International Journal of Applied Engineering Research
ISSN 0973-4562 Volume 10, Number 11 (2015) pp. 28669-28679

http://www.ripublication.com

Methodology To Reverse Engineer A JavaCard Applet using Pattern Matching Attack

Mohammed Amine Kasmi*, Mostafa Azizi and Jean-Louis Lanet
Lab MATSI, ESTO, Mohammed First University Oujda, 60000, Morocco
a.kasmi@ump.ma, azizi.mos@gmail.com
University of Limoges, Limoges, France
jean-louis.lanet@unilim.fr/jean-louis.lanet@inria

Abstract

In addition of power analysis, other techniques of side channel analysis exist, such as Electro-Magnetic Analysis (EMA). These techniques are widely used to obtain information about keys of implemented cryptographic algorithms. In this paper, we propose a similar methodology to apply reverse engineering of a JavaCard applet being executed over a JavaCard Virtual Machine (JCVM) in order to retrieve its source code, under several signal processing trails. Using EMA, detected signals are recorded during periods of time and then processed using different techniques. Here we are particularly interested in those techniques that allow us comparing recorded signals such as autocorrelation, variance, average and correlation, in the objective to build a database of instructions from captured signals and then recognize them from an unknown JavaCard bytecode. The obtained results using the correlation technique are very promising; for simple bytecode instructions, the reverse process is successful, but for those more complex, extensive work is needed.

Keywords: JavaCard platform, bytecode, side-channel attack, power analysis; electromagnetic analysis;reverse engineering; pattern matching; machine learning.

Introduction

JavaCard[1] is an open platform that allows emulating a smart card and a specific Java virtual machine called JCVM. JCVM interprets specific application instructions called "bytecodes". Multiple applications can be deployed on a single card, and new ones can be added to it even after it has been issued to the end user. Up to the mid-1990s, JavaCard smart cards were considered to be a black box that, given an input (plain-text), generates an output (cipher-text) by using a secret key. Thus, attacks were based on known plaintext, cipher-text or the pair of plain-text and cipher-text.

No more information was available. Today, there are additional outputs that can leak information, which can be observed and then exploited by cryptanalysts to break a cryptographic device, this kind of attacks is called the Side-Channel Attacks (SCA).

Side Channel Analysis (SCA) is a powerful technique to acquire information about running processes on devices such as smart cards by different methods like monitoring the power consumption [2],[3],[4],[5] or Electro-Magnetic(EM) emanations [6], [7], [8]. Recently, these methods have become of interest to be used for reverse engineering purposes (e.g. [3], [4], and [9]). In our work it is not the objective to reveal a secret key of a cryptographic device but to retrieve the program code being executed.

In this work, we want to examine what can be achieved in terms of reverse upon a JavaCard application using EMA.

In this paper, we introduce first the state of art in term of reverse engineering, and then we present our approach, in a second part we discuss the methodology proposed to apply reverse engineering process.

Related Work

In terms of reverse engineering, we present two popular attacks in this paper: the first one is called "Template Attacks" [9] [10] [11] which consist of identifying, for each instruction, a couple of statistical values. The model of captured side-channel information for one operation is called “Template”, this approach take into account the noise component that’s characterized by Multivariate-Gaussian noise model .A second approach, based on the pattern recognition [3], also called "Pattern Matching Attack", This attack consists in matching the patterns already identified in a database of instructions with the signal of an unknown applet to reverse-engineer it. The first step required to set up this attack is to build a dictionary of patterns for each bytecode instruction. The second step is to match patterns of the dictionary to the execution trace of an unknown applet to discover the operations effectively performed. This method tries to reduce noise by computing mean traces.

In our current work we apply “Pattern Matching Attack” in a first time, and then we will implement “Template Attack” in the aim to make comparison between these methods. As cited earlier in [3], the authors show how to acquire information about bytecodes executed on a JavaCard using power analysis. The method used in their work is based on averaging traces of certain bytecodes in order to correlate them to an averaged trace of an unknown sequence of bytecodes. This work describes the feasibility to acquire 75% of bytecode instructions. However, the authors did not describe the methodology and techniques used to build the database of patterns for each instruction.

Our Approach

Our final goal is the reverse engineering of program code and recognizing 100% of bytecode instructions. The approach we follow is different from the previous one in [3], since it is the intention to retrieve more information of a program running on a microcontroller by means of EM measurements and others information in the

correlation process to recognize more instructions. Under this promise we use more information in the correlation: the first information represents the execute "cycle" of virtual machine, the second and third information are obtained from the "prefetch-decode" cycle. Moreover, we describe clearly the methodology and techniques used in the reverse process. For this purpose we apply signal processing techniques that are known, such as average, autocorrelation, cross-correlation, variance and others to build database of instructions and then to recognize executed instruction sequences.

Reverse engineering process

Acquisition system

Figure 1 shows the acquisition System, which consists of an evaluation board with a specific JCVM connected to a digital oscilloscope.

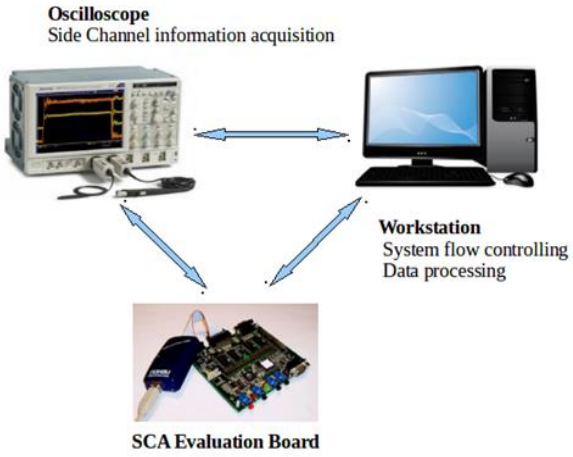

Figure 1: Acquisition system

In practice, we use a programmable JavaCard board as workbench in order to gain information and reverse an unknown JavaCard applet. A test program is stored into its memory and performed by the card processor. So by watching this memory, it is possible to find out which program is running and we are also able to manage the flow of instructions being submitted to the processor for acquisition needs.

Furthermore, in the JCVM interpretation loop, three sequences are discernible: The first part is the preamble, is to say the "prefetch-decode" cycle of a virtual processor, then the second part represents the execute cycle of bytecode, followed by a post-amble that depends on type of bytecode being executed.

When executing the preamble, the processor handles two parameters that are attached to the bytecode being read. These two parameters are: (1) the index vm_pc (virtual machine program counter) in the table of bytecodes, (2) the address of the function handler () to be executed. In addition, we have a third parameter (3): the execution trace of bytecode. vm_pc is the pointer to the array of bytecodes representing the method, so it is a pointer to a byte array. Thus, in a specific memory

address we could find the sequence of bytecodes which corresponds to our program. Then, we can acquire controlled acquisitions of curves by choosing the bytecodes to be executed in a memory area whose the instrumentation allows to measure. Through this technique we can execute and build a pattern for each bytecode instruction. Figure 2 shows an example of curve obtained when executing the following sequence of opcodes ‘0 0 10 0 0 0 0 57 0’. In JavaCard specification the value ‘10’ corresponds to the bytecode ‘BIPUSH’.

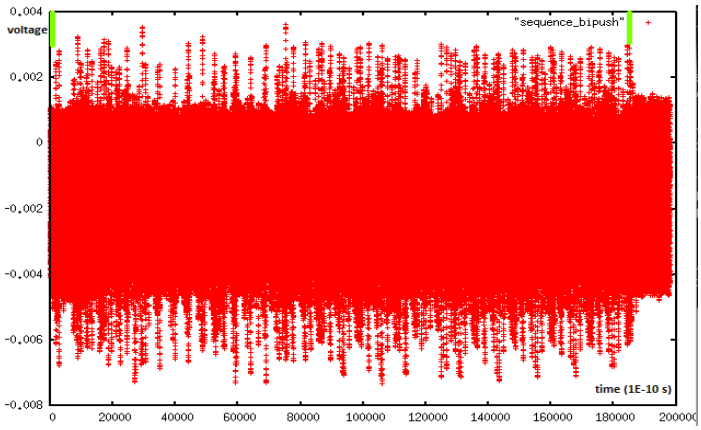

Figure 2: Execution trace of “BIPUSH” instruction

In figure 2, the green lines represent two calls to static methods: The first "Led On" and the second "Led Off". As discussed earlier “Pattern Matching Attacks" consists in two steps: the first step is to build a dictionary of bytecode instructions, the second step is to match patterns of the dictionary with the execution trace of an unknown applet to discover the sequence of instructions performed.

Figures 3 and 4 represent respectively the first and second step for the reverse engineering process:

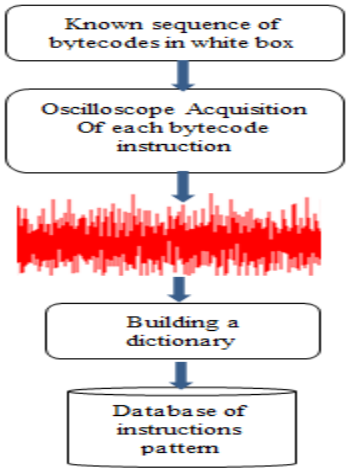

Figure 3: Pattern Building Phase

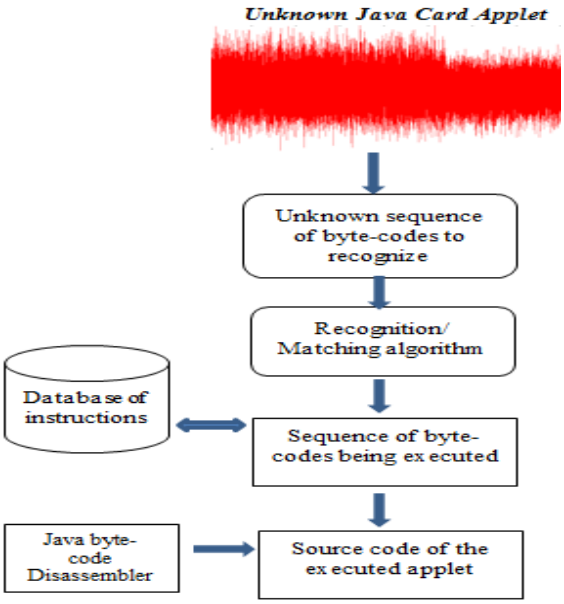

Figure 4: Pattern matching phase

Pattern Building Phase

In order to recognize bytecodes in an unknown signal, each bytecode needs to be represented by a unique pattern. To determine pattern for a specific bytecode, a test sequence that contains this bytecode is used.

The execution of the fetch-decode of the JCVM also corresponds to a specific pattern referred in this paper as "prefetch-decode pattern". This is advantageous, because this pattern can be used to split the EM emanations curves into separate parts representing the individual bytecodes.

Extracting The Prefetch-decode Pattern

The JavaCard bytecodes have all a common part; the "prefetch-decode" cycle, which is identical in time and EM emanations of all bytecodes. It could be very interesting to identify these common areas to extract the pattern of different bytecodes, and then relieves differences in execution between these bytecodes; the objective is creating a dictionary of bytecodes.

We can reasonably emit the hypothesis that during the execution of NOP bytecode, the only cycle performed of virtual processor is the "prefetch-decode" cycle. And thus the "prefetch-decode" is by the fact the NOP itself.

To extract the "prefetch-decode" pattern we study a sequence of several NOP instructions. We study a sequence of 9 NOP. By averaging multiple acquisitions, we obtain as depicted in figure 5 a repetition of the NOP pattern.

Average. Taking the arithmetic mean of a set of traces is an effective technique to remove noise.

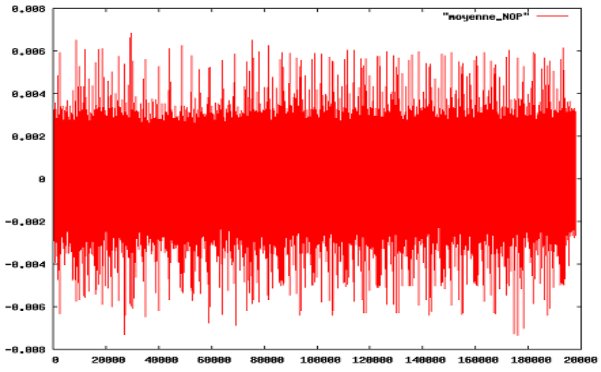

Figure 5: Curve of 9 NOP sequence

First we calculate the size of the NOP bytecode by using the autocorrelation technique. Detailed information about the autocorrelation function is given in (1).

Autocorrelation. Autocorrelation finds the correlation of a signal against different versions of itself time-shifted by various amounts. Each time-shift amount is called a lag time. The maximum value will always be at a lag of zero, since a signal is always perfectly correlated with an exact copy of itself. Other peaks in the autocorrelation indicate lag times at which the signal is relatively highly correlated with itself; these can be interpreted as periods at which the signal quasi-repeats. The autocorrelation coefficient at lag k of a series x_0 ,x_1,x_2,....x_{n-1} is normally given as :

$$\text{autocorr}(k) = \frac{\sum_{i=0}^{n-1}(x_i-\bar{x})(y_i-\bar{y})}{\sum_{i=0}^{n-1}(x_i-\bar{x})^2} \quad (1)$$

Where $\bar{x}$ is the mean of the series. When the term i+k extends past the length of the series n two options are available. The series can either be considered to be 0 or in the usual Fourier approach the series is assumed to wrap, in this case the index into the series is (i+k)% n.

Figure 6 depicts the autocorrelation curve obtained for the sequence of 9 NOP instructions.

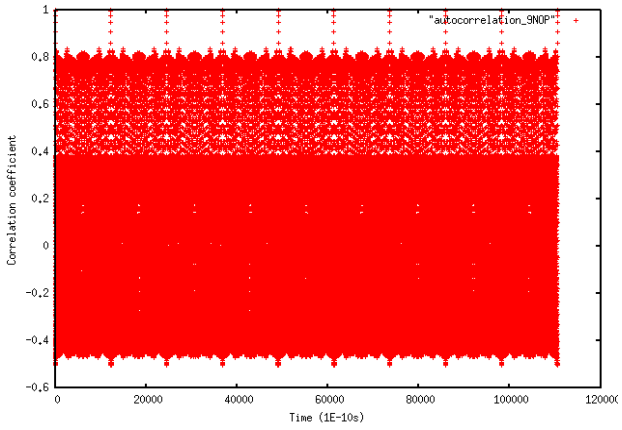

Figure 6: Autocorrelation of the curve of 9 NOP sequence

Thus, the distance between peaks in autocorrelation curve is exactly the size of the NOP. So we can measure size of the NOP instruction. Then we take a curve of sequence ‘NOP NOP BC NOP NOP NOP NOP POP NOP’ as 'c1' which BC is a bytecode instruction, and a curve of sequence for 9 NOP as 'c2', next we do a variance between 'c1' and 'c2'. Figure 7 shows the result of variance between the curves c1 and c2. Detailed information about the variance function is given in (2).

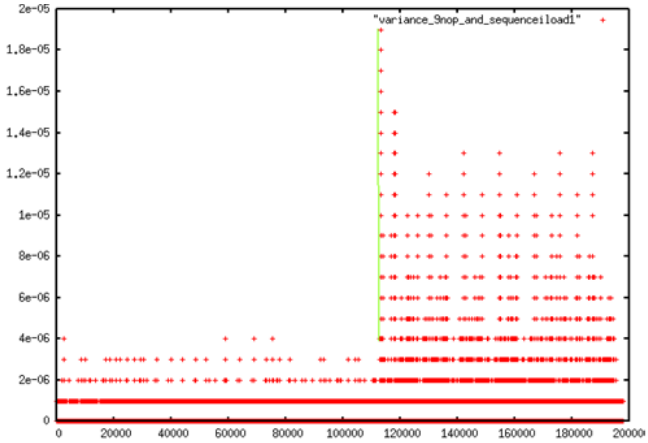

Figure 7: Variance between sequences of NOP and ILOAD1

Variance. In probability theory and statistics, variance measures how far a set of numbers is spread out: a variance of zero indicates that all the values are identical, a small variance indicates that the data points tend to be very close to the mean (expected value) and hence to each other, while a high variance indicates that the data points are very spread out from the mean and from each other. The variance for each point between n traces is given as:

$$\mathrm{var}(x) = \frac{1}{n}\sum_{i=0}^{n-1}(x_i - \bar{x})^2 \qquad (2)$$

The first part of the variance curve 'c3' will be almost flat (the Virtual Machine initialization, the two first NOP and the bytecode “prefetch-decode” cycle.).Then, starting from the first peak observed (green line); the “prefetch-decode” can be extracted using its size. As a result we will dispose a pattern of the “prefetch-decode” virtual processor cycle.

Building A Bytecodes Dictionary

In fact, now that we have a pattern of “prefetch-decode” cycle which is similar to a NOP instruction, then we can do a pattern matching using the method of correlation between this “prefetch-decode” pattern and the curves of each bytecode sequence, figure 8 shows the curve obtained by performing this pattern matching with a BIPUSH sequence ‘NOP NOP BIPUSH NOP NOP NOP NOP NOP’.

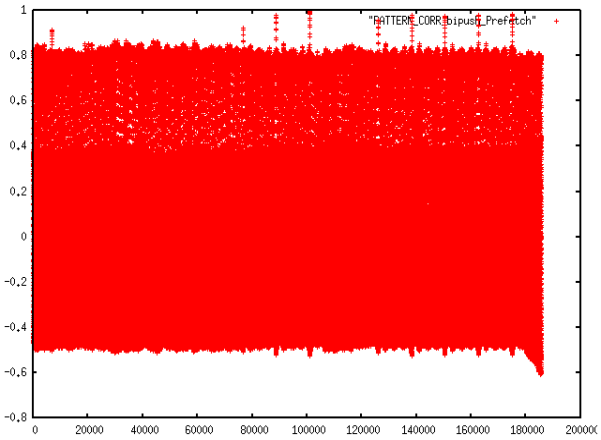

Figure 8: Correlation of a BIPUSH sequence with the 'prefetch-decode' Pattern

Correlation: Correlation gives a measure of association between variables. It returns a value between -1 and 1, where 1 means "identical in shape" and -1 means a "inverted in shape". Correlation 0 means that the values are uncorrelated. We use correlation to recognize specific bytecode patterns in an EM emanation trace.

The correlation peaks corresponds to the presence of 'prefetch-decode' pattern. Thus the distance between the peaks represents the full execution of a bytecode, that is to say, the execution of "prefetch-decode" cycle (common part to all bytecodes) and the "execute" cycle.

We can then extract and classify all bytecodes in categories per their size (number of points).

Pattern Matching Phase

The database of patterns determined in the previous section can be used to parse an unknown applet corresponding to an unknown sequence of bytecode.

Algorithm 1 shows the pseudo-code of the function "PatternMatching" that automatically matches n patterns against an average EM trace by using the correlation technique.

```
Function PatternMatching(curve)
        patterns[] = LoadPatterns()
        For i = 0 to patterns.length do
                pattern = patterns[i]
                For j = 0 to trace.length - pattern.length do
                correlationTrace[i] =
                                Correlation (pattern, trace[j..j+ pattern.length])
                End for
        End for
        Return correlationTrace
End function
```

Algorithm 1: Pseudo-code of PatternMatching function

Detailed information about the correlation function is described in (3), (4), (5), (6).

Pearson's correlation coefficient: The most familiar measure of dependence between two variables is the "Pearson's correlation coefficient". It is obtained by dividing the covariance of the two variables by the product of their standard deviations.

$$\text{corr}(x, y) = \frac{\text{cov}(x, y)}{\sigma_x \sigma_y} \quad (3)$$

The covariance of x and y provides a measure of how much x and y are related and is defined in (4):

$$\text{cov}(x, y) = \frac{1}{n} \sum_{i=0}^{n-1} (x_i - \bar{x})(y_i - \bar{y}) \quad (4)$$

The standard deviation and arithmetic mean are defined consecutively in (5) and (6):

$$\sigma_x = \sqrt{\frac{1}{n} \sum_{i=0}^{n-1} (x_i - \bar{x})^2} \; \sigma_y = \sqrt{\frac{1}{n} \sum_{i=0}^{n-1} (y_i - \bar{y})^2} \quad (5)$$

$$\bar{x} = \frac{1}{n} \sum_{i=0}^{n-1} x_i \qquad \bar{y} = \frac{1}{n} \sum_{i=0}^{n-1} y_i \quad (6)$$

The result of the correlation function is a set of n curves containing the correlation of the EM emanation curve with each bytecode pattern.

One basic example used for test: we take the trace corresponding to a sequence of "BIPUSH" bytecode (NOP NOP BIPUSH NOP NOP NOP NOP NOP) that we consider as a mysterious sequence to recognize, and we do its correlation with the "prefetch-decode" or the "NOP" pattern (figure 8), and with "BIPUSH" pattern (figure 9).

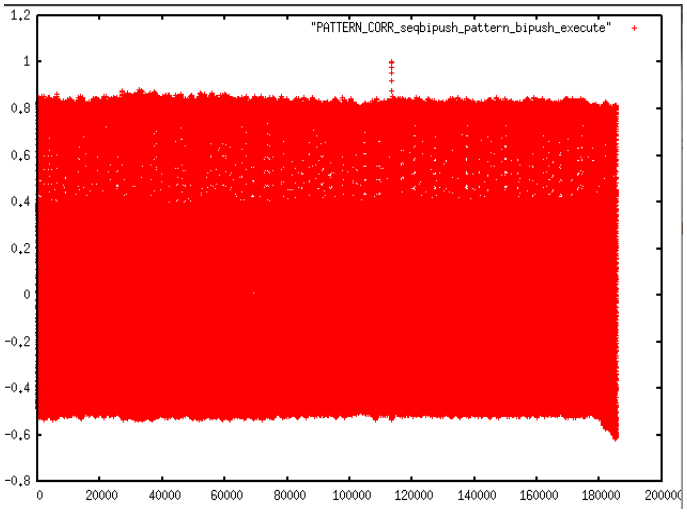

Figure 9: Correlation of a BIPUSH sequence with an extracted pattern of BIPUSH.

In figure 9 we match an extract pattern for bytecode 'BIPUSH' with the sequence of "BIPUSH" bytecode (NOP NOP BIPUSH NOP NOP NOP NOP NOP).

From peaks in Figure 8, it can be concluded that the bytecode "NOP" or "prefetch-decode" is executed eight times and from peak in figure 9 we see that the bytecode

"BIPUSH" is executed one time during the sequence execution. As a result we can find the sequence of bytecodes, by taking the pairs of values 'high correlation coefficient and time' we can deduce the order of bytecodes as follow: "NOP NOP BIPUSH NOP NOP NOP NOP NOP".

Conclusion and Perspectives

In this paper, we presented a methodology to reverse a JCVM applet bytecode using the "Pattern Matching Attack" based on signal processing techniques such as autocorrelation, variance and correlation. This attack is performed in two stages: (i) the pattern building stage with an open training device and (ii) the pattern matching stage on the device under attack. In addition, we proposed a new idea in the process of reverse engineering in the objective to recognize much more bytecodes with a high rate of accuracy. Our innovative idea is to use the cycles "prefetch-decode" and "execute" as three signatures of the same instruction. The idea is that the virtual processor fetches and decodes information before execution, so manipulated data should indicate what instruction will be executed and thus when we make our pattern recognition with the EM emanation curve of the bytecode "execute" cycle, we will get a redundancy to confirm or deny the information obtained with the "prefetch-decode" cycle, these three information are correlated together and can participate to the reverse engineering process. This idea represents the object of a future work which will be followed by concrete results. This is an ongoing research project and we are following our tests to obtain more specific results.

References

[1] Oracle: Java Card 3 Platform, Virtual Machine Specification, Classic Edition. Version 3.0.4. Oracle, Oracle America Inc, 500 Oracle Parkway, Redwood City, CA 94065 (2011)

[2] Kocher, P., Jaffe, J., Jun, B.: Differential power analysis. In: Wiener, M. (ed.) Advances in Cryptology - CRYPTO'99, Lecture Notes in Computer Science, vol. 1666, pp. 789–789. Springer, Berlin Heidelberg (1999).

[3] Vermoen, D.: Reverse engineering of Java Card applets using power analysis. Master's thesis, Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology,Computer Engineering, Mekelweg 4, 2628 CD Delft, The Netherlands (2006).

[4] Aranda, F.X., Lanet, J.L.: Smart card reverse-engineering binary code execution using side channel analysis. Théorie des Nombres, Codes, Cryptographie et Système de Communication (NTCCCS) (2012).

[5] Barbu, G.: On the security of Java Card™ platforms against hardware attacks. Ph.D. thesis, Grant-funded with Oberthur Technologies and Télécom ParisTech (2012).

[6] Gandolfi, K., Mourtel, C., Olivier, F.: Electromagnetic analysis: concrete results. In: Ko, C.C., Naccache, D., Paar, C. (eds.) Cryptographic Hardware and Embedded Systems ” CHES 2001, Lecture Notes in Computer Science, vol. 2162, pp. 251–261. Springer, Berlin Heidelberg (2001).

[7] Circuits, O., Ral, D., Guilley, S., Flament, F., Danger, J.L., Valette, F.: Characterization of the Electromagnetic Side Channel in Frequency Domain. In: Lai, X., Yung, M., D, D. (eds.) Information Security and Cryptology, Lecture Notes in Computer Science, vol. 6584, pp. 471–486. Springer, Berlin Heidelberg (2011).

[8] Quisquater, J.J., Samyde, D.: Electromagnetic analysis (EMA): measures and counter-measures for Smart Cards. In: Attali, I., Jensen,T. (eds.) Smart Card Programming and Security, Lecture Notes in Computer Science, vol. 2140, pp. 200–210. Springer, Berlin Heidelberg (2001).

[9] F.Xavier, JL.Lanet, Christophe Clavier &Jean-Christophe: Thèse “Méthode Automatisée de Rétro-Ingénierie sur Système Embarqué” , soutenue le 27/9/2012 à l’Université de Limoges.

[10] S. Chari, J. R. Rao, and P. Rohatgi.Template Attacks. In Cryptographic Hardware and Embedded Systems – CHES 2002, volume 2523 of LNCS, pages 13–28. Springer-Verlag, 2002.

[11] M. Goldack. Side-channel based reverse engineering for microcontrollers. Master’s thesis, Ruhr-Universität Bochum, Germany, 2008.