M_b\|M_{\text{pad}}$.

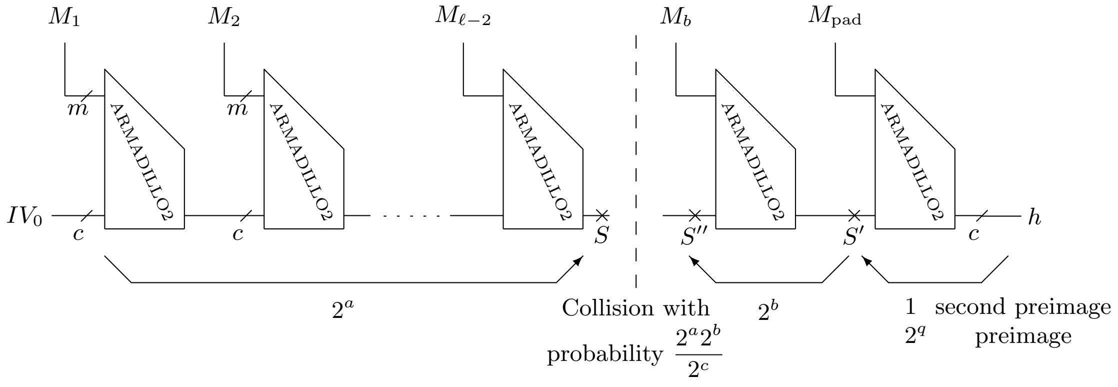

Fig. 2. Representation of the meet-in-the-middle (second) preimage attack.

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 considered 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 block of message, as the inversion 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 that put most of the m bits from U among the c most significant bits of X. This will result on better attacks, as the bits in Y known from U do not cost anything and this allows 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 that verify 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) while letting enough message blocks available 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 of $2^{-44.171}$. This leaves us $2^{48-44.171} = 2^{3.829}$ message blocks that allow us to know $y = \min(\ell_{\text{out}} + 48, c)$ 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 type of message blocks with $n = 47$, that allow us to know $\min(\ell_{\text{out}} + 47, c)$, 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	Time compl.	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. $\log_2$ of the complexities for inverting the compression function.

In this particular case the time complexity for $n = 48$ and for $n = 47$ is the same one, 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 $a = c - b_m$.

The best 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.

			Best time		Time-memory trade-off	
Version	c	m	Time	Memory	Time	Memory
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. $\log_2$ of the complexities of the (second) preimages attacks on ARMADILLO2.

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).

Table 6 shows that the average of the implementation results is very close to the theoretical estimations. In fact, all measurements were extremely close to the average. Thus, the implementation seems to confirm the above results.

		c	m	ℓ_{out}	ℓ_{in}	α	β	y	$\log_2 \lvert\mathcal{L}'_A\rvert$	$\log_2 \lvert\mathcal{L}'_B\rvert$	$\log_2(M)$
FIL-MAC	Impl.	30	18	12	18	8	6	14	23.477	25.002	27.537
	Theory	30	18	12	18	8	6	14	23.475	25.003	27.538
PRNG	Impl.	30	18	14	16	7	6	32	22.530	23.160	24.728
	Theory	30	18	14	16	7	6	32	22.530	23.160	24.735

Table 6. Key recovery attacks against a scaled version of ARMADILLO2 in the FIL-MAC and PRNG modes, where $M \equiv$ the number of partial matches.

7 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.

We believe that our attack could be countered if the permutation Q_C was used instead of the permutation Q_U, i.e. a permutation taking C as parameter instead of U.

Besides the results on ARMADILLO2, we have shown the first example of how to extend the application of the parallel matching technique in the case where the lists to merge do not have random elements.

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: CHES. 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: SAC. 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. J. Cryptology 15(1), 61–72 (2002)
4. Naya-Plasencia, M.: Scrutinizing rebound attacks: new algorithms for improving the complexities. Tech. Rep. Report 2010/607, Cryptology ePrint Archive (2010), http://eprint.iacr.org/2010/607.pdf