A Secure White Box Implementation of AES Against First Order DCA

Ana Clara Zoppi Serpa[1], **Giuliano Sider**[1], **Hayato Fujii**[1], **Félix Carvalho Rodrigues**[1], **Ricardo Dahab**[1], **Julio López**[1]

[1]Institute of Computing – University of Campinas (Unicamp)
Av. Albert Einstein, 1251 – 13083-852 – Campinas – SP – Brazil

{hayato.fujii,felix.rodrigues,rdahab,jlopez}@ic.unicamp.br,
{ra165880, ra146271}@students.ic.unicamp.br

Abstract. *The white box threat model considers an attacker with complete access to the implementation and execution environment of a cryptographic algorithm. Aiming towards secure implementation of cryptographic algorithms in this context, several implementations of the AES cipher were proposed in the literature. However, they were proven vulnerable to implementation specific attacks, as well as to refined side-channel and more robust attacks that do not rely on implementation knowledge of the cipher, such as DCA (differential computation analysis). In this paper we present a white box implementation of the AES cipher with recently proposed DCA countermeasures [Lee et al. 2018]. We provide a comparison of the performance difference these countermeasures incur in practice and report some preliminary experimental results on the security of our implementation.*

1. Introduction

Traditionally it is assumed that encryption and decryption are performed in trusted environments and attackers can only compare plaintext and ciphertext pairs to mount an attack. However, this scenario does not reflect reality, especially with the increased reliance in mobile computer systems. In practice, malicious users (or malware) can have complete access to the implementation and execution environment of the algorithms, hence the need to consider such context in the design of secure cryptographic algorithms.

The concept of *white box cryptography* was first introduced in [Chow et al. 2003] with the proposal of a white box version of the AES cipher [NIST 2001] which attempts to hide a secret key by replacing intermediate operations with protected lookup tables. Such implementation considers a scenario in which an attacker has complete access to the full implementation and execution environment of a cryptographic algorithm, the *white box context*. However, this proposed implementation was shown to be susceptible to practical algebraic attacks [Billet et al. 2005], and every new proposed white box implementation of the AES cipher [Karroumi 2011] was also successfully attacked [Lepoint et al. 2014]. Since all the reported attacks depend on knowledge of the inner workings of each implementation, the industry became secretive regarding cryptographic white box designs. Albeit security through obscurity thwarts implementation specific attacks, novel techniques which do not require implementation specific knowledge emerged, such as the DCA (Differential Computation Analysis) attack [Bos et al. 2016].

A *Differential Computation Analysis* (DCA) [Bos et al. 2016] attack is a powerful adaptation of a DPA (*Differential Power Analysis*) [Kocher et al. 1999] attack in a white box context. In a DPA attack, an attacker collects power consumption signals during a cryptographic algorithm's execution and correlates them to the algorithm's intermediate operations, attempting to identify the key. A DCA attack is similar, but exploits the white box environment's advantages: instead of collecting power consumption signals, it collects software execution traces of memory positions and values stored in them. Then, it computes *correlation coefficients* between stored key-dependent values and hypothetical key values to pinpoint the most likely key candidate, i. e., the key candidate with the maximum correlation coefficient.

In this paper we describe a white box software implementation of the AES cipher which applies the static masking technique described in [Lee et al. 2018] as a countermeasure against DCA attacks. We present performance results of different masking levels, as well as experiments regarding first order DCA attacks on our implementation, confirming the reported protection against them.

2. White Box AES Implementation

The AES cipher encrypts (or decrypts) a block of 128 bits with a given secret key. Given a block of 128 bits (a state), the algorithm performs the following steps in each round:

ShiftRows A byte permutation of the state (16 bytes);
AddRoundKey A XOR between the current state and a round key;
SubBytes A byte substitution operation, based on a map called S-box;
MixColumns Each word of the state is multiplied by a 4×4 byte matrix *MC*.

The complete description of the four operations are given in [NIST 2001], as well the key derivation process, which generates all round keys given a master key.

For the white box implementation of the AES algorithm, the following description is considered: an *AES regular round* applies *ShiftRows*, *AddRoundKey*, *SubBytes* and *MixColumns*, in that order, to a 128-bit intermediate state. The AES encryption (with key size of 128 bits) comprises of 10 rounds: 9 regular ones and then the last round, which consists of *ShiftRows*, *AddRoundKey*, *SubBytes* and another *AddRoundKey* operation. For more details on this description of AES, please refer to [Chow et al. 2003].

2.1. White Box Software Implementation with Lookup Tables

Since in the white box context the attacker has access to the execution environment and can, for instance, check memory values, the main idea of a white box AES implementation is to hide the key and the intermediate values instead of computing and storing them in variables. To achieve that, the algorithm is divided in two parts, the *white box compiler* and the *white box implementation*. The white box compiler is responsible for generating lookup tables that, for each possible state byte x_i, contain the result of part of the algorithm's operations on x_i. The white box implementation then uses those tables to perform lookups in the correct sequence for encryption (or decryption).

The lookup tables will be referred as type II, type III, type IV and type V throughout this paper, following the notation of Chow et al. [Chow et al. 2003]. Simply using lookup tables implementing the AES operations, however, is still insufficient to protect the algorithm, since if the attacker knows the table generation process, it is simple to extract the key from the tables. In order to prevent that, *input/output encodings* and *input/output mixing bijections* are created [Chow et al. 2003], which are applied prior and after a table's core operation. An encoding is a non-linear mapping, while a mixing bijection is an invertible matrix in which all its 4×4 non-overlapping sub-matrices are also invertible. Encodings and mixing bijections must be chosen in a way that they can be reversed from one table to another, so that the core operations accurately reflect the AES specification. Chow et. al refer to this as *networked encodings/bijections*.

Type II tables perform *AddRoundKey*, *SubBytes* and multiplications of a state's byte by a column of the MC matrix. Thus, to obtain the *MixColumns*' correct result, type II tables' outputs must be XORed. These operations are performed by sets of type IV tables (also referred to as XOR networks); each type IV table operates with two input nibbles. Usage of the type III tables follows type II transformation and their XOR networks. Type III tables do not map a core operation, only reverting a type II's output encodings/bijections and applying new ones (which are reverted by the next type II tables). After the type III tables, more type IV tables must be applied. Hence, a regular AES round is composed by *ShiftRows* (performed without lookup tables), type II tables (round operations), type IV tables (type II's XOR networks), type III tables and more type IV tables (type III's XOR networks). The last round, which lacks *MixColumns* and comprises two *AddRoundKey* operations, relies on type V tables. Type I tables [Chow et al. 2003], further divided into type Ia and Ib tables, are related to external encodings, and thus out of the scope of this work. Figure 1 illustrates type II, III, IV and V tables usage.

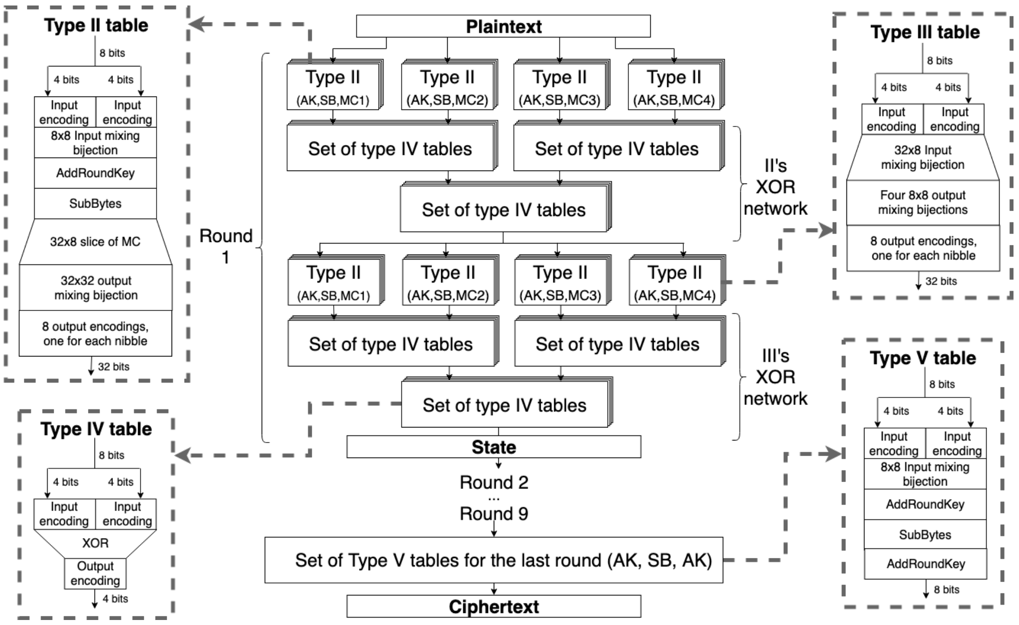

Figure 1. A scheme of the AES encryption in a white box context.

3. Static Masking

A DCA (differential computation analysis) attack consists in collecting memory positions and stored values to compute correlation coefficients between key guesses and key-dependent collected values, where the maximum correlation coefficient shows which key candidate is most likely to be the correct secret key of the white box implementation under attack. These correlation coefficients are calculated using *Walsh transforms*. A function's

Walsh transform indicates whether it is a *balanced m^{th} order correlation immune function*. A function is said to be m^{th} order correlation immune if every subset of n input variables $x_1, x_2, ..., x_n, n \leq m$, is statistically independent of $f(x_1, x_2, ..., x_n)$, where f is a boolean function. Considering this concept, the DCA attack exploits the fact that white box AES encodings are not correlation immune and thus the correlation coefficient for a correct key candidate will be easily distinguishable among all coefficients.

The *static masking* countermeasure proposal [Lee et al. 2018] focuses in reducing the correlation between intermediate values and the key before applying encodings. Towards this goal, uniformly random masks are generated prior to the tables' generation such that each byte i has an associated mask to be selected when needed. Lee et al. [Lee et al. 2018] claim that, since masking techniques were originally used to force power consumption signals to be uncorrelated with respect to secret keys to thwart DPA (*Differential Power Analysis*), applying *static masking* to a white box AES implementation can reduce correlation coefficients and thwart key identification in DCA. They also present 3 security levels and their changes on Chow et. al's design:

Level 1. Type II tables are replaced by type II-M tables, which perform the AES operations over a byte, like the type II tables, but also XOR the result with a random mask before applying the output mixing bijections and the output encodings.

Level 2. In addition to the changes from level 1, type III tables with larger input encodings replace type III tables on the 9^{th} round. Type IVs with larger input encodings replace type IVs in round 9 XOR networks. Type V with larger input encodings replaces type V used in the 10^{th} round.

Level 3. In addition to level 2 masking changes, type II-M tables with larger input encodings are required for round 3; type IIIs are replaced by type IIIs with larger input encodings in round 1 and type IVs are replaced by type IVs with large input encodings in round 1. The changes regard only encoding sizes, not the tables' core operations.

4. Applying a First Order DCA Attack on our Implementations

After implementing four variants of white box AES (one unmasked and three masked), we mounted a first order DCA attack using the tools from [Bos et al. 2016]. The attack output consists of 10 probable keys regarding maximum sum of correlation coefficients and 10 probable keys considering maximum correlation coefficients themselves.

For our first experiment, we compiled single instances of each implementation, all embedded with an example key provided in the AES specification [NIST 2001], `2b7e151628aed2a6abf7158809cf4f3c`, and varied the number of collected traces and attack positions considering experiments from [Bos et al. 2016] and the attack tools' available options. For the unmasked implementation, which we refer to as WBAES, when collecting 1000 traces the correct key was among the 10 most likely keys regarding correlation coefficient sum. Furthermore, all bytes of the most probable candidate key were correct except for the first and the last one. For the masked implementations, which we refer to as WBAES-M1 (masking level 1), WBAES-M2 (masking level 2) and WBAES-M3 (masking level 3), however, no key bytes were recovered. Table 1 shows this experiment's results. All attacks were performed on an Intel i7-7500U CPU, clocked at 2.70 to 2.90 GHz, with 8 GB of RAM. The trace collection and attack times were measured using the *time* command.

Table 1. DCA attack recovered keys and times, “%”” refers to bytes recovered. Correct key: 2b7e151628aed2a6abf7158809cf4f3c

Implementation	Traces	Collection	Attack	Most likely key found	%
WBAES	500	23m52.321s	2m59.800s	`ee`**`7e151628aed2a6abf7158809cf4f3c`**	94%
WBAES	1000	47m0.327s	5m52.476s	`84`**`7e151628aed2a6abf7158809cf4f`**`7d`	88%
WBAES-M1	500	32m52.781s	4m47.168s	`16e172bb6c31306999c2ef89db2669e3`	0%
WBAES-M1	1000	66m13.026s	8m49.485s	`1e25e6f36c718db71fc262f15ee6b14a`	0%
WBAES-M1	2000	145m3.757s	18m17.860s	`1140e654432ae672ef6a62d11ce6a908`	0%
WBAES-M2	2000	158m51.509s	19m26.333s	`62cffc8f4371e6b76ac2428975e63b4a`	0%
WBAES-M3	2000	160m57.974s	16m43.609s	`1eb4e60725718d9b540f62f1d1e3aec5`	0%

Our second experiment replicates the one described in [Lee et al. 2018]. We compiled 20 instances of each implementation (keys were generated by a PRNG). For the unmasked instances, with 200 traces the probable keys according to maximum correlation coefficient sum had 2 to 5 wrong bytes whilst the probable keys according to maximum correlation coefficient had 1 to 3 wrong bytes. For the masked instances, 10000 traces were collected and no key bytes were recovered, as reported in [Lee et al. 2018].

Table 2. Replication of Lee et al. [Lee et al. 2018] DCA experiments.

Implementation	Traces	Collection (avg)	Attack (avg)	Average number of key bytes found
WBAES	200	16m11.595s	1m10.507s	14.05 (= 88%)
WBAES-M1	10000	858m14.048s	36m30.370s	0

5. Performance Results

In order to analyze each masking level's performance, we compiled 32 instances (each instance uses a different, randomly generated, secret key) for each protection level. Table 3 shows the average cycles per byte (CPB) required to decrypt a 128-bit block of data, for 2^{15} iterations. Each repetition took the previous output as the input, except the first, in which the input was extracted from `/dev/urandom`. Tests were executed on a machine with a A53-based quad-core processor clocked at 1.15GHz. This CPU features 64KiB of L1 and 512KiB of L2 cache memory, with 2GB of RAM available. On the software side, the Linux kernel version 3.10.107 is used. The ciphers were implemented in C/C++, and compiled with `gcc` 5.4.0. Cycle count was measured using the Performance Monitoring Unit present in ARM CPUs.

Table 3. Performance comparison for different masking level implementations.

Implementation	Table lookups	Tables Size (bytes)	Performance (cycles per byte)
WBAES	2032	299008	23603.70
WBAES-M1	3328	593920	46340.20
WBAES-M2	3328	1642496	46525.50
WBAES-M3	3328	2691072	46131.49

Performance was similar for all levels of protection. This is due to the fact that while the size of tables increased between each masking level, the actual number of table lookups remained constant. Implementation size might lead to a difference in performance in hardware with larger available cache memory.

6. Conclusions

In this paper we report performance results and preliminary security analysis of our white box software implementation of the AES cipher [Chow et al. 2003] together with countermeasures (the static masking cases from [Lee et al. 2018]) against DCA attacks. Regarding performance, masked implementations spent about twice the cycles (per byte) of the unmasked one. For the unmasked implementation, first order DCA attacks were successful: the correct key was listed among the most likely keys recovered. However, no key bytes were recovered for the masked implementations. Recently a few attacks [Rivain and Wang 2019, Zeyad et al. 2019] were successful in recovering key bytes of a publicly available CASE 1 implementation of Lee et al.[1]. Since attack tools are not available, an interesting future direction would be to adapt existing attack tools to implement these attacks, and propose modifications to the static masking to prevent them.

7. Acknowledgments

This research was supported by Samsung Eletrônica da Amazônia Ltda., through the "White Box Cryptography" project, within the scope of the Informatics Law No. 8248/91.

References

Billet, O., Gilbert, H., and Ech-Chatbi, C. (2005). Cryptanalysis of a white box AES implementation. In Handschuh, H. and Hasan, M. A., editors, *Selected Areas in Cryptography*, pages 227–240, Berlin, Heidelberg. Springer.

Bos, J. W., Hubain, C., Michiels, W., and Teuwen, P. (2016). Differential computation analysis: Hiding your white-box designs is not enough. In Gierlichs, B. and Poschmann, A. Y., editors, *Cryptographic Hardware and Embedded Systems – CHES 2016*, pages 215–236, Berlin, Heidelberg. Springer.

Chow, S., Eisen, P., Johnson, H., and Van Oorschot, P. C. (2003). White-box cryptography and an AES implementation. In Nyberg, K. and Heys, H., editors, *Selected Areas in Cryptography*, pages 250–270, Berlin, Heidelberg. Springer.

Karroumi, M. (2011). Protecting white-box AES with dual ciphers. In Rhee, K.-H. and Nyang, D., editors, *ICISC 2010*, pages 278–291, Berlin, Heidelberg. Springer.

Kocher, P., Jaffe, J., and Jun, B. (1999). Differential power analysis. In Wiener, M., editor, *Advances in Cryptology — CRYPTO' 99*, pages 388–397, Berlin, Heidelberg. Springer.

Lee, S., Kim, T., and Kang, Y. (2018). A masked white-box cryptographic implementation for protecting against differential computation analysis. *IEEE Transactions on Information Forensics and Security*, 13(10):2602–2615.

Lepoint, T., Rivain, M., De Mulder, Y., Roelse, P., and Preneel, B. (2014). Two attacks on a white-box AES implementation. In Lange, T., Lauter, K., and Lisoněk, P., editors, *Selected Areas in Cryptography – SAC 2013*, pages 265–285, Berlin, Heidelberg. Springer.

NIST (2001). *Announcing the Advanced Encryption Standard (AES).* National Institute of Standards and Technology. Federal Information Processing Standards 197, `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`.

Rivain, M. and Wang, J. (2019). Analysis and improvement of differential computation attacks against internally-encoded white-box implementations. Cryptology ePrint Archive, Report 2019/076. `https://eprint.iacr.org/2019/076`.

Zeyad, M., Maghrebi, H., Alessio, D., and Batteux, B. (2019). Another look on bucketing attack to defeat white-box implementations. In *Constructive Side-Channel Analysis and Secure Design*, pages 99–117, Cham. Springer.

[1]`https://github.com/SideChannelMarvels/Deadpool/tree/master/wbs_aes_lee_case1`