Trumping the Elephant: Fast Side-Channel Key-Recovery Attack against Dumbo

Louis Vialar
louis@louisvialar.me

EPFL, Kudelski Security Research Team

Abstract. In this paper, we present an efficient side-channel key recovery attack against Dumbo, the 160-bit variant of NIST lightweight cryptography contest candidate Elephant. We use Correlation Power Analysis to attack the first round of the Spongent permutation during the absorption of the first block of associated data. The full attack runs in about a minute on a common laptop and only requires around 30 power traces to recover the entire secret key on an ARM Cortex-M4 microcontroller clocked at 7.4MHz. This is, to the best of our knoweledge, the first attack of this type presented against Elephant.

1 Introduction

Lightweight Cryptography (LWC) is an area of cryptography that studies and develops cryptographic primitives for resource-constrained devices, such as smart sensors, smart cards or RFID tags. These devices need to communicate in a secure fashion, and therefore need to use cryptographic protocols, however traditional protocols intended for desktop and mobile processors consume too much power and require too much memory for an embedded system.

In 2017, National Institute of Standards and Technology (NIST) published a report [13] on the state of the field, in particular regarding already-existing NIST cryptographic standards, which was followed in 2018 by a *Call for Algorithms* [7] for lightweight symmetric authenticated ciphers and hash functions, initiating the NIST Lightweight Cryptography project (NIST LWC), a standardization process for what will become the equivalent(s) of AES-GCM [9, 14] and SHA-3 [10] for resource-constrained devices. While the main objective for the submissions is to be efficient both in terms of timing, throughput and power consumption, resistance to side-channel analysis (SCA) is also an evaluation criterion. Indeed, if a smart device using a symmetric cipher is compromised (or if the user is the adversary, as with smart cards [18]), the secret key should remain inaccessible.

The symmetric authenticated cipher Elephant [19] is a finalist to this NIST standardization project. Elephant is based on Spongent [4], a lightweight hash function, and has a variant that is based on Keccak [3], the family of hash functions that led to SHA-3.

In this paper, we introduce a side-channel attack based on Correlation Power Analysis (CPA) [6] against the 160-bit variant of Elephant, based on Spongent and dubbed "Dumbo". The rest of this paper is structured as follows: in the first section, we present the relevant state of the art. In the second section, we introduce the Dumbo cipher and its underlying permutation, $\mathsf{Spongent}\text{-}\pi[160]$. Then, in the third section, we introduce our side-channel attack on Dumbo. Finally, in the fourth section, we present experimental results of our attack on an ARM Cortex-M4 microcontroller.

1.1 Notations used in this paper

In the rest of this paper, we define $\{0,1\}^n$ the set of n-bit bitstrings for some $n \in \mathbb{N}$ and $\{0,1\}^\star$ the set of bitstrings of arbitrary length. We denote the length of bitstring $X \in \{0,1\}^\star$ as $|X|$ and we denote with $X_0, X_1, \ldots, X_{l-1}$ the $l = \lceil \frac{|X|}{160} \rceil$ blocks of size 160 bits (20 bytes) of X, where the last block is appended with 0s. We designate by bit i (or i^{th} bit) the b^{th} rightmost bit of the B^{th} leftmost byte of bitstring X, where $i = 8 \cdot B + b$. We denote $X_{[i]}$ the i^{th} bit of X, and $X_{[a:b]}$ the substring of X that starts at bit a (inclusive) and ends at bit b (exclusive).

The concatenation of two bitstrings A and B is denoted as $A \| B$, their bitwise exclusive or is denoted as $A \oplus B$, and their bitwise and is denoted as $A \& B$. $X \ll i$ (resp. $X \lll i$) represent a shift (resp. rotation) of X to the left over i positions. $X \gg i$ and $X \ggg i$ represent the same operations to the right.

We denote with 0^n the bitstring made of n zeroes, and we denote the random sampling of a bitstring A of length n with $A \;\$\!\leftarrow \{0,1\}^n$.

2 Related work

Since the launch of the NIST standardization process, researchers have studied the implementation security of the candidates. For instance, CAESAR's "lightweight applications" winner and NIST finalist Ascon [8] was found to be vulnerable to two side-channel key recovery attacks by Ramezanpour et al. in [16]: a passive attack based on deep learning, called SCARL, and an active fault injection analysis attack. The hardware implementation of NIST finalist GIFT [2] has also been found to be vulnerable to a SCA attack by Hou et al. in [12], and the software implementation was also found to be vulnerable to a side-channel assisted differential cryptanalysis attack in [5], allowing key recovery in only 36 encryptions.

Side-channel attacks are not a mere theoretical threat, and can have real world consequences. In this respect, a SCA attack against AES-GCM was for example used by Ronen et al. in [17] to extract the secret keys used by Philips to sign the firmware of their smart light-bulbs. This enabled the attackers to build a worm that spreads from an infected object to another wirelessly. More recently, an electromagnetic CPA attack was used successfully to recover AES secret-keys in Apple's CoreCrypto by Haas et. al in [11].

To the best of our knowledge, our attack is the first attack of this type presented against Dumbo or any other Elephant instance.

3 The Dumbo NIST LWC Candidate

Dumbo is one of the three variants of NIST LWC candidate Elephant [19], and is the primary member of the submission. Elephant is a cryptographic mode of operation that uses a pseudo-random permutation P to build a symmetric cipher. It uses a 16-bytes (128 bits) secret key K for encryption and authentication and a 12-bytes (96 bits) nonce N. It is authenticated, with a 64-bits authentication tag T that authenticates both the ciphertext C and the optional associated data A.

In Dumbo, the underlying permutation is $\mathsf{Spongent}\text{-}\pi[160]$, the 80 rounds $\mathsf{Spongent}\text{-}\pi$ permutation of the Spongent lightweight hash function (introduced by Bogdanov et al. in [4]) with a 20-bytes (160 bits) long state. The two other variants of this cipher are Jumbo (using $\mathsf{Spongent}\text{-}\pi$ with 90 rounds and a 22-bytes long state, $\mathsf{Spongent}\text{-}\pi[176]$) and Delirium (replacing Spongent with a reduced version of the permutation used in Keccak [3], and using a 25-bytes long state). In the next sections, we will describe the design of Dumbo.

3.1 The $\mathsf{Spongent}\text{-}\pi[160]$ Permutation

$\mathsf{Spongent}\text{-}\pi[160]$ is a permutation described by Bogdanov et al. in [4]. In the rest of the paper, we denote as $P : \{0,1\}^{160} \to \{0,1\}^{160}$ the 80-round Spongent permutation defined in Algorithm 1, where:

— rev is a function that reverses the order of the bits in its input.

— $\mathsf{sBoxLayer}$ is a function that applies the $\{0,1\}^4 \to \{0,1\}^4$ substitution box defined in Table 1 to all nibbles of its input. In the reference implementation, it is applied on two nibbles at a time by using an extended $\{0,1\}^8 \to \{0,1\}^8$ look-up table.

— pLayer is a function that moves bit j from the input to bit $pL(j)$ in the output, such that

$$\mathsf{pL}(j) = \begin{cases} 40j \bmod 159 & \text{if } j < 159, \\ 159 & \text{if } j = 159. \end{cases}$$

— lfsr represents the computation of one cycle of the 7-bit LFSR defined by the primitive polynomial $p(x) = x^7 + x^6 + 1$. $\mathsf{lfsr}(c) = (c_{[0:6]} \lll 1) \| (c_{[6]} \oplus c_{[5]})$.

```
   Input: $X$, a 160-bits block of data
   Output: $X$, a 160-bits block of data updated by the permutation
1: $c \leftarrow$ 0b1110101
2: for i = 0, ..., 79 do
3:     $X \leftarrow X \oplus (0^{153} \| c) \oplus \mathsf{rev}(0^{153} \| c)$
4:     $X \leftarrow \mathsf{sBoxLayer}(X)$
5:     $X \leftarrow \mathsf{pLayer}(X)$
6:     $c \leftarrow \mathsf{lfsr}(c)$
   return X
```

Algorithm 1. The Spongent-π[160] permutation

X	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
SBox(X)	E	D	B	0	2	1	4	F	7	A	8	5	9	C	3	6

Table 1. the Spongent S-Box

Invertibility of *P* While Elephant does not require P to be invertible to encrypt plaintexts or to decrypt ciphertexts, we leverage its invertibility in our key-recovery attack.

We notice that all functions in P can be inverted. The inverse of the exclusive or operation is the exclusive or operation itself. The sBoxLayer is inverted by swapping the two lines in the substitution table presented before and reordering columns accordingly. The pLayer is inverted by building the bitstring in reverse order: instead of moving bit i to position $40i \bmod 159$, we move bit $40i \bmod 159$ to position i (for $i < 159$). Finally, the LFSR counter is inverted by computing its formula in reverse: the bit that was removed can be computed from the value of the other bits in the counter and of the bit that was generated from it.

Because of this, we can easily compute the inverse of P by starting the counter c to its value after 80 rounds (127), then by computing successively all the inverse operations of each round in reverse order. First, we apply the inverse LFSR on c, then we inverse the pLayer, the sBoxLayer, and finally we add the counter to the state.

3.2 The Dumbo Mode of Operation

In this section, we describe on a high level the process used to encrypt and authenticate a message in Dumbo. The decryption process is not precisely described but naturally follows from the encryption process.

Before encryption, the associated data and message lengths are not necessarily multiples of the block size. Therefore, the associated data and message are padded by adding a `0x01` byte, followed by as many `0x00` bytes as needed to complete the block. The empty message (i.e. M s.t. $|M| = 0$) is padded in the same way.

The encryption of the i^{th} message block M_i with nonce N is computed as follows ($i < l_M$):

$$C_i = \mathsf{mask}_K^{i,1} \oplus P(\mathsf{mask}_K^{i,1} \oplus (N \| 0^{64})) \oplus M_i$$

Similarly, decryption is computed using the same operation by swapping C_i and M_i.

The authentication tag is computed iteratively as follows:

1. The tag buffer T is initialized with the nonce concatenated with the first eight bytes of the associated data (A_0).
2. For each remaining 20-byte block of associated data A_i ($0 < i < l_A$), the tag buffer is updated as $T \leftarrow T \oplus \mathsf{mask}_K^{i,0} \oplus P(\mathsf{mask}_K^{i,0} \oplus A_i)$.
3. For each ciphertext block C_i, the tag buffer is updated as $T \leftarrow T \oplus \mathsf{mask}_K^{i,2} \oplus P(\mathsf{mask}_K^{i,2} \oplus C_i)$.
4. The tag buffer is updated by computing $T \leftarrow \mathsf{mask}_K^{0,0} \oplus P(T \oplus \mathsf{mask}_K^{0,0})$.
5. The first eight bytes of the tag buffer T are returned as the tag.

The entire encryption and authentication procedure is depicted in Figure 1.

We can notice that the key doesn't appear directly in this encryption procedure: it only appears as a parameter to the masking function $\mathsf{mask}_K^{a,b}$. The $\mathsf{mask}_K^{a,b}$ function will be described in the next sub-section.

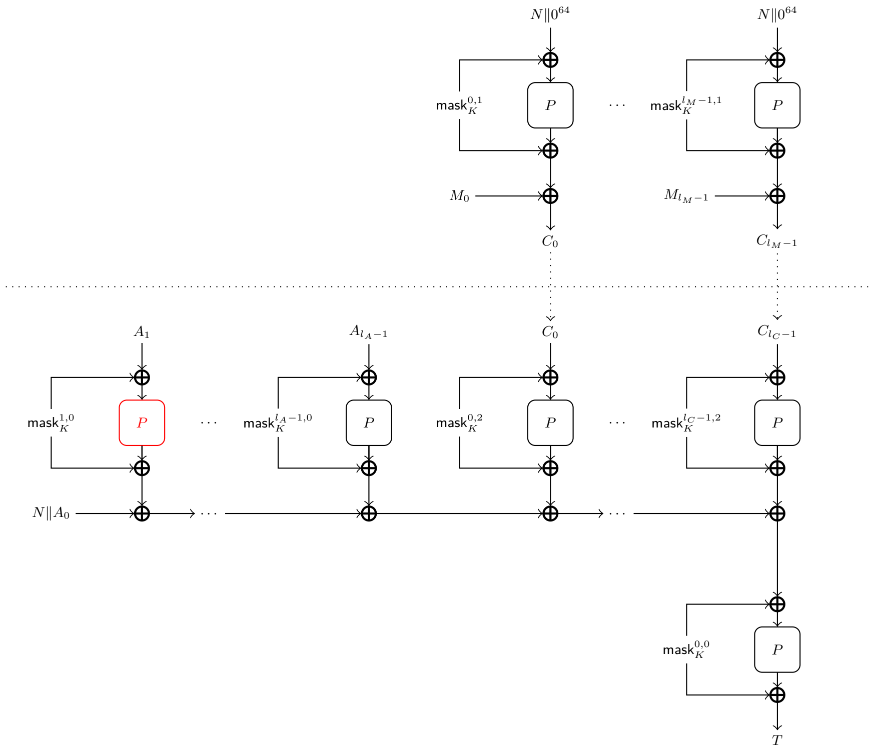

Fig. 1. Sketch of the Encryption and Authentication procedure in Dumbo, with the attack point highlighted

The Masking Functions The $\mathsf{mask}_K^{a,b}$ functions are defined for $b = \{0, 1, 2\}$ as follows:

— $\mathsf{mask}_K^{a,0} = \mathsf{lfsr}^a(\mathsf{P}(\mathsf{K}\|\mathsf{0}^{32}))$
— $\mathsf{mask}_K^{a,1} = \mathsf{lfsr}^a(\mathsf{P}(\mathsf{K}\|\mathsf{0}^{32})) \oplus \mathsf{lfsr}^{a+1}(\mathsf{P}(\mathsf{K}\|\mathsf{0}^{32}))$
— $\mathsf{mask}_K^{a,2} = \mathsf{lfsr}^{a-1}(\mathsf{P}(\mathsf{K}\|\mathsf{0}^{32})) \oplus \mathsf{lfsr}^{a+1}(\mathsf{P}(\mathsf{K}\|\mathsf{0}^{32}))$

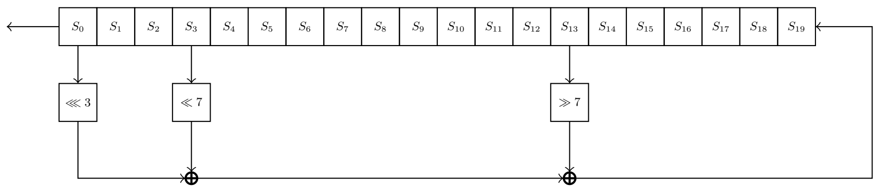

Fig. 2. The Dumbo LFSR

lfsr^a denotes a successive applications of the LFSR pictured in Figure 2 in which each S_i corresponds to one byte of the state, starting with the leftmost byte S_0. The state of the LFSR is initialized to $\mathsf{P}(\mathsf{K}\|\mathsf{0}^{32})$, also called $\mathsf{expandedKey}$.

As described in the previous subsection, $\mathsf{mask}_K^{a,0}$ is used to process the associated data during tag generation, $\mathsf{mask}_K^{a,1}$ is used to encrypt plaintext blocks, and $\mathsf{mask}_K^{a,2}$ is used to process the ciphertext blocks during tag generation.

We note that the LFSR used in the masking function is invertible, which means that it is possible to recover $\mathsf{mask}_K^{a-1,0}$ from $\mathsf{mask}_K^{a,0}$. In other words, it is easy to recover $\mathsf{expandedKey}$ from $\mathsf{mask}_K^{1,0}$. Given $\mathsf{mask}_K^{1,0}$, we simply shift the bytes to the right and compute $\mathsf{expandedKey}_0$ as follows:

$$\mathsf{expandedKey}_0 = (\mathsf{mask}_K^{1,0}{}_2 \lll 7)\&(\mathsf{mask}_K^{1,0}{}_{12} \ggg 7) \oplus \mathsf{mask}_K^{1,0}{}_{19}$$

In the next section, we demonstrate an attack that recovers $\mathsf{mask}_K^{1,0}$ using power analysis. Using that, we can then inverse the LFSR to recover $\mathsf{expandedKey}$ and finally inverse P to recover the key.

4 Our proposed attack

In this section, we describe how we use a CPA attack [6] against the first round of P during the computation of the authentication tag to recover $\mathsf{mask}_K^{1,0}$, and therefore K.

We recall that during the computation of the tag, the first block of associated data (after the first eight bytes) A_1 is absorbed by computing $\mathsf{mask}_K^{1,0} \oplus P(A_1 \oplus \mathsf{mask}_K^{1,0})$. We notice that the permutation P receives a 20-byte block of user-controlled data (A_1), which is bitwise XORed with the secret we want to recover ($\mathsf{mask}_K^{1,0}$). Our target for this attack is the first round of this particular invocation of the permutation P.

If we combine the exclusive or operation and the beginning of the first round, we can construct a model that, given the i^{th} byte of $\mathsf{mask}_K^{1,0}$ and of A_1, outputs the value of the i^{th} byte of the state after the $\mathsf{sBoxLayer}$ in the first round of P. We denote this model as $\text{Model}(a, k, i)$, where $i \in 0 \dots 19$ is the position of the byte, a is the i^{th} byte of A_1 and k is the i^{th} byte of $\mathsf{mask}_K^{1,0}$. This model is described in Algorithm 2.

```
Input: $a$, a 8-bit portion of the associated data
Input: $k$, a 8-bit portion of the key
Input: $i$, the byte of the state to compute
Output: $S$, byte $i$ of the state after the first round sBoxLayer
$S \leftarrow a \oplus k$        ▷ First operation of the first round: add c to the first and last bytes
if i=0 then
    $S \leftarrow S \oplus$ 0x75                                        ▷ 0b01110101
else if i=19 then
    $S \leftarrow S \oplus$ 0xae                          ▷ 0b1110101 in reverse order
$S \leftarrow \mathsf{SBox}(S)$        ▷ Second operation of the first round: sBoxLayer return S
```

Algorithm 2. $\text{Model}(a, k, i)$

What is interesting with this model is that if we have an oracle that can retrieve the value of the i^{th} byte of the state at this point in a real invocation of the cipher with a known associated data byte a, we can easily find k by running the model with all possible values for $k \in \{0, 1\}^8$ and stopping when it gives the correct output. Doing this again for all values of i recovers the entire $\mathsf{mask}_K^{1,0}$ in at most $2^8 \cdot 20$ attempts.

In our case, this oracle does not exist, but we can use CPA with this model to approach it. The idea behind CPA is to capture power traces of the target device encrypting with multiple known arbitrary values of a, then to compute the model with all possible values for k and all values we used for a to see which k predicts best the power consumption observed on the device. This works because the power consumption of a cryptographic device depends on the data that is being processed on that device. We will now describe the process in more detail.

4.1 Recovery of $\mathsf{mask}_K^{1,0}$

To perform a CPA attack on a cryptographic device, we need a general idea of the relation between the data processed by the device and its power consumption. In our case, we assume that the power consumption is proportional to the Hamming weight (the number of bits set to 1) of the data read from or written to memory.

The recovery of $\mathsf{mask}_K^{1,0}$ by CPA works as follows:

1. We generate an arbitrary number n of nonce and associated data pairs (N_j, A_j) with $j \in 0 \ldots n-1$, $N_j \ \$ \leftarrow \{0,1\}^{12}$ and $A_j \ \$ \leftarrow \{0,1\}^{28}$. Only the last 20 bits of A_j really need to be random, but for simplicity we generate all these values randomly.
2. We encrypt each of these pairs on the attacked device and record its power consumption, which we denote as T_j ; a vector of m power samples. All T_j have the same length and are synchronized on the same operation (they are *aligned*).
3. Using this data, we can now run the CPA on a computer. We describe the procedure to recover the i^{th} byte of $\mathsf{mask}_K^{1,0}$. We denote a_j the i^{th} byte of A_j ($a_j = (A_j)_i$).
    1. For each candidate value $k \in \{0,1\}^8$ and for each j, we compute the Hamming weight of the model prediction as $(H_k)_j = \mathsf{HammingWeight}(\text{Model}(a_j, k, i))$
    2. Then, we group each position in the power traces $t \in 0 \ldots m-1$ as vectors $P_t = ((T_0)_t, \ldots, (T_{n-1})_t)$
    3. For each k and each t, we compute the *Pearson Correlation Coefficient* [6] between samples H_k and P_t as

$$\rho_{k,t} = \frac{\text{cov}(H_k, P_t)}{\sigma_{H_k} \cdot \sigma_{P_t}}$$

    4. For each candidate k, we find the maximal correlation coefficient $\bar{\rho_k} = \max_t(\rho_{k,t})$
    5. We sort candidates by decreasing $\bar{\rho_k}$. The most likely value for the i^{th} byte of $\mathsf{mask}_K^{1,0}$ is $\arg\max_k(\bar{\rho_k})$.

In general terms, this means that for each timestamp we compute the correlation between the power consumption samples at that timestamp and the predictions of our model given a candidate for the key byte k. If the candidate is the correct value of $(\mathsf{mask}_K^{1,0})_i$, we expect that all the predictions of the model will be correct and correspond to a value that is processed during computation in the cryptographic device, which leads to a high correlation coefficient. On the other hand, while incorrect keys will sometimes give the same Hamming weight as the correct key (by the pigeonhole principle), computing the model with an incorrect key will most of the time return a value for which the Hamming weight is uncorrelated to the observed power consumption, which leads to a lower correlation coefficient. By doing this on all possible values for $(\mathsf{mask}_K^{1,0})_i$ and taking the highest correlation coefficient, we recover the correct value.

By running the CPA for each byte i of $\mathsf{mask}_K^{1,0}$, we get a sorted list of potential values that we call $\hat{K_i}$. We denote $(\hat{K_i})_0$ the value with the highest correlation and $(\hat{K_i})_{255}$ the value with the lowest correlation. We can derive a potential value for $\mathsf{mask}_K^{1,0}$ $\hat{K} = (\hat{K_0})_0 \| \dots \| (\hat{K_{19}})_0$ and use this value to recover a potential value for $\mathsf{expandedKey}$ by inverting the LFSR, as described in Section 3.2. Then, we can inverse P to recover a potential key.

4.2 Verification of key candidates

To verify that this key candidate is correct, we recall that $\mathsf{expandedKey} = P(K\|0^{32})$. This means that when we compute $P^{-1}(\mathsf{expandedKey})$, we expect to recover $K\|0^{32}$, and we can verify that $\mathsf{expandedKey}$ is correct by making sure the four last bytes of the result we obtained are indeed 0x00. Since P is pseudo-random, the likelihood of an incorrect $\mathsf{expandedKey}$ being inverted to a byte-array ending with four 0x00 bytes is 2^{-32}, which we consider low enough for this attack. If the attacker wants extra confidence, it is possible to verify the obtained key by obtaining a known (plaintext, nonce, ciphertext) triple on the attacked device, then trying to re-encrypt the plaintext with the obtained key and making sure it gives the same ciphertext.

Because CPA is a statistical method, it can be imprecise, and sometimes the potential $\mathsf{mask}_K^{1,0}$ we recover is incorrect, because the correct value for byte i is $(\hat{K_i})_1$ and not $(\hat{K_i})_0$. To account for this, we suggest a form of *exhaustive search* among potential keys. To make this analysis fast, we get rid of unlikely candidates and only keep $(\hat{K_i})_0$ to $(\hat{K_i})_3$ for all i. The number of candidates kept is arbitrarily chosen and may vary depending on the attacked device, but it is important to keep this number small. Then, we proceed to the exhaustive search by making a guess on the number of errors. First, we try to find the key by assuming only one byte of $\mathsf{mask}_K^{1,0}$ is wrong (i.e. $\exists j \in 0 \dots 19$ s.t. $(\mathsf{mask}_K^{1,0})_j \neq (\hat{K_j})_0$ and $(\mathsf{mask}_K^{1,0})_i = (\hat{K_i})_0 \forall i \neq j$). We ignore what j is, so we iterate on all possible values of j, and for each we try alternative values $(\hat{K_j})_1$ to $(\hat{K_j})_3$, until we find the correct key. If no correct key is found this way, we proceed in the same way but this time supposing thats there are two incorrect bytes, then three incorrect bytes. We could go on further, but we stop at three to keep the runtime in acceptable bounds, since the runtime of the exhaustive search with e errors is $4^e \cdot \binom{20}{e}$ checks.

5 Experimental results

In this section, we present the experimental results of our Python implementation on this attack on a ChipWhisperer [15] board.

5.1 Our Setup and Methodology

We confirmed that our attack works by implementing it on the ChipWhisperer [15] framework, using the LASCAR [1] toolbox for the optimized CPA computation. We chose this framework because it combines a target processor and an Analog to Digital Converter (ADC) on the same circuit board, making the attack easy to carry out and demonstrate. Another advantage of using this framework is that it makes the attack simple to reproduce by anyone, as the board used to demonstrate it is widely available. We also published the source code of our attack on GitHub to facilitate the reproduction in the kudelskisecurity/nist-lwc-power-analysis repository.

The ChipWhisperer board we used is the ChipWhisperer Lite ARM kit, containing a CW303 32-bit STM32F303RCT6 ARM Cortex-M4 microcontroller as the attacked device and a CW1173 ChipWhisperer-Lite capture board. The microcontroller clock is set at 7.4 MHz, and the sampling frequency is set to 29.6 MHz. The ADC has a 10-bit resolution and a 24k sample buffer.

The implementation we attacked is the reference implementation provided by the cipher authors as part of their submission to the NIST LWC, written in C. It was compiled with the ChipWhisperer toolchain, with an `-O3` optimization parameter. The compilation script and instructions are provided with the attack source code. While this implementation is an ideal case for our attack, other unpublished attacks on NIST LWC candidates show us that optimized versions written in assembly can usually still be attacked by using more traces. We are therefore confident that our attack would still work on alternative implementations as long as no power analysis countermeasures are implemented.

To validate our number of power traces, we ran our attack 200 times with a number of power traces set between 25 and 40. We did not test any lower value because of the very high failure rate. Each power traces contains precisely 24'000 voltage samples, which we only capture after the 974'000th sample. This attack point was selected by visual inspection of complete power traces to find the targeted S-Box operation. We carried the attack on a common laptop with an Intel Core i7-8565U CPU with four cores and a 1.8GHz base frequency. Since the laptop was running other processes, the exact time of the attacks is imprecise, but the goal of the benchmark was to give a general idea of the runtime. To limit the effects of multitasking on the results, we alternated the number of traces when running the benchmark: instead of running the attack 200 times for 25 traces, then for 26 traces, and so on, we ran the attack once for each number of traces, and then repeated this process a total of 200 times.

5.2 Results

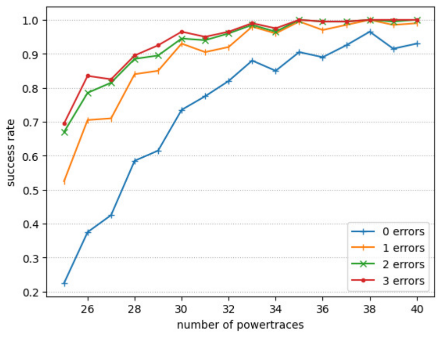

Fig. 3. Success rate by maximal number of errors

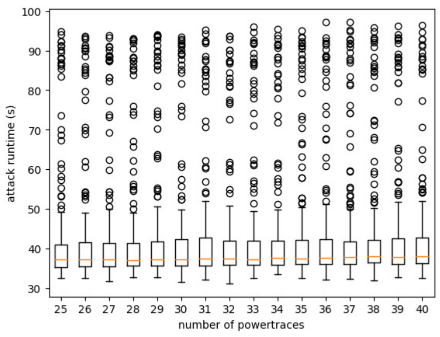

Fig. 4. Runtime of an entire attack attempt

The success rate (that is the number of successful key recoveries over the number of attempted key recoveries) was evaluated without any kind of exhaustive search step: we only evaluated the success rate of the CPA itself. The results are displayed in Figure 3. We can see that the CPA already recovers the correct mask in more than 90% of the cases when using more than 35 power-traces (max. 96% for 38 traces). However, it is also interesting to see *by how much* the CPA misses when it finds an incorrect mask. This is what the three other curves display: they show what the success rate would be if we used an exhaustive search step with up to 1 (respectively 2, 3) errors. We don't include in these curves errors that could not be recovered by the exhaustive search, that is attack attempts where at least one byte of the mask was not present in the top four most likely values returned by the CPA. These occur rarely when using more than 35 traces: only one error of the type was reported for 36 and 37 traces, and 0 for more. They are however more frequent on a lower number of traces: 28% of attempts with 25 power traces had at least one byte which could not be recovered by exhaustive search. Overall, we see that the attack has an almost 100% success rate with more than 35 traces and an exhaustive search step with up to 3 errors. We stress that this exhaustive search step is not even used in more than 90% of the attack attempts with that number of traces.

The performance measurement was done with a rather imprecise setup (the computer was also working on other tasks) and excluded any exhaustive search step. It only shows the time it takes to capture the power traces and to run the analysis, measured using Python's `process_time`. The results are displayed in Figure 4. Overall, we observe that 75% of the attacks take about 40 seconds or less, for all number of power traces. Only 10% of the attack attempts took more than one minute, and not a single attempt took more than two minutes.

6 Extending the attack to Jumbo

As we detailed earlier, Elephant has three members: Dumbo, Jumbo and Delirium. While Delirium uses a different permutation and cannot be attacked with the same method, Jumbo uses the same permutation as Dumbo with a different block size. This makes porting our attack to Jumbo easy.

The main differences between Jumbo and Dumbo are highlighted below:

— The underlying permutation is $\mathsf{Spongent}\text{-}\pi[176]$ instead of $\mathsf{Spongent}\text{-}\pi[160]$. This means the internal state is 176-bits (22-bytes) long instead of 160-bits (20-bytes) long. The permutation also has 10 additional rounds.

— The initial value of c in $\mathsf{Spongent}\text{-}\pi[176]$ is `0b1000101` instead of `0b1110101` in $\mathsf{Spongent}\text{-}\pi[160]$.

— The pLayer is slightly different:

$$\mathsf{pL}(j) = \begin{cases} 44j \bmod 175 & \text{if } j < 175, \\ 175 & \text{if } j = 175. \end{cases}$$

— The LFSR used to generate the masks is different. It operates on 22 bytes (instead of 20), and S_{21} is updated to $S_0 \lll 1 \oplus S_3 \ll 7 \oplus S_{19} \gg 7$.
— The number of 0 bits appended to the key to compute the first mask is 48 instead of 32.

The attack therefore works very similarly, by updating the model to reflect the changes in the constants and length of the state. We do not provide a detailed performance analysis of this variant of the attack, but its source code is included with the other attack.

7 Conclusion

We presented an efficient attack on Dumbo, the 160-bit version and primary instance of the NIST lightweight candidate Elephant, that can recover the secret key in about a minute using only 35 power traces. We described how this attack can be extended to the 176-bit variant of the cipher, which uses the same underlying permutation.

While we only attacked the software implementation of Dumbo, it would be interesting to see if hardware implementations of the cipher are vulnerable to this attack, and if so, how many traces are required to recover the secret key.

8 Acknowledgements

This work was supported by an internship at the Kudelski Security Research Team.

We would like to thank Nils Amiet, Aymeric Genêt, Sylvain Pelissier, Antonio De La Piedra, and Serge Vaudenay for their insightful feedback and suggestions.

References

1. LASCAR - Ledger's Advanced Side-Channel Analysis Repository, December 2021. original-date: 2018-10-31T15:38:09Z.
2. Subhadeep Banik, Sumit Kumar Pandey, Thomas Peyrin, Yu Sasaki, Siang Meng Sim, and Yosuke Todo. GIFT: A Small Present. Technical Report 622, 2017.
3. Guido Bertoni, Michaël Peeters, Gilles Van Assche, and others. The keccak reference. 2011. Publisher: Citeseer.
4. Andrey Bogdanov, Miroslav Knežević, Gregor Leander, Deniz Toz, Kerem Varıcı, and Ingrid Verbauwhede. spongent: A Lightweight Hash Function. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern,

John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Bart Preneel, and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2011*, volume 6917, pages 312–325. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. Series Title: Lecture Notes in Computer Science.

5. Jakub Breier, Dirmanto Jap, Xiaolu Hou, and Shivam Bhasin. On Side Channel Vulnerabilities of Bit Permutations in Cryptographic Algorithms. *IEEE Transactions on Information Forensics and Security*, 15:1072–1085, 2020.
6. Eric Brier, Christophe Clavier, and Francis Olivier. Correlation Power Analysis with a Leakage Model. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Gerhard Weikum, Marc Joye, and Jean-Jacques Quisquater, editors, *Cryptographic Hardware and Embedded Systems - CHES 2004*, volume 3156, pages 16–29. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. Series Title: Lecture Notes in Computer Science.
7. Information Technology Laboratory Computer Security Division. Request for Nominations for Lightweight Cryptographic Algorithms | CSRC, August 2018.
8. Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Ascon v1.2, 2019. Published: Submission to Round 1 of the NIST Lightweight Cryptography project.
9. M J Dworkin. Recommendation for block cipher modes of operation :: GaloisCounter Mode (GCM) and GMAC. Technical Report NIST SP 800-38d, National Institute of Standards and Technology, Gaithersburg, MD, 2007. Edition: 0.
10. Morris J. Dworkin. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. Technical Report NIST FIPS 202, National Institute of Standards and Technology, July 2015.
11. Gregor Haas and Aydin Aysu. Apple vs. EMA: Electromagnetic Side Channel Attacks on Apple CoreCrypto. Technical Report 230, 2022.
12. Xiaolu Hou, Jakub Breier, and Shivam Bhasin. DNFA: Differential No-Fault Analysis of Bit Permutation Based Ciphers Assisted by Side-Channel. Technical Report 1554, 2020.
13. Kerry A McKay, Larry Bassham, Meltem Sonmez Turan, and Nicky Mouha. Report on lightweight cryptography. Technical Report NIST IR 8114, National Institute of Standards and Technology, Gaithersburg, MD, March 2017.
14. National Institute of Standards and Technology. Advanced encryption standard (AES). Technical Report NIST FIPS 197, National Institute of Standards and Technology, Gaithersburg, MD, November 2001.
15. Colin O'Flynn and Zhizhang Chen. ChipWhisperer: An Open-Source Platform for Hardware Embedded Security Research. In Emmanuel Prouff, editor, *Constructive Side-Channel Analysis and Secure Design*, volume 8622, pages 243–260. Springer International Publishing, Cham, 2014. Series Title: Lecture Notes in Computer Science.
16. Keyvan Ramezanpour, Abubakr Abdulgadir, William Diehl, Jens-Peter Kaps, and Paul Ampadu. Active and Passive Side-Channel Key Recovery Attacks on Ascon.

17. Eyal Ronen, Adi Shamir, Achi-Or Weingarten, and Colin O'Flynn. IoT Goes Nuclear: Creating a ZigBee Chain Reaction. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 195–212, May 2017. ISSN: 2375-1207.
18. Adam Shostack and Bruce Schneier. Breaking Up Is Hard To Do: Modeling Security Threats for Smart Cards. 1999.
19. Tim Beyne, Yu Long Chen, Christoph Dobraunig, and Bart Mennink. Elephant v2.