Applying the Differential Collision Cache Attack on MPSoCs

Bruno Endres Forlin, Bruna Ramos de Carvalho, Altamiro Susin, Cezar Reinbrecht

***Abstract*—Side-Channel Cache attacks are known to be an efficient means of attacking desktops and embedded systems. In this paper, we present the adaptation of Bogdanov's [1] differential collision attack to an MPSoC platform. The attack was directed to an T-table implementation of the Advanced Encryption System (AES). It focuses on the cache access collisions caused by the sequence of lookups the encryption process requires. Specifically a situation called a $wide$, where a chain reaction of collisions happens in the same process. We evaluated it's execution on a real MPSoC implemented in an FPGA, results show that the attack can be successful in this scenario, but it is sensitive to the cache size.**

Keywords*—MPSoC, Side-Channel Attack, timing attack, collision attack, hardware security, AES.

I. Introduction

Side-Channel Attacks are a group of methods that monitor and explores any system, hardware or software, to reveal its information, which could be personal information, bank accounts, passwords. These side-channels focus on secondary and unintentional inputs and outputs of the computation performed, typically, a cryptographic task. For instance, the time to process an algorithm, the radiation emitted by integrated circuits, due to internal switching, the energy consumed during specific tasks and many others. Usually, SCAs explores the physical device, with highly accurate instrumentation. However, the present work targets the exploitation of the logical features, such as the timing leakage. The first mention in the literature about this leakage was a paper presented by Kocher [2] in 1996. There he described how the information of execution time of a cryptoprocess can be correlated and analyzed in order to find the secret key.

In recent years the use of cache memories made time oriented attacks more attractive, since the access speed of a CPU to a cache memory is at least an order of magnitude higher than to the main memory. These kind of attacks benefit greatly from different time accesses within the memory hierarchy. In addition, the last decade saw a shift from hardware implementations of symmetric key algorithms to software look-up table implementations, encouraging even more the use of cache oriented attacks. These attacks can usually be classified as trace driven, access driven and time driven [1]. Trace driven attacks rely on the attacker being able to observe every memory and cache transactions. As such, he knows the behavior of cache misses and hits at each access performed [3]. The access driven attack fills the cache with the attackers data, and after encryption takes place, checks which data is still present [4]. Time driven attacks take advantage of the fact that cache hits and misses cause huge differences in processing time, resulting in delays noticeable even by software. Many authors developed methods capable of manipulating cache behavior [5] [6] [7] to increase the likelihood of the attack strategy, and accentuate the timing behavior in some cases.

A different technique, called Differential Collision Cache Attack was developed by Bogdanov *et al*[1] in 2010. It relies on cache memory collisions, a situation where the CPU requests information to the memory hierarchy and the requested information is already inside the cache. This process speeds up the memory requisition from the CPU considerably. During the execution of the performance-oriented AES algorithm, a sequence of lookups is performed by the encryption at each round. It is possible that a situation called a $wide\ collision$ may happen. It is an specific case in which a series of collisions happen in the rounds of execution of the same encryption process. It can yield a significant difference in the execution time. It accomplishes such results by encrypting a pair of plaintexts in sequence, as to reduce interference by other processes. In addition, these plaintexts are created with this behavior in mind, and as such are designed to amplify the number of collisions. Reducing the number of samples required to achieve full key recovery.

In this attack pairs of plaintexts are created in a specific manner so that five AES S-boxes (one in round two and four in round three) process either pairwisely equal (what we call a $wide\ collision$) or pairwisely different values in two adjacent AES executions. When a $wide\ collision$ happens, the number of S-boxes collisions between the two AES runs will be the 5 higher than the average of times, when a $wide\ collision$ doesn't happen. Given enough samples it is possible to detect enough $wide\ collisions$ against the background, allowing the construction of four systems of nonlinear equations. These systems represent only part of the key, the rest being resolved by brute-force. Bogadnov's attack environment was an ARM920T set-up as a server running the AES implementation of OpenSSL [8], which was queried via the Ethernet interface of the board.

This technique presents several advantages over other attacks targeted at the cache memory. The original author [1] presented this attack as a viable alternative to previously published attacks, even in sight of practical setups. This assumption was contested by Bonneau and Mironov [6], as they disagreed about Bogdanov's attack being feasible in a real world scenario. The main contribution of our work is the implementation of the Differential Collision Cache Attack in an MPSoC environment. Where the attacker shares the same network as the secure CPU, reducing communication noises. We also present all adaptations required to implement the attack in such architectures. Results showed that the

implementation is feasible, even though the success of the attack is deeply dependent of the size of the cache and the configuration of the MPSoC.

This paper is divided in eight sections. Section II introduces the AES algorithm and the concept of the performance oriented implementation. Section III presents the related works regarding collision attacks. In Section IV, a detailed description of the attack is presented. Section V contains the information related to the execution in our MPSoC platform, and the adaptation performed. Section VI presents the experiments and results achieved through a platform developed by our team. Finally, we conclude the paper in section VII.

II. Advanced Encryption Cryptography

Advanced Encryption Standard is the preferred cryptography in many (mainly commercial) applications. This symmetric encryption operates inputs of 128 bits and keys of 128, 192 or 256 bits. This cipher algorithm uses iterations, called rounds, to perform a series of linked operations. In the case of a key of 128 bits, it is completed in 10 rounds. These activities refer to a substitution-permutation network because it replaces inputs by specific outputs and then shuffles the bits.

1) Encryption Process: The input is a plaintext of 128 bits organized as a block of 16 bytes, represented as $Plaintext \rightarrow P_i$, where $0 <= i <= 15$. AES arranges this block as a matrix of four columns and four rows:

$$Plaintext = \begin{bmatrix} x_0 & x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 & x_7 \\ x_8 & x_9 & x_{10} & x_{11} \\ x_{12} & x_{13} & x_{14} & x_{15} \end{bmatrix}$$

The same structure is applied to the key, for example, a 16 byte key is represented as $Key \rightarrow K_i$, where $0 <= i <= 15$:

$$Key = \begin{bmatrix} k_0 & k_1 & k_2 & k_3 \\ k_4 & k_5 & k_6 & k_7 \\ k_8 & k_9 & k_{10} & k_{11} \\ k_{12} & k_{13} & k_{14} & k_{15} \end{bmatrix}$$

Before starting the encryption, it is performed the key expansion, which transforms the key in several keys to be used for each round, called subkeys. This process can be represented as $expanded(K_i) \rightarrow k_i^{round}$, where $0 <= i <= 15$ and $0 <= round <= 10$. Figure 1 shows this process. Then, four operations are executed at each round, with an exception on the last one (figure 1):

- AddRoundKey: The 16 bytes of the plaintext or intermediate state (x) are considered as 128 bits and are XORed to the 128 bits of the subkey (k^{round}). If this is the last round, then the output is the ciphertext. Otherwise, the resulting 128 bits are interpreted as 16-bytes and continue with the following operations.
- SubBytes: The 16 input bytes are substituted according to a fixed table (S-box) given in design. The result is a new matrix of four rows and four columns.
- ShiftRows: Each of the four rows of the matrix is shifted to the left, in a circular manner (no data lost).
- MixColumns: Each column of four bytes is now transformed using a unique mathematical function. This function takes as input the four bytes of one column and outputs four entirely new bytes, which replace the original column. The result is another new matrix consisting of 16 new bytes. It should be noted that this step is not performed in the last round.

All these four operations can be represented as an iterative set of equations. Each equation presented at 1 represents the computation of each byte (of all 16 bytes) in the intermediate value. These intermediate bytes are iterated through these equations for ten rounds to accomplish the 128 AES algorithm. To calculate each part, the intermediate value from the current round is used as an index of the S-Box table, represented as the S. After changing its value, a circular shift operation is performed, where the number besides the S represents the number of shifts (01 is one shift to the left, 02 is two shifts to the left, and so on). When everything is ready, one can compute the AddRoundKey operation from the next round to be ready for the next iteration. Considering this algorithm, before beginning with these equations the inputs must perform an AddRoundKey in advance.

$$\begin{aligned}
(x_0^{r+1}) &\leftarrow 01.S[x_0^r] \oplus 01.S[x_5^r] \oplus 02.S[x_{10}^r] \oplus 03.S[x_{15}^r] \oplus k_0^{r+1} \\
(x_1^{r+1}) &\leftarrow 01.S[x_0^r] \oplus 02.S[x_5^r] \oplus 03.S[x_{10}^r] \oplus 01.S[x_{15}^r] \oplus k_1^{r+1} \\
(x_2^{r+1}) &\leftarrow 02.S[x_0^r] \oplus 03.S[x_5^r] \oplus 01.S[x_{10}^r] \oplus 01.S[x_{15}^r] \oplus k_2^{r+1} \\
(x_3^{r+1}) &\leftarrow 03.S[x_0^r] \oplus 01.S[x_5^r] \oplus 01.S[x_{10}^r] \oplus 02.S[x_{15}^r] \oplus k_3^{r+1} \\
(x_4^{r+1}) &\leftarrow 01.S[x_1^r] \oplus 01.S[x_6^r] \oplus 02.S[x_{11}^r] \oplus 03.S[x_{12}^r] \oplus k_4^{r+1} \\
(x_5^{r+1}) &\leftarrow 01.S[x_1^r] \oplus 02.S[x_6^r] \oplus 03.S[x_{11}^r] \oplus 01.S[x_{12}^r] \oplus k_5^{r+1} \\
(x_6^{r+1}) &\leftarrow 02.S[x_1^r] \oplus 03.S[x_6^r] \oplus 01.S[x_{11}^r] \oplus 01.S[x_{12}^r] \oplus k_6^{r+1} \\
(x_7^{r+1}) &\leftarrow 03.S[x_1^r] \oplus 01.S[x_6^r] \oplus 01.S[x_{11}^r] \oplus 02.S[x_{12}^r] \oplus k_7^{r+1} \\
(x_8^{r+1}) &\leftarrow 01.S[x_2^r] \oplus 01.S[x_7^r] \oplus 02.S[x_8^r] \oplus 03.S[x_{13}^r] \oplus k_8^{r+1} \\
(x_9^{r+1}) &\leftarrow 01.S[x_2^r] \oplus 02.S[x_7^r] \oplus 03.S[x_8^r] \oplus 01.S[x_{13}^r] \oplus k_9^{r+1} \\
(x_{10}^{r+1}) &\leftarrow 02.S[x_2^r] \oplus 03.S[x_7^r] \oplus 01.S[x_8^r] \oplus 01.S[x_{13}^r] \oplus k_{10}^{r+1} \\
(x_{11}^{r+1}) &\leftarrow 03.S[x_2^r] \oplus 01.S[x_7^r] \oplus 01.S[x_8^r] \oplus 02.S[x_{13}^r] \oplus k_{11}^{r+1} \\
(x_{12}^{r+1}) &\leftarrow 01.S[x_3^r] \oplus 01.S[x_4^r] \oplus 02.S[x_9^r] \oplus 03.S[x_{14}^r] \oplus k_{12}^{r+1} \\
(x_{13}^{r+1}) &\leftarrow 01.S[x_3^r] \oplus 02.S[x_4^r] \oplus 03.S[x_9^r] \oplus 01.S[x_{14}^r] \oplus k_{13}^{r+1} \\
(x_{11}^{r+1}) &\leftarrow 02.S[x_3^r] \oplus 03.S[x_4^r] \oplus 01.S[x_9^r] \oplus 01.S[x_{14}^r] \oplus k_{14}^{r+1} \\
(x_{15}^{r+1}) &\leftarrow 03.S[x_3^r] \oplus 01.S[x_1^r] \oplus 01.S[x_6^r] \oplus 02.S[x_{11}^r] \oplus k_{15}^{r+1}
\end{aligned} \tag{1}$$

2) Decryption Process: The decryption process follows the same algorithm. However, each step has to be made on the contrary. The expansion of the key remains the same. Then, the ciphertext (the input of this process) goes through the AddRoundKey step, but the first sum with the last part of the key (opposite way). The MixColumn and the ShiftRow also perform its operations in opposite way. In the end, the process outputs the plaintext recovered.

A. Performance-oriented AES

All operations performed by AES can be implemented using just logical and arithmetic operations. However, to obtain better performance, the cipher can be optimized for software implementations using a table with the operations pre-computed, as presented in [9]. The pre-computed operations comprise the

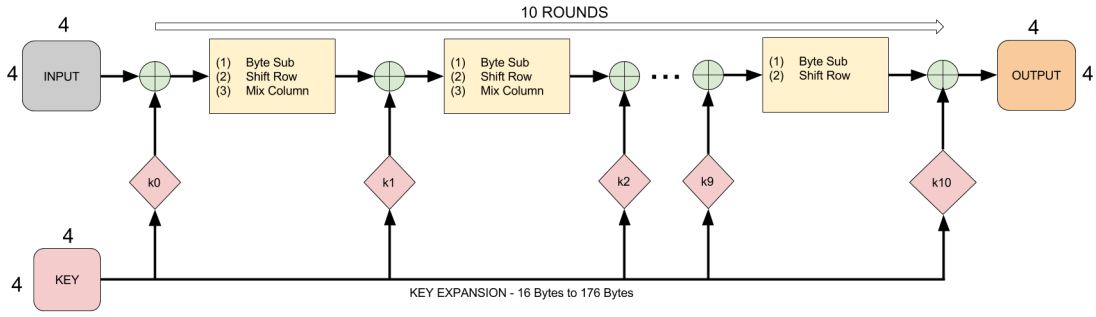

Figure 1. AES-128 encryption diagram, representing the main operations executed over the iterative process of ten rounds.

execution of SubBytes, ShiftRows and MixColumns for all possibilities (entries of 0 to 255), resulting in four tables of 1 kB, called the T-tables (T_0, T_1, T_2 and T_3). There is one more table (T_4) for the last round that does not use the MixColumns operation.

The performance-oriented AES has two main phases. The first phase generates the subkeys by key expansion ($expansion(K) \rightarrow k^{round}$). Each subkey is used in the AddRoundKey step to provide the next round input matrix. Each byte of this input matrix is related to an index of the T-tables, where its content represents all operations performed for such byte. As a consequence, the output of the T-tables consulting are XORed resulting in a new output matrix, that can be represented as an intermediate state as follows x_i^{round}, where $0 <= i <= 15$ and $0 <= round <= 9$. In summary, each intermediate state is used for the next round computation, which executes a XOR with the next round subkey (AddRoundKey operation) and the accessed T-tables values (SubBytes, ShiftRows and MixColumns operations) generating the next intermediate state. This mathematical iterated operation can be observed in 2.

$$\begin{aligned}(x_0^{r+1}, x_1^{r+1}, x_2^{r+1}, x_3^{r+1}) &\leftarrow T_0[x_0^r] \oplus T_1[x_5^r] \oplus T_2[x_{10}^r] \oplus T_3[x_{15}^r] \oplus k_0^{r+1} \\ (x_4^{r+1}, x_5^{r+1}, x_6^{r+1}, x_7^{r+1}) &\leftarrow T_0[x_4^r] \oplus T_1[x_9^r] \oplus T_2[x_{14}^r] \oplus T_3[x_3^r] \oplus k_1^{r+1} \\ (x_8^{r+1}, x_9^{r+1}, x_{10}^{r+1}, x_{11}^{r+1}) &\leftarrow T_0[x_8^r] \oplus T_1[x_{13}^r] \oplus T_2[x_2^r] \oplus T_3[x_7^r] \oplus k_2^{r+1} \\ (x_{12}^{r+1}, x_{13}^{r+1}, x_{14}^{r+1}, x_{15}^{r+1}) &\leftarrow T_0[x_{12}^r] \oplus T_1[x_1^r] \oplus T_2[x_6^r] \oplus T_3[x_{11}^r] \oplus k_3^{r+1}\end{aligned} \quad (2)$$

The last round is computed by repeating the equation 2 with $r = 9$, except that $T_0, ..., T_3$ is replaced by T_4. The resulting x_i^{10} is the ciphertext.

B. Crypto-libraries

In this section, we also analyze the commercial implementations of AES in software. Three widely used crypto libraries are described, namely OpenSSL [8], PolarSSL [10], and Libgcrypt [11]. All these AES solutions use the performance oriented approach. However, each one differs in the last round. Also, some of them do already contain methods to reduce or nullify cache-based side channel leakage.

OpenSSL:: It performs the last round using a table T_4, where the S-box and the ShiftRow are previously computed.

PolarSSL:: PolarSSL executes the last round using an S-Box table. It provides more security than OpenSSL since the granularity of such table is in bytes not words. The computation effort increases a little, mainly to perform the ShiftRow operation.

Libgcrypt:: This library calculates the S-Box values used in the last round during the encryption. The timing leakage generated by cache access can be mitigated since there is no table, but a high computation effort is inserted. Depending on the sensitivity of the attacker, this methodology could not be secure.

III. Related Works

Collision attacks have already been explored by several authors. Bonneau and Mironov [6] presented for the first time the cache collision atttack, and they describe three approaches: i) first round, ii) last round, and iii) expanded last round. The first round collision attack explores the identical accesses that may happen to the same T table. So, the attack aims to find all combinations of plaintexts and keys that result in the same index of the target T table. The drawback is that it was not possible to guess exactly which address was accessed, only the set. As a consequence, the attacker was capable of retrieving only 68 bits from the key. Even recovering just part of the key, the attack succeeded in retrieving the full key with an average of $2^{14.6}$ samples, a speed up compared to previous works, but considered even by the authors impractical in real life attacks.

$$\begin{aligned}C = \{&T_4[x_0^{10}] \oplus k_0^{10}, T_4[x_5^{10}] \oplus k_1^{10}, T_4[x_{10}^{10}] \oplus k_2^{10}, T_4[x_{15}^{10}] \oplus k_3^{10}, \\ &T_4[x_4^{10}] \oplus k_4^{10}, T_4[x_9^{10}] \oplus k_5^{10}, T_4[x_{14}^{10}] \oplus k_6^{10}, T_4[x_3^{10}] \oplus k_7^{10}, \\ &T_4[x_8^{10}] \oplus k_8^{10}, T_4[x_{13}^{10}] \oplus k_9^{10}, T_4[x_2^{10}] \oplus k_{10}^{10}, T_4[x_7^{10}] \oplus k_{11}^{10}, \\ &T_4[x_{12}^{10}] \oplus k_{12}^{10}, T_4[x_1^{10}] \oplus k_{13}^{10}, T_4[x_6^{10}] \oplus k_{14}^{10}, T_4[x_{11}^{10}] \oplus k_{15}^{10}\}\end{aligned} \quad (3)$$

To overcome such drawback, Bonneau and Mironov proposed an attack to the last round of AES. Typically, the last round requires the usage of an extra T table, known as table T_4, because the last round does not compute the MixColumn operation. Hence, the T_4 accesses are exclusive for the last round, and are represented as follows in 3, where C is the 16 byte output cyphertext. The enhanced last round attack uses not only the access of the equal indexes, but all index that provoke a cache hit (collision). Greatly increasing the speed of online

sample collection, at the expense of increasing processing in the offline stage. The different results of these methods can be observed in table I

Attack	Samples needed
First round	$2^{14.58}$
Last round	2^{15}
Expanded last round	2^{13}

Table I. NUMBER OF SAMPLES REQUIRED FOR EACH ATTACK

Bogdanov *et al* [1] proposed the attack target of study in this work, the differential collision cache attack. The main idea behind this attack is to choose pairs of plaintexts, so that in the first three rounds of AES encryption, five S-Boxes are processed pairwisely equal (hits) or different (misses). In the case of five collisions, as we call pairwisely equal processed S-Boxes, a *wide collision* happens. Bogdanov's tests show a success rate of around 65% on achieving full key recovery. Considering a low rate of false positives candidates detected, due to an optimized detection setup, he described the complexity of key search to be 2^{41}. A result that was acceptable for that time standards. This attack will be further detailed in section IV.

Spritzer and Plos [12] investigated the applicability of Bogdanov's attack in mobile phone devices. The evaluation used the same strategy of the original technique, exploiting the wide-collision behavior. However, the work showed that the wide-collision detection in mobile hardware platform was a big challenge. The main reason was the size of the cache line, which was 64 bytes for the tested devices. This line size made the attack unpractical, since few cache misses occurred. As a consequence, the chance of success in the detection stage of *wide collision* was about 10%. Besides, Spreitzer and Plos analysed the key recovery phase, and concluded that this lack of precision in the detection would increase significantly the false-positives. Therefore, to check all the possibilities would be in the order of 2^{52}, which is not feasible.

IV. DIFFERENTIAL COLLISION CACHE ATTACK

This attack is known as differential collision cache attack. This name refers to the exploration of collisions between pairs of plaintexts. These collisions affect the encryption time of the second plaintext, which becomes lower. It is challenging to detect the variations in time caused by the collisions, so Bogdanov explored a particular condition. He called it as the wide-collision. This situation has the potential to provoke five S-Boxes collisions in the first three rounds of the AES algorithm.

To create the wide-collision situation, the pair of plaintexts (P_1, P_2) have to follow a specific formation rule. Firstly, the attacker has to define a target diagonal. Then, he creates randomly both plaintexts, choosing pairwisely equal elements out of the target diagonal, and pairwisely different ones inside the target diagonal. The example below shows the main diagonal as the target, where the elements from P_1 are represented as a_i and from P_2 as e_i, where $0 < i < 4$:

$$P_1 = \begin{bmatrix} a_0 & x_1 & x_2 & x_3 \\ x_4 & a_5 & x_6 & x_7 \\ x_8 & x_9 & a_{10} & x_{11} \\ x_{12} & x_{13} & x_{14} & a_{15} \end{bmatrix} \qquad P_2 = \begin{bmatrix} e_0 & x_1 & x_2 & x_3 \\ x_4 & e_5 & x_6 & x_7 \\ x_8 & x_9 & e_{10} & x_{11} \\ x_{12} & x_{13} & x_{14} & e_{15} \end{bmatrix}$$

The encryption process of this pair (P_1, P_2) can provoke a wide-collision if at least one position of the S-Box is the same for both plaintexts in the second round of AES. The position of the second round S-Box is a result of the first round computation (Addround, SubBytes, ShiftRow and MixColumn). For example, to compute the element a_0 that results in S_0, the following equation 4 is used:

$$\begin{aligned} S_0 \leftarrow \ & \{02.Sbox(a_0 \oplus k_0) \oplus \\ & 03.Sbox(a_5 \oplus k_5) \oplus \\ & 01.Sbox(a_{10} \oplus k_{10}) \oplus \\ & 01.Sbox(a_{15} \oplus k_{15})\} \end{aligned} \tag{4}$$

After computing the first round for all elements (S_i), one obtains the new pair of plaintext (P_1^2, P_2^2) used to perform the second round, as follows:

$$P_1^2 = \begin{bmatrix} s_0 & s_4 & s_8 & s_{12} \\ s_1 & s_5 & s_9 & s_{13} \\ s_2 & s_6 & s_{10} & s_{14} \\ s_3 & s_7 & s_{11} & s_{15} \end{bmatrix} \qquad P_2^2 = \begin{bmatrix} s_0 & s_4 & s_8 & s_{12} \\ s_1 & s_5 & s_9 & s_{13} \\ s_2 & s_6 & s_{10} & s_{14} \\ s_3 & s_7 & s_{11} & s_{15} \end{bmatrix}$$

Note that the elements in grey represent pairwisely different values. The non-marked elements are pairwisely equal, but this is inherent to the wide-collision situation. This example considers a collision in S_0, identified by the elements in green (also pairwisely equal). Since all elements in the main diagonal are pairwisely equal, applying again the equation 4 in the second round pairs (P_1^2, P_2^2), one obtains four more collisions in the third round. The new pair of plaintexts (P_1^3, P_2^3) will collide for all elements in the first column. The plaintexts pair of the third round is represented as follows:

$$P_1^3 = \begin{bmatrix} t_0 & t_4 & t_8 & t_{12} \\ t_1 & t_5 & t_9 & t_{13} \\ t_2 & t_6 & t_{10} & t_{14} \\ t_3 & t_7 & t_{11} & t_{15} \end{bmatrix} \qquad P_2^3 = \begin{bmatrix} t_0 & t_4 & t_8 & t_{12} \\ t_1 & t_5 & t_9 & t_{13} \\ t_2 & t_6 & t_{10} & t_{14} \\ t_3 & t_7 & t_{11} & t_{15} \end{bmatrix}$$

Three main stages comprise the attack:

Online Stage:: For a pair of chosen 16-byte plaintexts (P_1, P_2), the main goal of the *online stage* is to measure the encryption time of P_2. To produce a wide-collision situation, the pair of plaintext has to follow the rule described above. Since this rule only applies for one diagonal at a time, this process repeats four times, one for each target diagonal. The total amount of encryptions required for each diagonal is $N * I * r$. The variable N represents the number of random values the same diagonal will explore. The variable I accounts for each iteration that the pairwisely equal values are changed to perform an in-depth exploration of all possibilities inside the same diagonal. The variable r represents how many times one performs the same encryption. The objective of this variable is

to reduce noise interferences. The duration of all encryptions is stored and sent to the detection stage.

Detection Stage:: Detection is achieved by analysis of all encryption times t. It is expected that t will be lower than average (the result of five look-up tables already in the cache memory) if a collision occurs. The result of this stage is a list of the pairs of plaintexts (P_1, P_2) that possibly caused a $wide\ collision$. They are also classified as candidates.

Key Recovery Stage:: This stage is divided into two steps. The first step supposes we found 4+m candidates on the detection stage, $m \in \{0, 1, 2, ..\}$. We consider all $\binom{4+m}{4}$ possibilities with all 2^{32} subkey candidates. For each choice of four pairs of (A_i, E_i) we execute AddRoundKey, SubBytes, ShiftRows and MixColumns. If a collision occurs for at least one position in the second round; for each of the four pairs in the group, the subkey candidate joins the final candidates' list. The total complexity of this step is in the order of $2^{32} * \binom{4+m}{4}$.

The second phase of this stage completes the full key recovery. It concatenates the subkey final candidates and performs an exhaustive key search. The total complexity of this searching algorithm is $(2^{32} * \binom{4+m}{4})^4$.

V. Practical Attack Implementation

The differential collision cache attack was adapted to attack our target MPSoC. This is the first practical implementation of this attack in a embedded MPSoC platform. Typical interferences by communication or parallel applications are lower in such devices. To accomplish the attack described by Bogdanov, the following adaptations were performed in each stage:

- Online Stage: The IP_{13} is a trusted processor inside a secure zone responsible for cryptography tasks in the system. The attacker runs from any processor outside the secure zone, in our example, IP_{04}. The malicious software performs the online stage generating the pairs of plaintext according to the rule described in the section IV. Before each pairs of encryption, the shared cache is filled with an attacker's data. The objective of that is to maximize the observation of cache hits and misses, since all data in the cache will be initialized in the same state. Only the time of the second encryption is collected. To optimize the online stage, a relaxed threshold is applied to collect only the results below the average. Then, less data storage is required for this step. The collected data was organized as the following array $Collected[t, A_i, E_i]$. The t is the encryption time, the A_i is the array of the elements in the target diagonal from the first plaintext, and the E_i is the elements from the second plaintext. Besides, it was possible to ignore the r loop since our platform was almost absent of external application interferences. The reason is because the latency of the NoC and other system bottlenecks does not impact in the same magnitude as a cache miss.
- Detection Stage: All the recorded data are then filtered with an even lower threshold. The key factor here is to select the minimum amount of candidates, because the key recovery stage will explore all combinations in arranges of four against all subkey possibilities, which brings a complexity of $\binom{4+m}{4}^4 * 2^{32}$. However, few samples implies in less possibilities to find the correct key. This trade-off is the most challenging issue regarding this attack.
- Key Recovery Stage: The input of this stage is given by $\binom{4+m}{4}$, which is the combination in arranges of four of the detected candidates. Each combination is checked against all 2^{32} subkeys. Using the equation 2, one searches over each subkey possibility which one generates a collision after the first round for all the four candidates in a group. As a result, each combination checked will output a subkey proposal. Then, all combinations of the proposals have to be tested in a exhaustive search key step. This stage was performed offline in a program written in C.

VI. MPSoC GLASS

Idealized not only for evaluating but also software simulating of the multiple processes that simultaneously occurs in a system-on-a-chip, the MPSoC Glass IDE supports C programming and compilation. Besides data access and I/O performance. According to these needs, Visual C was the perfect development environment of this IDE, once it introduces the Microsoft C and its .NET Framework programming library allied to dynamic design possibilities such as project templates, property pages, code wizards, an object model and more.

The interface is shown in Figure 2. There are many applications for MPSoCs that the interface can support. However the focus is the development, compilation and execution of applications in a MPSoC running in an FPGA. It basically consists in a sequence of steps which leads to a software simulation of data flow between fifteen IPs. IP0 and IP3 are unavailable to any application of the interface because they are respectively the shared cache and the external interface. Each IP button contains a programming environment where you can write your own code and save it as a text file. These files will become executable ones when the compiler is configured and the process of cross-compilation is called.

A. Experimental Environment

The evaluations of the differential collision cache attack were performed in a real MPSoC platform running on an FPGA. Thirteen processing elements compose the target MPSoC, one shared cache and two external interfaces, as shown in Fig. 3, interconnected by a 4x4 mesh-based NoC. The processing element is an ARM processor with L1 cache, a timer, and a network interface. The L1 is a direct-mapped cache with 32kB for instruction and data. The shared cache L2 is a set-associative 16-way cache with 256kB, and cache line size of 16 bytes (four words per line). The NoC is composed by a 5-port router, with input buffers of eight flits, each flit of 32 bits, and using an XY as the routing algorithm. A direct-mapped cache was used as L1, to reduce the noise during experiments. The hardware platform was connected to a host computer through UART protocol (USB adapter). The external interface does not affect any experiment, because only the data after processing was sent to the host. The key recovery stage

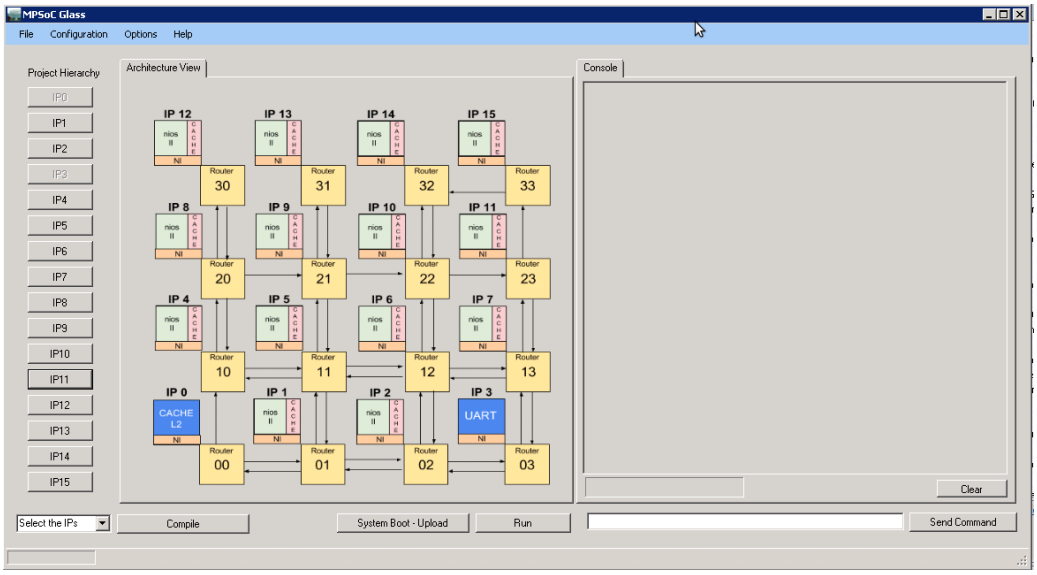

Figure 2. MPSoC Glass IDE Graphical User Interface.

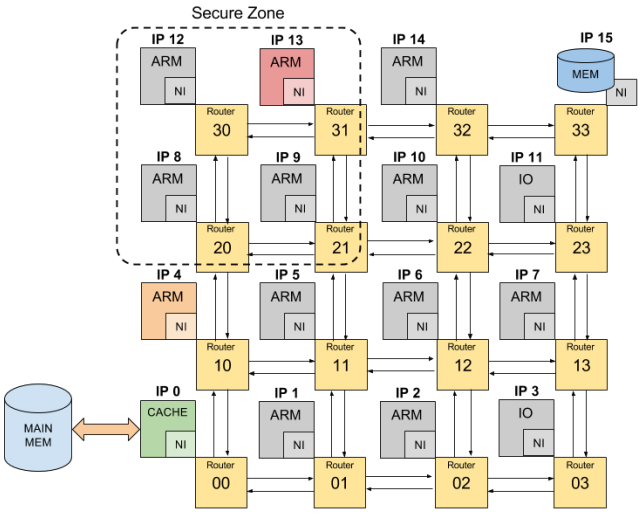

Figure 3. Profiling Phase - Signature of the position 0.

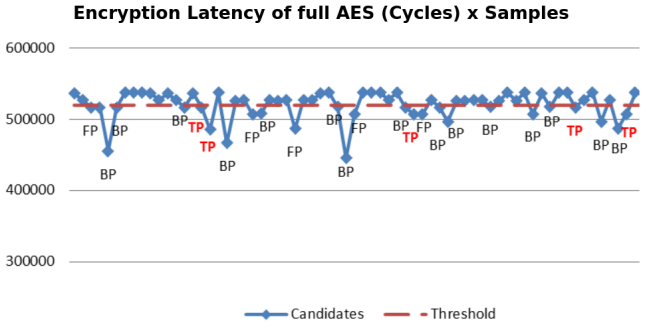

Figure 4. Detected candidates. The elements below the threshold are selected to the key recovery search.

was performed in the host computer through Python scripts and a program written in C.

The malicious software was downloaded to a non-trusted processor, located outside the secure zone at IP_4. The trusted processor at IP_{13} was responsible to run the AES encryptions with a secret key. To access the T tables, the crypto processor needs to communicate with the shared cache at IP_0.

B. Attack Execution

The online stage used the following parameters $n = 1000$, and $I = 800$. After an initial analysis, a relaxed threshold was applied in the online stage to select less elements. The time to process the online stage in the MPSoC running on an FPGA was about four hours. It resulted in eighty-six samples to be analyzed by the key recovery stage. Figure 4 shows the online stage output.

For analyses purposes, marks in the figure presents the status of each sample. The possibilities are true positive (TP), bad positive (BP), and false positive (FP). The true positives are the *wide collisions* that leads to the correct subkey values. The bad positives are the candidates that obtained cache hits due to the access of the same cache line. The false positives are the candidates without *wide collision*.

The attack requires that at least four true positives are present in the candidates to be analyzed by the key recovery stage. Hence, the attack of our experiment only works if a threshold of 510000 was applied. This resulted in thirty-four candidates to be analyzed, where five are true positives. Thirty-four candidates represents $\binom{4+34}{4} \rightarrow 46376$ groups to be checked. The key recovery stage looks for each group a subkey

candidate that leads to a collision in some position, meaning a search of $46376 * 2^{32}$ (or $2^{47.5}$). After this step, the attacker has 46376 possibilities for one subkey, and all combinations have to be checked together with the other subkey parts to compose the final key. The last step complexity stays in $(2^{15.5})^4 \rightarrow 2^{62}$.

VII. Conclusion

Works on the state-of-the-art have shown different threats for the SoCs, as well as countermeasures against them. However, few works were directed to multi-processor systems. This lack of information motivated this author to review in the bibliography the main attack proposals for SoCs, in order to identify a critical vulnerability of these new systems, specially for remote scenarios. The present research culminated in the investigation of a promising timing attack, based on the behavior of cache hits, known as collision attacks. The Bogdanov's collision attack was chosen to be studied and adapted to an MPSoC architecture. The process resulted in the first contribution of this paper, a threat model and a methodology of this attack when applied to MPSoCs. The practical evaluation running in an FPGA was made through an MPSoC environment, developed as part of this research, the MPSoC Glass. This platform was crucial to the implementation of the attack, enabling the execution of different rounds of tests, which were important to guarantee the quality of the results. Such emulation platform became even more important, in sight of the development of security research, to evaluate attacks such as countermeasures. Results of our experiments have shown that Bogdanov's differential collision attack is capable of revealing the secret key of AES. However, it had demonstrated a significant issue that must be addressed before any execution. The attack tries to identify a variation in the timing, caused by five cache hits during the cryptography. During an encryption, several cache hits between the elements can occur, which makes hard to detect the time from this five cache hits situation (also known as wide collision). Hence, a high number of samples must be selected for the key search space. The output of our tests in the online stage presented a high number of false positives, that originated from the detection stage. As a consequence, a higher number of samples was required. Since the key recovery stage analyses all combinations of the detected candidates, the final search complexity was $2^{47.5}$ for only thirty-eight elements. Such complexity is high but still possible to compute. Considering that we used four words per line in the cache, in a scenario with bigger caches, for example high performance systems, the attack might not be practical.

References

[1] A. Bogdanov, T. Eisenbarth, C. Paar, and M. Wienecke, *Differential Cache-Collision Timing Attacks on AES with Applications to Embedded CPUs*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 235–251.

[2] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, ser. CRYPTO 96. London, UK, UK: Springer-Verlag, 1996, pp. 104–113.

[3] D. Page, "Theoretical use of cache memory as a cryptanalytic side-channel," 2002.

[4] D. A. Osvik, A. Shamir, and E. Tromer, *Cache Attacks and Countermeasures: The Case of AES*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 1–20.

[5] D. J. Bernstein, "Cache-timing attacks on aes," Tech. Rep., 2005.

[6] J. Bonneau and I. Mironov, *Cache-Collision Timing Attacks Against AES*. Springer Berlin Heidelberg, 2006, pp. 201–215.

[7] M. Neve, J. pierre Seifert, and Z. Wang, "Cache time-behavior analysis on aes."

[8] T. O. S. Project, "Openssl: The open source toolkit for ssl/tls." Available at: www.openssl.org, accessed at 2017-10-07.

[9] J. Daemen and V. Rijmen, *The Design of Rijndael*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2002.

[10] PolarSSL, "Polarssl: Straightforward,secure communication." Available at: www.polarssl.org, accessed at 2017-10-07.

[11] G. Libgcrypt, "The libgcrypt reference manual." Available at: http://www.gnupg.org/documentation/manuals/gcrypt/, accessed at 2017-10-07.

[12] R. Spreitzer and T. Plos, "On the applicability of time-driven cache attacks on mobile devices (extended version)," in *Network and System Security*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 656–662.