Practical Cryptanalysis of ARMADILLO2

María Naya-Plasencia[1,⋆] and Thomas Peyrin[2,⋆⋆]

[1] University of Versailles, France
maria.naya-plasencia@prism.uvsq.fr

[2] Division of Mathematical Sciences, School of Physical and Mathematical Sciences, Nanyang Technological University, Singapore
thomas.peyrin@gmail.com

Abstract. The ARMADILLO2 primitive is a very innovative hardware-oriented multi-purpose design published at CHES 2010 and based on data-dependent bit transpositions. In this paper, we first show a very unpleasant property of the internal permutation that allows for example to obtain a cheap distinguisher on ARMADILLO2 when instantiated as a stream-cipher. Then, we exploit the very weak diffusion properties of the internal permutation when the attacker can control the Hamming weight of the input values, leading to a practical free-start collision attack on the ARMADILLO2 compression function. Moreover, we describe a new attack so-called local-linearization that seems to be very efficient on data-dependent bit transpositions designs and we obtain a practical semi-free-start collision attack on the ARMADILLO2 hash function. Finally, we provide a related-key recovery attack when ARMADILLO2 is instantiated as a stream cipher. All collision attacks have been verified experimentally, they require negligible memory and a very small number of computations (less than one second on an average computer), even for the high security versions of the scheme.

Keywords: ARMADILLO2, hash function, stream-cipher, MAC, cryptanalysis, collision.

1 Introduction

Hash functions are among the most important and widely spread primitives in cryptography. Informally a hash function H is a function that takes an arbitrarily long message as input and outputs a fixed-length hash value of size n bits. The classical security requirements for such a function are collision resistance and (second)-preimage resistance. Namely, it should be impossible for an adversary to find a collision (two different messages that lead to the same hash value) in less than $2^{n/2}$ hash computations, or a (second)-preimage (a message hashing to a given challenge) in less than 2^n hash computations. In general, a hash function H

⋆ Supported by the French Agence Nationale de la Recherche through the SAPHIR2 project under Contract ANR-08-VERS-014.

⋆⋆ Supported by the Lee Kuan Yew Postdoctoral Fellowship 2011 and the Singapore National Research Foundation Fellowship 2012.

is built from an iterative use of a n-bit output compression function h in a Merkle-Damgård-like operating mode [6,4]. The compression function takes a chaining variable CV (fixed to an initial value IV at the beginning) and a message block M as inputs and in order to allow security proofs on the operating mode, one requires the same security properties as a hash function, namely collision and (second)-preimage resistance. However, the compression function allows several flavors of security properties depending on how well the attacker can control the chaining variable:

- free-start collision: the attacker fully controls the chaining variable, i.e. both its value and difference
- semi-free-start collision: the attacker control partially the chaining variable, i.e. only its value, and the difference is null
- collision: the attacker does not control the chaining variable, the value is defined by the IV and the difference is null

For all three flavors, it should be impossible for an adversary to find a collision in less than $2^{n/2}$ compression function computations. Note that free-start collision is required as necessary assumption regarding the compression function in the Merkle-Damgård-like security proofs. Moreover, a semi-free-start collision means there exists initial values IV for which it is possible to find collisions for the hash function. Therefore, both these two notions are very important and should be verified for a secure compression function.

ARMADILLO2 [2] is a very novel primitive dedicated to hardware, defining a FIL-MAC, a stream cipher and a hash function. Originally, two versions were proposed, ARMADILLO and ARMADILLO2, the later being the recommended one. A key recovery attack on ARMADILLO was rapidly published by a subset of the designers [9]. ARMADILLO2 remained unbroken until Abdelraheem *et al.* [1] found a meet-in-the-middle technique that allows to invert the ARMADILLO2 main function. This cryptanalysis eventually led to a key recovery attack on the FIL-MAC and the stream cipher, and a (second)-preimage attack on the hash function. However, while being the first weakness published on ARMADILLO2, this work is an improved meet-in-the-middle technique, therefore requiring a lot of computations and memory, often close to the generic complexity. For example, the preimage attack on the 256-bit output hash function requires either 2^{208} computations and 2^{205} memory or 2^{249} computations and 2^{45} memory. With its data-dependent bit transpositions and original compression function construction, ARMADILLO2 is clearly not following the classical design trends for symmetric-key primitives (for example RC5 [7] and RC6 [8] use data-dependent rotations, while IDEA [5] use data-dependent multiplication). As a consequence, it would be interesting to look at this proposal without necessarily relying on known cryptanalysis techniques.

Our Contributions. In this paper, we first observe the very unpleasant property that the parity bit is preserved through all ARMADILLO2 internal permutations. This allows us for example to derive a very cheap distinguisher for the stream-cipher. Then, we analyze the differential diffusion of the permutations and we provide practical free-start collision attacks for all versions of the compression function of ARMADILLO2. We extend our results by introducing a new technique, the *local linearization*, that seems very efficient against data-dependent bit transpositions. This method led us to practical semi-free-start collision attacks for all versions of ARMADILLO2. All attacks require very few computations (at most $2^{10.2}$ operations for 256-bit output version) and negligible memory. Moreover, our implementations validate our techniques and we provide collision examples. Finally, we provide a related-key recovery attack when ARMADILLO2 is instantiated as a stream cipher.

2 The ARMADILLO2 Function

We let $X[i]$ denote the i-th bit of a word X. 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 $k = c + m$, where $||$ denotes the concatenation operation. The internal ARMADILLO2 function transforms the vector (C, U) into (V_c, V_t) as described in Figure 1, $(V_c, V_t) =$ ARMADILLO2(C, U). The internal ARMADILLO2 function relies on a parameterized permutation on k bits Q, instantiated by Q_U and Q_X, where U is a m-bit parameter and X is a k-bit parameter.

Let σ_0 and σ_1 be two fixed bitwise permutations of size k. In [2], the permutations are not specifically defined but some criteria they should fulfill is given.

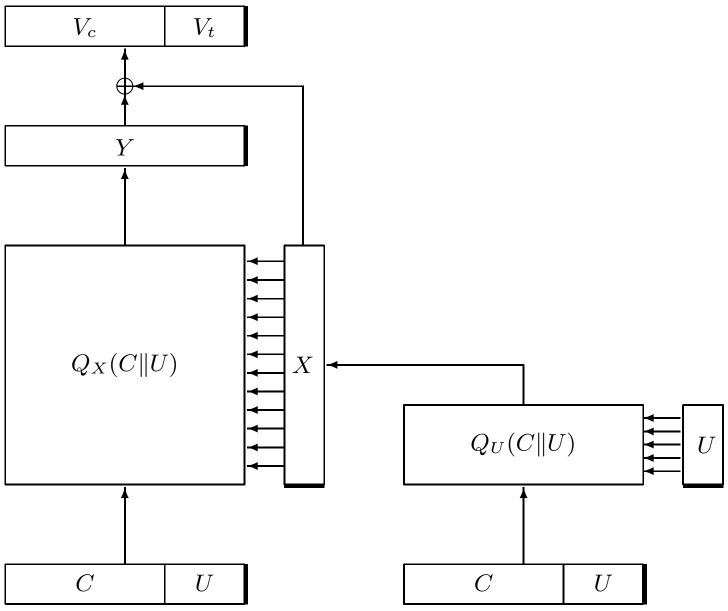

Fig. 1. The internal function of ARMADILLO2. The thick line at the side of a register represents the least significant bit.

We denote by cst a constant of size k defined by alternating 0's and 1's, i.e. : $cst = 1010 \cdots 10$. Using these notations, we can specify Q which is used twice in the internal ARMADILLO2 function. Let A be the a-bit parameter and B be the k-bit input of Q, the parameterized permutation Q_A can be divided into $a = |A|$ simple steps. The i-th step of Q_A (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-bit of A is 0 we apply σ_0 to B, otherwise we apply σ_1.
- a **constant addition** (bitwise XOR) of cst: $B \leftarrow B \oplus cst$.

The internal ARMADILLO2 function first computes $X = Q_U(C||U)$, then $Y = Q_X(C||U)$, and finally outputs $(V_c, V_t) = Y \oplus X$.

Using this internal primitive, ARMADILLO2 builds a FIL-MAC, a stream-cipher and a hash function:

- **Stream-cipher**: the secret key is inserted in the C register and the output sequence is obtained by taking the k bits of the output (V_c, V_t) after one iteration. The keystream is composed of k-bit frames indexed by U (which is a public value).
- **Hash function**: it uses a strengthened Merkle-Damgård construction, where V_c represents the output of the compression function (i.e. the next chaining value or the hash digest), U is the incoming message block and C is the incoming chaining variable.
- **FIL-MAC**: the secret key is inserted in the C register and the challenge, considered known by the attacker, is inserted in the U register. The response to the challenge is the m-bit output V_t.

Five different sets of register sizes (k, c, m) are provided, namely $(128, 80, 48)$, $(192, 128, 64)$, $(240, 160, 80)$, $(288, 192, 96)$ and $(384, 256, 128)$.

3 First Tools

We denote $\texttt{HAM}(X)$ the Hamming weight of the word X. We recall from [1] that for two random k-bit words A and B of Hamming weight a and b respectively, the probability that $\texttt{HAM}(A \wedge B) = i$ (where $\wedge$ stands for the bitwise AND function) is given by the formula

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

Moreover, we would like to deduce from it the probability that $\texttt{HAM}(A \oplus B) = i$ (where $\oplus$ stands for the bitwise XOR function) for two randomly chosen k-bit words A and B of Hamming weight a and b respectively. We remark that $\texttt{HAM}(A \oplus B) = a + b - 2 \cdot \texttt{HAM}(A \wedge B)$ and therefore the probability that $\texttt{HAM}(A \oplus B) = i$ is given by the formula.

$$P_{\texttt{xor}}(k,a,b,i) = \begin{cases} P_{\texttt{and}}(k,a,b,\frac{a+b-i}{2}) & \text{for } (a+b-i) \text{ even} \\ 0 & \text{for } (a+b-i) \text{ odd} \end{cases}$$

Since they have not been specified in the original ARMADILLO2 document, in the following we assume that σ_0 and σ_1 are randomly chosen bit permutations.

4 Parity Preservation

We call the parity bit of an a-bit word A the bit value $\bigoplus_{i=0}^{a-1} A[i]$. Regardless of the parameter A of the internal permutation Q_A, we have that **the parity of the input is always maintained through the permutation**. This can be easily verified by remarking that Q_A is composed of several identical rounds, all satisfying this property. Indeed, one round is composed of a bit permutation (which fully maintains the Hamming weight) and an XOR of the internal state with the constant $cst = \texttt{1010...10}$. This constant being always the same during the whole ARMADILLO2 computation and its parity being even, the parity of the internal state remains the same after application of the XOR. Note that even if this constant was changed during the rounds, the attacker would only have to compute the parity of the XOR of all constants to be able to tell if the parity bit will be maintained or negated. This property is moreover maintained whatever number of rounds is applied in the permutations, thus the attack proposed in this section is independent of the number of rounds.

Distinguisher for the Stream Cipher Mode. We can exploit the previous property to build a cheap distinguisher on ARMADILLO2 when used as a stream-cipher. In the attack model, the whole output of the function is assumed to be known as it is a frame of the keystream. This output is generated by a XOR of internal states X and Y. Since permutations Q_U and Q_X will maintain the parity, their respective outputs X and Y will both have the same parity as $(C||U)$. As a consequence, the output of the function $X \oplus Y$ always has an even parity. For a random sequence, this will only happen with probability 1/2, as for ARMADILLO2 this happens with probability 1. In other words, the entropy of the ARMADILLO2 function output is reduced by one bit.

5 Controlled Diffusion: Practical Free-Start Collision Attack

In this section, we show how an attacker can control the bit difference diffusion in ARMADILLO2 function by using the available inputs. This leads to a very cheap free-start collision attack against the compression function.

5.1 General Description

Assume that we insert a single bit difference in C, that is $\mathtt{HAM}(\Delta C) = 1$, and no difference in U that is $\Delta U = 0$. We can use c distinct ΔC, one for each active bit position. The attack is depicted in Figure 2.

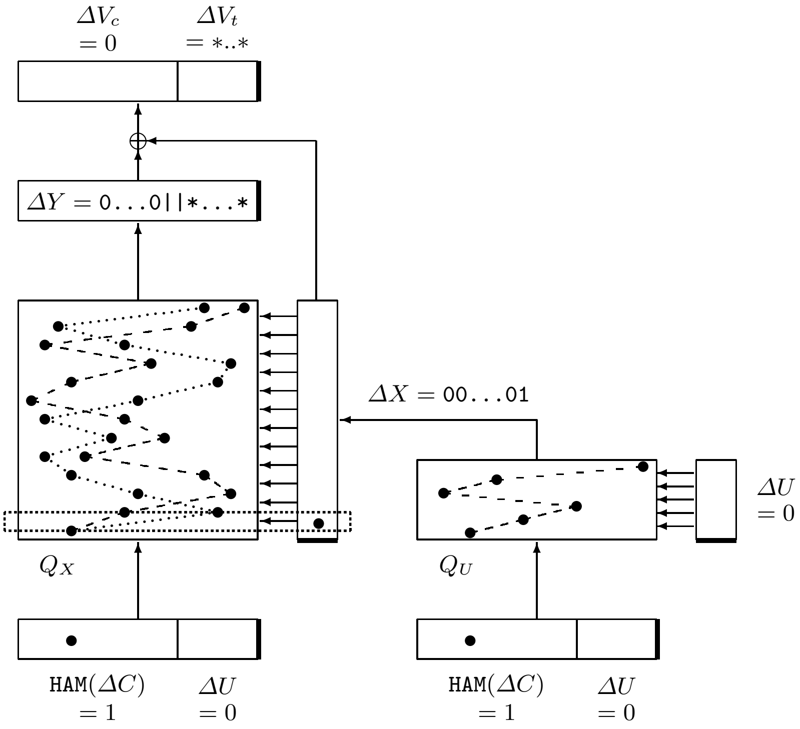

Fig. 2. A schematic view of the free-start collision attack on ARMADILLO2. The thick line at the side of a register represents the least significant bit and black circles stand for bit differences. The dashed box indicates the first round of Q_X, which contains a difference on its corresponding parameter input bit.

Difference Propagation in Q_U. Since we have no difference in U, the permutation Q_U always remains the same. We only have to study the propagation of the bit difference in C through Q_U. Note that one round of the internal permutation Q_U provides no difference diffusion since it is only composed of a bit permutation and a constant addition. Therefore, the single bit difference in C will be just transfered to some random bit position in X at the end of Q_U and we have $\mathtt{HAM}(\Delta X) = 1$. We would like the single bit difference in X to be positioned in bit 0, i.e. $\Delta X = \mathtt{00...01}$ (this will later allow us to use the freedom degrees efficiently). For a randomly chosen value of U and C, this happens with probability

$$P_X = \frac{1}{k}.$$

Difference Propagation in Q_X. Since we have a single difference on the first bit of X (corresponding to the first step of Q_X), the permutation Q_X remains the same except for the first step where we switch from bit permutation σ_0 to σ_1 or from σ_1 to σ_0. We denote by $P_{step}(in, out)$ the probability that *in* active bits are mapped to *out* active bits through a step of data-dependent permutation with a difference (i.e. σ_0 and σ_1 are swapped). Assume for the moment that after this first step, only b bits are active in the internal state. This happens with probability $P_{step}(1, b)$. Since the next rounds of the internal permutation Q_X provide no difference diffusion, we end up in Y with b active bits randomly distributed. We need to ensure that all the b active bits remaining in Y will go to the m-bit V_t part of the k-bit output, so that all differences will be truncated and we eventually obtain a collision on the output of the compression function. For $b \leq m$, this happens with probability

$$P_{out}(b) = P_{\mathtt{and}}(k, m, b, b) = \frac{\binom{b}{b}\binom{k-b}{m-b}}{\binom{k}{m}} = \prod_{i=0}^{i=b-1} \frac{m-i}{k-i}.$$

During the feed-forward after Q_X the single active bit of X is already on the V_t part of the output. Overall the probability of obtaining a compression function collision for randomly chosen U and C values is:

$$P_{collision} = P_X \cdot \sum_{i=1}^{i=m} P_{step}(1, i) \cdot P_{out}(i).$$

the sum stopping at m because when $i > m$, we trivially have $P_{out}(i) = 0$. At this point our problem is that in order for the probability $P_{out}(i)$ to be high enough, we need the number i of active bits to be small. On the other side, if i is small, $P_{step}(1, i)$ will be very low (we do not explain how to compute $P_{step}(1, i)$ here as we will study a slightly more detailed problem in the next section). However, in this scenario we only considered an attacker that randomly chooses the value of U and C and the bit difference position in C, but we can do much better by using the available degrees of freedom efficiently.

5.2 Using the Freedom Degrees

First, note that the event related to the probability P_X only depends on the position of the bit difference in C and on the value of U. We can therefore attack Q_U in a first phase (by fixing the position of the bit difference in C and the value of U), and then independently attack Q_X by choosing the value of C.

Handling Q_U. We will see later that we would like C and U values to have an extremely low or extremely high Hamming weight. Therefore, we fix $\Delta X =$ `00...01` and test with the two values $U =$ `00..00` and $U =$ `11..11` how the bit difference will propagate through Q_U^{-1} (note that we are dealing with the inverse of Q_U, thus attacking backwards from ΔX). For each try, we have a probability

$P_{\texttt{and}}(k, c, 1, 1) = c/k$ that the single bit difference is mapped to the C part of the input. Since for all ARMADILLO2 versions we have $2c/k > 1$, we expect at least one of the two U candidates to satisfy $\Delta X = \texttt{00...01}$, $\texttt{HAM}(\Delta C) = 1$ and $\texttt{HAM}(\Delta U) = 0$. Overall, this phase costs us only 2 operations. We assume without loss of generality that the selected candidate has value $U = \texttt{00..00}$.

Handling Q_X. At the present time, everything is fixed except the value of C and we have $\Delta X = \texttt{00...01}$ and $U = \texttt{00..00}$. We now describe a simple criteria in order to choose the values of C such that the first round probability $P_{step}(1, i)$ in Q_X is high, even for small i. As an example, let's assume that $C = 0$, that is $\texttt{HAM}(C||U) = 0$. In that case, we trivially have that $P_{step}(1, 1) = 1$ (and $P_{step}(1, i) = 0$ for all other i) since changing the bit positions of the word $\texttt{00..00}$ (switching from σ_0 to σ_1 or from σ_1 to σ_0) will not have any effect at all and the single bit difference in C will just be placed to some random bit position. Similarly, with a single one-bit in C, that is $\texttt{HAM}(C||U) = 1$, we have that $P_{step}(1, 1) = \frac{1}{128} + \frac{2 \cdot 127}{128^2}$ and $P_{step}(1, 3) = \frac{127 \cdot 126}{128^2}$ (and $P_{step}(i) = 0$ for all other i). More generally, we have to compute the probability $P_{step}(1, b, hw)$ which corresponds to the probability $P_{step}(1, b)$ knowing that the input word hamming weight is hw. This can be modeled as follows: choose two random k-bit words x and y both with Hamming weight hw (they represent $\sigma_0(C||U)$ and $\sigma_1(C||U)$) and compute $z = x \oplus y \oplus 1$ (the 1 represents the single bit difference in C). Then $P_{step}(1, b, hw)$ is the probability that $\texttt{HAM}(z) = b$ (note that $\texttt{HAM}(z)$ is always odd thus we have $P_{step}(1, 2i, hw) = 0$ for all i) and we have:

$$P_{step}(1, b, hw) = \frac{hw}{c} \cdot P_{\texttt{xor}}(k, hw, hw - 1, b) + \frac{c - hw}{c} \cdot P_{\texttt{xor}}(k, hw, hw + 1, b).$$

The complexity for handling Q_X is finally

$$Comp = \frac{1}{\sum_{i=1}^{i=m} P_{step}(1, i, hw) \cdot P_{out}(i)}.$$

5.3 Complexity Results

The number C of candidate values we can generate with Hamming weight hw is $\binom{c}{hw}$ and in order to have a good chance to find a collision after Q_X with this amount, we need to ensure that

$$\binom{c}{hw} \geq 1 / \sum_{i=1}^{i=m} P_{step}(1, i, hw) \cdot P_{out}(i).$$

One can check that in order to minimize the complexity $Comp$, the dominant factor of the sum is when i is small. Then, for i small, $P_{step}(1, i, hw)$ is higher when hw is close to 0 or close to k, in other words the input should have very low or very high Hamming weight. Since we previously chose $U = \texttt{00..00}$ our goal is to find for each ARMADILLO2 versions the smallest hw value hw_{min} that ensures enough C candidate values to handle the collision probability in Q_X (but the same reasoning is possible with $U = \texttt{11..11}$ and the biggest hw value hw_{max}). Overall, the full attack runs in $2 + Comp$ operations (i.e. compression function calls) and negligible memory in order to find a free-start collision for the ARMADILLO2 compression function. We depict in Table 1 our results relative to all proposed versions of ARMADILLO2. This attack has been implemented and verified in practice for $k = 128$ and we give free-start collision examples in the Appendix.

Table 1. Summary of results for free-start collision attack on the different size variants of the ARMADILLO2 compression function. The number of C candidates must always be enough so as to handle the collision probability in Q_X.

scheme parameters				**attack parameters**			
k	c	m	generic complexity	hw_{min}	nber of C candidates	collision prob. in Q_X	attack complexity
128	80	48	2^{40}	1	$2^{6.3}$	$2^{-4.1}$	$2^{7.5}$
192	128	64	2^{64}	1	2^{7}	$2^{-4.6}$	$2^{7.8}$
240	160	80	2^{80}	1	$2^{7.3}$	$2^{-4.7}$	$2^{8.1}$
288	192	96	2^{96}	1	$2^{7.6}$	$2^{-4.7}$	$2^{8.3}$
384	256	128	2^{128}	1	2^{8}	$2^{-4.8}$	$2^{8.7}$

6 Local Linearization: Practical Semi-free-Start Collision Attack

In this section, we show how one can obtain a semi-free-start collision attack (no difference on the input chaining variable) with a very low computational complexity for the ARMADILLO2 compression function.

6.1 General Description

The previous method only allows to add differences on the capacity part of the input, thus leading to free-start collision attacks. One can directly extend this technique to allow only differences in the message part of the input, but this only leads to semi-free-start collisions for randomly chosen bit permutations σ_0 and σ_1 with a not-so-high probability of success.

We would like to derive a semi-free-start collision attack that will output a result with very high probability. In order to achieve this goal we propose a new technique for data-dependent bit transposition ciphers, so-called **local linearization**: by guessing some part of the input we are able to render a few rounds of the internal permutation linear. Indeed, by knowing the g first bits of

U we completely determine the permutations applied during the first g rounds of Q_U. Therefore, for those g rounds the primitive Q_U only consists of known bit permutations and known constant additions. With this method we neutralize for the first g rounds the only non-linearity source: the fact that we don't know which bit permutation σ_0 or σ_1 is applied each round.

On a high-level view, our semi-free-start collision attack will force a collision on the X value at the output of Q_U thanks to the local linearization technique. This collision on X will ensure that the Q_X permutation will be the same for both inputs. Therefore, the difference Hamming weight on the input of Q_X will remain the same in the output. We then hope that those bit differences will be mapped in the truncated part of the output in order to eventually obtain the semi-free-start collision (no difference is feed-forwarded from X since we forced a collision on it). The attack is depicted in Figure 3.

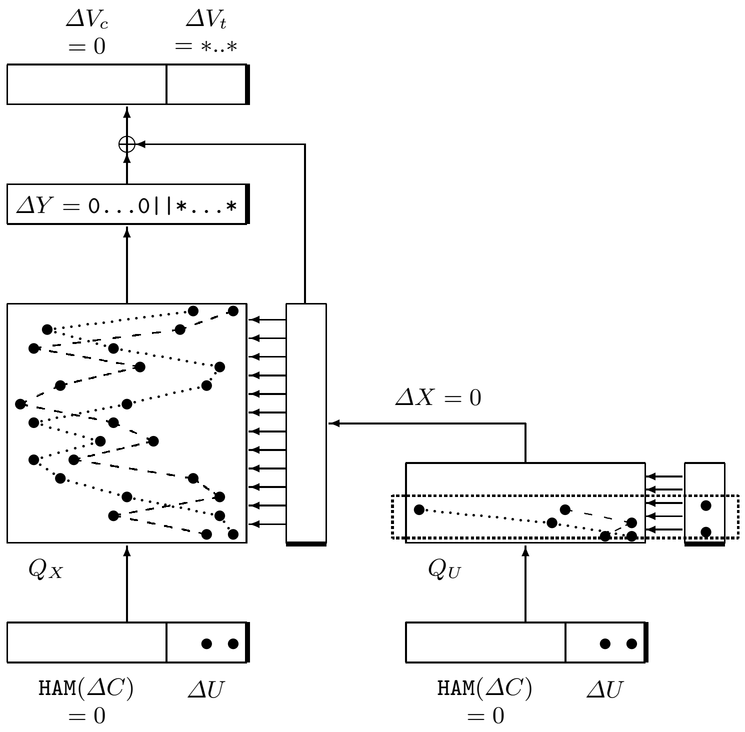

Fig. 3. A schematic view of the semi-free-start collision attack on ARMADILLO2. The thick line at the side of a register represents the least significant bit and black circles stand for bit differences. The dashed box indicates the linearized part.

During a first phase, the input will be divided into two parts: the fixed and the unfixed part. The fixed part $z \in \{0,1\}^g$ is composed of the g first bits of U and we choose random values for those g bits (so as to know the g first choices of σ_0 or σ_1). The unfixed part $w \in \{0,1\}^{k-g}$ is composed of the rest of the input bits and we will be set to a value later. We force the input difference to be contained in the fixed part and we denote it $\Delta z \in \{0,1\}^g$ (since we are looking for semi-free-start collisions we obviously have $g \leq m$, otherwise we would have a difference in the input chaining variable C). Let $I_1 = (C_1||U_1)$ (resp. $I_2 = (C_2||U_2)$) be the k-bit value of the first input (resp. second output), we have:

$$I_1 = (x||z) \text{ and } I_2 = (x||z \oplus \Delta z).$$

and our goal is to have the collision $X = Q_{U_1}(I_1) = Q_{U_2}(I_2)$.

Assume for the moment that this collision on X happens. Then the same permutation Q_X will be used for both inputs I_1 and I_2 on the right side of Figure 1. As a consequence, no additional bit difference will be introduced during the computation of Q_X, but the bit difference positions will be randomly moved. In order to obtain a semi-free-start collision on the output of the function, we need the $b = \mathtt{HAM}(\Delta z)$ active bits of the input to be mapped in the truncated part of the output through Q_X. As already explained in Section 5, this happens with probability

$$P_{out}(b) = P_{\mathsf{and}}(k, m, b, b) = \prod_{i=0}^{i=b-1} \frac{m-i}{k-i}.$$

6.2 Colliding on X

We need now to evaluate the probability of getting a collision on X. Note that for any round, if there is no difference on the bit choosing the permutation to apply σ_0 or σ_1, the bit differences at the input of this round will only have their position changed and cannot be erased. Therefore, if we want to obtain a collision on X, we need to obtain it at latest just after the last round of Q_U for which a difference is inserted on the side (in U). We consider from now on that the input difference Δz contains at least one active bit on its MSB, thus this last round is the g-th one.

We know the value of the g first bit of U, therefore we know exactly the permutation applied to I_1 and I_2 for the g first rounds of Q_U. For a collision after g rounds of Q_U, we want that

$$\begin{aligned} &\sigma_{U_1[g-1]}(\cdots(\sigma_{U_1[1]}(\sigma_{U_1[0]}(I_1) \oplus cst) \oplus cst)\cdots) \\ &= \sigma_{U_2[g-1]}(\cdots(\sigma_{U_2[1]}(\sigma_{U_2[0]}(I_2) \oplus cst) \oplus cst)\cdots) \end{aligned}$$

and since **all operations are linear**, this can be rewritten as

$$\rho(I_1) \oplus A = \rho'(I_2) \oplus B = \rho'(I_1 \oplus \Delta z) \oplus B = \rho'(I_1) \oplus \rho'(\Delta z) \oplus B$$

where

$$\begin{aligned} \rho &= \sigma_{U_1[g-1]} \circ \cdots \sigma_{U_1[1]} \circ \sigma_{U_1[0]} & A &= \sigma_{U_1[g-1]}(\cdots(\sigma_{U_1[1]}(cst) \oplus cst)\cdots) \\ \rho' &= \sigma_{U_2[g-1]} \circ \cdots \sigma_{U_2[1]} \circ \sigma_{U_2[0]} & B &= \sigma_{U_2[g-1]}(\cdots(\sigma_{U_2[1]}(cst) \oplus cst)\cdots). \end{aligned}$$

Finally, we end up with the equation

$$\rho(I_1) \oplus \rho'(I_1) = A \oplus B \oplus \rho'(\Delta z) \tag{1}$$

Since we know the value of the g first bit of U, we can compute the value of A and B. Moreover, assuming that we already chose a Δz, then the collision condition (1) can be rephrased as

$$I_1 \oplus \tau(I_1) = C$$

where $C = \rho^{-1}(A \oplus B \oplus \rho'(\Delta z))$ and $\tau = \rho^{-1} \circ \rho'$.

In order to study this system $\mathcal{S}$ of k bit equations, we model τ as a random bit permutation and C as a random k-bit word. Note that since this equation system is linear finding the potential solutions requires only a few operations, but we would like to know how many such systems we need to generate before finding a solution, i.e. a collision on X. Thus, our goal is now to deduce the probability that this system has at least one solution and what is the average number of expected solutions.

The structure of this equation system is very particular and the number of independent groups of bit equations is exactly the number of cycles of the bit permutation τ. More precisely, let CYCLE(τ) represent the number of cycles of the permutation τ and let S_i denote the set of bits belonging to the i-th cycle of τ.

Theorem 1. *The equation system $\mathcal{S} : I_1 \oplus \tau(I_1) = C$ admits a solution if and only if for every cycle set S_i of τ the parity of the sum of the corresponding C bit is null, that is*

$$\bigoplus_{p \in S_i} C[p] = 0.$$

If this system is solvable, then the number of solutions that can be generated is exactly equal to $2^{\mathtt{CYCLE}(\tau)}$.

The Idea of the Theorem is that when we want to find a solution for the system, we can start by fixing one bit a_0 to a random value. This bit is involved into two binary equations from $\mathcal{S}$. All equations having only two terms, one of the two equations directly links bit a_0 with say bit a_1, and we can deduce the value of a_1. The bit a_1 is in turn linked with bit a_2 through his second equation and we directly deduce the value of a_2. This chain of dependency will eventually cycle (the new bit deduced will be a_0 again) and will be validated if and only if the sum of the C bits of the equations visited is null (otherwise we encounter a inconsistency). This check is then performed for all cycles.

Proof. Since τ is a bit permutation, the equation system $\mathcal{S}$ can be represented as a collection of cycles, each cycle depicting the direct cyclical dependencies between some set of bits: if bit x and bit y are linked by one of the k equations, then they belong to the same cycle. The vertex weight between two members x and y of the cycle is the value $C[x]$.

If we fix the bit value of a member of a cycle S_i, then this determines entirely all the other bits of that cycle (according to the vertices values). Then, if the XOR of all the vertex weights is different from zero, we have a direct contradiction. A solution can only exist if all cycles present no internal contradiction.

Each cycle can have either zero or two solutions (the two solutions being their mutual complement). If every cycle has no contradiction, then there exists exactly $2^{\mathtt{CYCLE}(\tau)}$ distinct combinations of cycle solutions, each one leading to a distinct solution for the whole equation system $\mathcal{S}$. □

From Theorem 1, we directly deduce that the probability that the system admits a solution is equal to $2^{-\mathtt{CYCLE}(\tau)}$. The expected number of cycles for a randomly chosen permutation on k elements is $\log(k)$. Therefore, we have to try at least $2^{\log(k)}$ different equation systems before finding one admitting a solution. When one system admits a solution, we directly get $2^{\log(k)}$ solutions for free. Overall, the cost for finding one solution of the system is 1 on average (the average cost is the meaningful one here since we will have to find several inputs colliding on X during the whole attack).

6.3 Complexity Results

We now look for a solution such that the original guess of the g first bits of the input was right (with probability 2^{-g}) and such that the b bit differences in Q_X are mapped to the truncated part of the output (with probability $P_{out}(b)$). Overall, the total complexity of the semi-free-start collision attack is $2^g \cdot P_{out}^{-1}(b)$ with $b \leq g$. Minimizing g and b will minimize the overall complexity, but we need to ensure that we can go through enough equation systems in order to have a good chance to find a collision eventually. More precisely, we need

$$1/2 \cdot 2^g \cdot \binom{g}{b} \geq 2^g \cdot P_{out}^{-1}(b)$$

which can be rewritten as

$$\binom{g}{b} \geq 2 \cdot P_{out}^{-1}(b).$$

We depict in Table 2 our results relative to all proposed versions of ARMADILLO2. This attack has been implemented and verified in practice for $k = 128$ and we give semi-free-start collision examples in the Appendix.

7 Related-Key Recovery in Stream Cipher Mode

In this section we will present a related key attack that will allow us to recover all key bits in practical time when using ARMADILLO2 in the stream cipher mode. We will first present the main idea of this attack, and afterwards, we will give a more detailed analysis of the probabilities and complexities.

Table 2. Summary of results for semi-free-start collision attack on the different size variants of the ARMADILLO2 compression function

scheme parameters				attack parameters			
k	c	m	generic complexity	g	b	$P_{out}(b)$	time complexity
128	80	48	2^{40}	6	2	$2^{-2.9}$	$2^{8.9}$
192	128	64	2^{64}	7	2	$2^{-3.2}$	$2^{10.2}$
240	160	80	2^{80}	7	2	$2^{-3.2}$	$2^{10.2}$
288	192	96	2^{96}	7	2	$2^{-3.2}$	$2^{10.2}$
384	256	128	2^{128}	7	2	$2^{-3.2}$	$2^{10.2}$

7.1 Using Related-Keys for Recovering the Key

First of all, we consider a pair of related keys (K_1, K_2) that have one only bit of difference, that is $\mathtt{HAM}(K_1 \oplus K_2) = \mathtt{HAM}(\Delta_K) = 1$. Our analysis will work for any bit difference position d amongst all the bits of the key. Note that we expect a pair of keys valid for performing the related-key attack to appear after using about $(2^k/k)^{1/2}$ keys.

Let us consider a value of U for generating k bits of key-stream with each of both keys K_1 and K_2. We use the index i for the intermediate states generated from the key K_i. We first make the following observations, important in order to understand the whole attack procedure:

- Since no difference is inserted in the U part (it is a public value) and since $\mathtt{HAM}(\Delta_K) = 1$, we have $\mathtt{HAM}(X_1 \oplus X_2) = 1$. Let e be the bit position of this difference in X.
- The first $(e - 1)$ intermediate states of Q_X will also have a difference of Hamming weight 1.

We assume that the attacker can choose the values of U. In this case, we can make the bit difference in the key to go from position d to any wanted position e in X through Q_U. We expect $2^m/k$ distinct values of U that make the bit difference go from position d to e for $e \in [0, k-1]$. We denote by U_e each one of these k subgroups of U values.

The output of the function $(V_c, V_t) = X \oplus Y$ is known to the attacker, but concerning X he only knows the m bits of the U part (since U is known, he can deduce directly where the bits coming from U and C will be eventually located in X). Thus, he can recover m bits from the outputs of Q_X, Y_1 and Y_2. If he could compute backward from Y_1 and Y_2 until the beginning of the e-th step of Q_X, the colliding positions of the bits known from Y_1 and from Y_2 will have the same values with maybe the exception of one, which would be the original single bit difference before the step e.

Our attack basically consists in choosing several values for U from U_e, for decreasing e values (starting from $e = k - 1$), that will gradually increase the number of key bits appearing in X after position e. Each time we will guess the value of the new key bits appearing and discard the guesses that will not lead to collisions on the bit values in the colliding positions just before step e when computing backward from Y_1 and Y_2 in Q_X. The complexity of this attack depends on the bit permutations σ_0 and σ_1, but in the next subsection we give a complexity analysis assuming that these permutations are randomly chosen.

7.2 Generic Complexity Estimation

We start at $e = k - 1$. First, we choose the value of i (denoted i_{max}), that maximizes the probability $P_{\mathtt{and}}(k, m, m, i)$ that we denote p_{max}. For instance, if we consider the smallest version of ARMADILLO2, where $k = 128$, $c = 80$ and $m = 48$, then we have $i_{max} = 18$ and the probability of obtaining 18 positions of known bits that collide is equal to $p_{max} = 2^{-2.72}$.

Amongst the values from U_{k-1}, we choose ${p_{max}}^{-1}$ random ones. Each of them is introduced in the ARMADILLO2 function parametrized with the keys K_1 and K_2. For each of the ${p_{max}}^{-1}$ pairs of values, we guess the bit at position $k - 1$ of X_1 and of X_2 (for example 1 and 0 respectively since there is a difference on this bit position) and we end up with $2 \cdot {p_{max}}^{-1}$ pairs. Then, we can undo the last round of Q_X for the known bits from Y_1 and Y_2. We consider that a guess passes the test if it verifies the conditions on the number of colliding values on the colliding bit positions. For one of these $2 \cdot {p_{max}}^{-1}$ pairs (in our example $(Q_1^{-1}(Y_1), Q_0^{-1}(Y_2)))$, the number of colliding bit positions will be i_{max}. When this is the case, if the guess on the bit of X_1 and X_2 was incorrect, we have a probability of $2^{-i_{max}+1}$ to pass the test, while we will pass it with probability one if the guess was correct. Finally, we have determined one bit of each key K_1 and K_2 with a complexity of $2 \cdot {p_{max}}^{-1}$, which in our example would be $2^{3.72}$.

We can continue the process by considering $e = k - 2$ and ${p_{max}}^{-1}$ values from U_{k-2} that have a key bit at position $k - 1$. Following the same method as before, we will recover one key bit, i.e. the one at position $k - 1$ in X when we have 18 colliding bits before the step $k - 1$ of Q_X. Let us remark here that in practice we do not have to wait for having a collision on 18 bits, but most of the time collisions on a different number of bits will also be enough for determining if a guess passes the test or not. We can repeat this step in order to obtain the biggest possible number of key bits and determining each bit will add at most a complexity of ${p_{max}}^{-1}$.

The next steps depend on the number of bits that we have already determined. All in all, we conjecture that when both bit permutations behave like random ones, the complexity will not exceed $2 \cdot c \cdot {p_{max}}^{-1}$.

8 Conclusion

We have presented some new and practical analysis of ARMADILLO2. Notably a free-start and semi-free-start collision attacks for the full ARMADILLO2 hash functions. Extending this work to real collisions (i.e. with a predefined IV) might be possible but it is not very appealing because it is likely that several message blocks are required (all versions have $c > m$) and therefore the task of the cryptanalyst would be quite complex to handle. ARMADILLO2 should not be used in any security application since our attacks have a very low complexity. This work and the local-linearization method is a first step in order to evaluate the security of data-dependent bit transpositions cryptographic designs.

Acknowledgements. The authors would like to thank the anonymous referees and the ARMADILLO2 team for their helpful comments.

References

1. Abdelraheem, M.A., Blondeau, C., Naya-Plasencia, M., Videau, M., Zenner, E.: Cryptanalysis of ARMADILLO2. In: Lee, D.H., Wang, X. (eds.) ASIACRYPT 2011. LNCS, vol. 7073, pp. 308–326. Springer, Heidelberg (2011)
2. Badel, S., Dağtekin, N., Nakahara Jr., J., Ouafi, K., Reffé, N., Sepehrdad, P., Sušil, P., Vaudenay, S.: ARMADILLO: A Multi-purpose Cryptographic Primitive Dedicated to Hardware. In: Mangard, S., Standaert, F.-X. (eds.) CHES 2010. LNCS, vol. 6225, pp. 398–412. Springer, Heidelberg (2010)
3. Brassard, G. (ed.): CRYPTO 1989. LNCS, vol. 435. Springer, Heidelberg (1990)
4. Damgård, I.: A Design Principle for Hash Functions. In: Brassard [3], pp. 416–427
5. Lai, X., Massey, J.L.: A Proposal for a New Block Encryption Standard. In: Damgård, I.B. (ed.) EUROCRYPT 1990. LNCS, vol. 473, pp. 389–404. Springer, Heidelberg (1991)
6. Merkle, R.C.: One Way Hash Functions and DES. In: Brassard [3], pp. 428–446
7. Rivest, R.L.: The RC5 Encryption Algorithm, pp. 86–96. Springer (1995)
8. Rivest, R.L., Robshaw, M.J.B., Yin, Y.L.: RC6 as the AES (2000)
9. Sepehrdad, P., Sušil, P., Vaudenay, S.: Fast Key Recovery Attack on ARMADILLO1 and Variants. In: Prouff, E. (ed.) CARDIS 2011. LNCS, vol. 7079, pp. 133–150. Springer, Heidelberg (2011)

A Implementation of the Collision Attacks for $k = 128$

We implemented all attacks for $k = 128$ and they require less than a second and negligible memory on an average computer (Intel Core2 Duo CPU @ 2.13 GHz) in order to find a collision. Since no specific σ_0 and σ_1 bit transpositions are defined for ARMADILLO2, we run the attack for many randomly chosen instances so as to ensure the soundness of our reasoning. We give here examples of (semi)-free-start collisions for ARMADILLO2 with a σ_0 and σ_1 bit transpositions instance that fulfill the criteria required in [2] for $k = 128$. Namely, we denote λ the second largest eigenvalue of the matrix $M = \frac{1}{4}(P_{\sigma_0} + P_{\sigma_0}^{128} + P_{\sigma_1} + P_{\sigma_1}^{128})$, then for the σ_0 and σ_1 instance found we have $\lambda = 0.87$. This means that there exists a distinguisher with advantage $\lambda^{256} = 2^{-51.4}$, while our attacks have much better advantage.

Free-Start Collision for ARMADILLO2 with $k = 128$, $c = 80$, $m = 48$:

$$\texttt{ARMADILLO2}(\texttt{ffffffffffffffffbfff}, \texttt{ffffffffffff}) =$$
$$\texttt{ARMADILLO2}(\texttt{fffffdffffffffffbfff}, \texttt{ffffffffffff}) =$$
$$\texttt{dfb0d8f2b763ce97f785}$$

Semi-Free-Start Collision for ARMADILLO2 with $k = 128$, $c = 80$, $m = 48$:

$$\texttt{ARMADILLO2}(\texttt{6bc8c848de5ff533cd6f}, \texttt{0850b04b82e2}) =$$
$$\texttt{ARMADILLO2}(\texttt{6bc8c848de5ff533cd6f}, \texttt{0850b04b82f0}) =$$
$$\texttt{26827e3d614d2fc75d64}$$

Bit Transpositions σ_0 and σ_1 Used:

$\sigma_0 =$ 62, 98, 14, 114, 36, 77, 55, 3, 28, 88, 29, 122, 57, 90, 66, 52, 44, 22, 95, 118, 69, 86, 35, 56, 58, 82, 18, 97, 78, 21, 85, 101, 19, 65, 10, 6, 116, 121, 70, 99, 61, 102, 4, 91, 39, 119, 79, 16, 84, 50, 113, 45, 93, 104, 73, 112, 8, 5, 51, 9, 105, 46, 64, 94, 41, 54, 127, 67, 106, 23, 63, 49, 123, 15, 60, 81, 96, 72, 110, 37, 30, 89, 7, 92, 2, 68, 40, 32, 53, 11, 71, 26, 103, 59, 109, 111, 38, 74, 20, 48, 24, 43, 126, 117, 13, 124, 31, 33, 100, 125, 87, 27, 83, 128, 12, 42, 80, 107, 108, 17, 25, 120, 76, 75, 115, 47, 1, 34

$\sigma_1 =$ 10, 60, 111, 78, 38, 57, 110, 75, 104, 56, 88, 79, 23, 99, 16, 22, 128, 94, 120, 24, 64, 3, 6, 55, 42, 51, 43, 82, 114, 89, 26, 35, 61, 73, 77, 36, 28, 21, 105, 15, 67, 70, 113, 65, 39, 80, 122, 31,101,100, 107, 124, 18, 46, 85, 19, 49, 14, 12, 71, 86, 68, 102, 91, 58, 95, 1, 53, 83, 125, 66, 98, 81, 44, 48, 59, 27, 9, 119, 40, 45, 74, 92, 112, 93, 69, 5, 108, 106, 115, 90, 13, 84, 126, 7, 109, 54, 127, 33, 121, 62, 87, 30, 29, 63, 2, 97, 116, 4, 47, 11, 8, 34, 96, 118, 72, 52, 103, 37, 25, 123, 50, 76, 17, 20, 41, 117, 32