International Journal of Security and Its Applications
Vol.9, No.11 (2015), pp.347-356
http://dx.doi.org/10.14257/ijsia.2015.9.11.32

Reversing Bytecode of Obfuscated Java Based Smart Card Using Side Chanel Analysis

Mohammed Amine Kasmi*, Mostafa Azizi and Jean-Louis Lanet

Lab MATSI, ESTO, Mohammed First University Oujda, 60000, Morocco
Lab MATSI, ESTO, Mohammed First University Oujda, 60000, Morocco
University of Limoges, Limoges, France
a.kasmi@ump.ma,
azizi.mos@gmail.com
jean-louis.lanet@unilim
.fr/jean-louis.lanet@inria

Abstract

Side-channel Analysis (SCA) has become a reliable method for cryptanalysts to break cryptographic algorithms. Recently, SCA is used to reverse engineer the applet bytecodes on Java based smart cards. In addition of Power Analysis, other techniques of SCA exist, such as Electromagnetic Analysis (EMA). EMA of smart cards is a powerful technique that allows extracting information about the executed bytecode as well as about the processed data. In our work, we study the possibility to apply reverse engineering upon a Java Card applet in which the virtual machine is obfuscated by using SCA techniques. Even if this process of bytecode obfuscation is an effective way to prevent execution of an arbitrary and malicious bytecode, we believe that it can be systematically bypassed regardless the level of the platform encryption under the reverse engineering trails. In this paper, we present a methodology that could be used to find out the encryption key of the obfuscation process using the reverse engineering process through SCA. To perform reverse engineering, a white box approach providing access to the Java Card is needed in a learning stage, and then the technique used can be used on a black box approach where the code of applications is not accessible at the matching stage.

Keywords*: Java Card, bytecode obfuscation, side-channel analysis, power analysis; electromagnetic analysis, reverse engineering, pattern matching attack, template attack*

1. Introduction

A Java Card is a smart card that implements the standard Java Card [1] in one of the two editions: Classic Edition or Connected Edition. Such a smart card embeds a specific java virtual machine called Java Card Virtual Machine (JCVM) which interprets a specific instructions called bytecodes already romized with the operating system or downloaded after issuance. Due to security reasons, the ability to download code into the card is controlled by a protocol of GlobalPlatform [2].

The security of smart cards is studied at different levels, from the hardware to software level. Some families of the well-known attacks related to the hardware level are Power Analysis (PA) attacks [3-5] or EMA attacks [6-12]. There exists also other kinds of attacks, such as fault injection attacks and logical attacks [13-18].

*Mohammed Amine Kasmi is the corresponding author.

This attack is based on the exploitation of "getstatic bytecode" instruction and Converted APplet (CAP) file manipulation at pre-installation step, the common accepted assumption of this attack is that the targeted card is not equipped with an on-card Byte Code Verifier (BCV). To prevent this kind of attacks the obfuscation process seems to be an effective measure that consists to encrypt the value of instructions loaded with a "xor byte" [18].

Up to the mid1990s, smart cards were considered to be black boxes that received commands and retrieved answer to the terminal. After the arrival of the Java Card paradigm, with the possibility to download code after the post issuance, developers had the opportunity to load them self their code if they own the cryptographic keys to perform the mutual authentication process. This opportunity to load code offers the possibility to perform platform characterization which induces information leakage. Information leakage is obtained through the analysis of side channels like power consumption or electromagnetic emanations, which is called SCA. SCA is a powerful kind of hardware attacks used to acquire information about running processes on devices such as smart cards by different methods like monitoring the power consumption or electromagnetic emanation. Recently, these methods have become of interest to be used for reverse engineering purposes [3,4 ,9].

We study in this work the possibility of applying reverse engineering process on a Java Card applet for which the virtual machine is obfuscated. We think that this protection of obfuscation can be bypassed using SCA; from the signal of power consumption or electromagnetic emanations we can recognize patterns of encrypted bytecodes and then deduce the encryption key. Having the key, we are able to execute a 'shell code' and thus make logical attacks such as a 'memory dump' for example. In this paper, we introduce first the problem of code injection in the Java Card platform and a protection against this kind of attack which consists of instruction set obfuscation, and then we present the experimental scope of our work; in a second part, we discuss the methodology proposed to bypass this protection using the reverse engineering technique through SCA.

2. Issue Raised by Code Injection

The instruction set supported by the JCVM is published in the specification [1], thus the attacker knows the mapping between bytecode instructions (aload_0, invokevirtual, ifeq, istore_2 for instance) and their hexadecimal values (respectively 0x2A, 0xB6, 0x99, 0x3D). If the attacker is able to redirect the execution flow in memory to a malicious byte array by laser disruption or a logical attack EMAN2 as shown in [15][16] and [17], this bypasses the security provided by the firewall.

3. Logical attack « EMAN2 »

The smart card used is an open Java Card platform [1], which means that we are able to load and download new applications after delivery (post-issuance). In [14] [15] and [16], authors show an experimental result of the attack EMAN2 which consists to the redirection of execution control flow to a bytecodes array that corresponds to a sequence of malicious instructions, this attack is based on the knowledge of the instructions set supported by the JCVM specification.

The idea is to be able to execute a shellcode stored somewhere in the memory. The aim of EMAN1 attack [14], explained by Iguchi-Cartigny *et al.*, is to abuse the Firewall mechanism with the unchecked static instructions (like getstatic, putstatic and invokestatic) to call malicious bytecodes, this behavior is allowed by the Java Card specification. In a malicious CAP file, the parameter of an invokestatic instruction may redirect the Control Flow Graph (CFG) of another installed applet in the targeted smart card. The EMAN2 [15-17] attack was related to the return address stored in the Java Card stack, they used the unchecked local variables to modify the return address. The EMAN2 attack allows modifying the value of the return address of a method by storing a short into a local, by choosing the right value for the local number, the return address can be overwritten. In a given card, the return address register is stored at MAX_LOCAL + 2. The value stored in this register will be the address where Java Program Counter (JPC) will be updated while returning from the current method to the caller. We just need to define a static array which is stored close to the method area. Then, after returning from the method, the JCVM will execute the content of the array. Due to the fact that getstatic and putstatic instructions are not checked by the Firewall [14], we are able to read the content of the memory. The shellcode is presented in Listing 1:

```
7C 01 00 getstatic_b 0x0100
78 sreturn
```

Listing 1. Executing a Basic Shellcode

This piece of code pushes on the top of the stack, the content of the memory from the address 0x0100 and returns this value. The caller has just to store it into the APDU buffer and the value is sent to the terminal, then the third byte of the static array (the low byte of the getstatic_b instruction parameter) must be incremented and the next call will return the value of the address 0x0101. We just need to manage the carry from the low byte to the high byte representing the address.

4. The Bfuscation Process

Barbu in [13] proposed a countermeasure which prevents the malicious bytecode from being executed. His idea was to scramble each instruction during the installation step. For that, each Java Card instruction 'ins' performs a 'xor' with the 'Kbytecode' key. Thus, the obfuscated instructions are computed as the equation: $ins\_obfuscated = ins \oplus Kbytecode$. If an attack succeeds, as the EMAN2 described previously, the attacker cannot execute his malicious bytecode without the knowledge of the Kbytecode key. The obfuscation mechanism aims at preventing the interpretation of injected bytecodes. This countermeasure xors each instruction with a secret key value to randomise the byte stored in the memory, if someone tries to execute an arbitrary array, he will not be able to obtain the desired behavior.

5. Reverse Engineering Methodology

5.1. Related Work in Term of Reverse Engineering

In terms of reverse engineering, we present two popular attacks in this paper: the first one is called "Template Attacks" [9] [10] [11] and [12] 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 takes into account the noise component that is 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 signal identified for the pattern in the malicious applet with the signal of an official applet to reverse-engineer it. The first step required to set up this attack is to build a dictionary storing patterns for each bytecode instruction. The second step is to identify patterns of the dictionary in the execution trace of the unknown applet to discover the operations effectively performed. This method tries to reduce noise by computing mean traces.

5.2. Our Acquisition Platform

Figure 1, shows the acquisition System which consists of an evaluation board with a specific JCVM connected to a digital oscilloscope. At the top of the evaluation board processor we put an electromagnetic probe “Langer Near Field Probe”

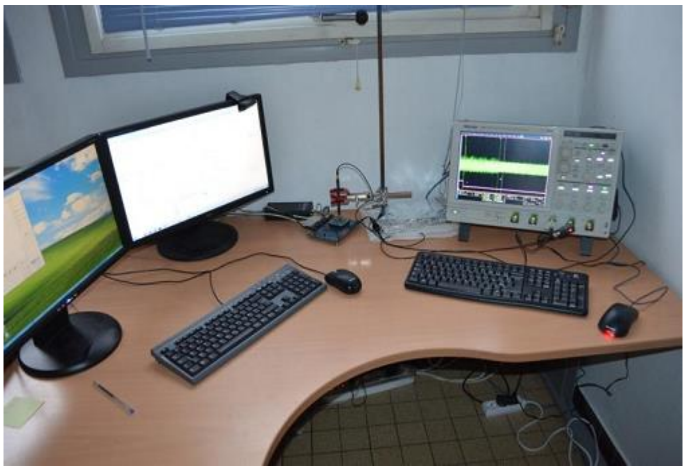

Figure 1. Acquisition System

In practice, we use a programmable Java Card board in order to gain information and reverse an unknown Java Card 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, as described in Figure 2, two 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.

```
for(;;) {
      // Prefetch/decode cycle
      bytecode= readU1(pByteCode, pPC);
     // execute cycle
      switch(bytecode) {
      case 0x0:
            COUT<<"nop"<<ENDL;
            break;
      case 0x1:
            COUT<<"aconst_null"<<ENDL;
            vm_aconst_null();
            break;
      case 0x2:
           COUT<<"sconst_m1"<<ENDL;
            vm_sconst(-1);
          …
    default:
    COUT<<"Unknown bytecode!"<<ENDL;
                  break;
    }
```

Figure 2. Bytecode Interpretation Code in an Open JCVM Implementation

In JCVM interpretation loop, the parameter ‘pPC’ of function ‘readU1’ is a pointer to a byte array that contains the code to execute in a memory area. Thus, in a specific memory address, we could find the sequence of bytes 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 know the pattern for each bytecode, Figure 3 shows an example of curve obtained when executing the following sequence of opcodes ‘nop nop aload1 nop nop nop nop nop nop’.

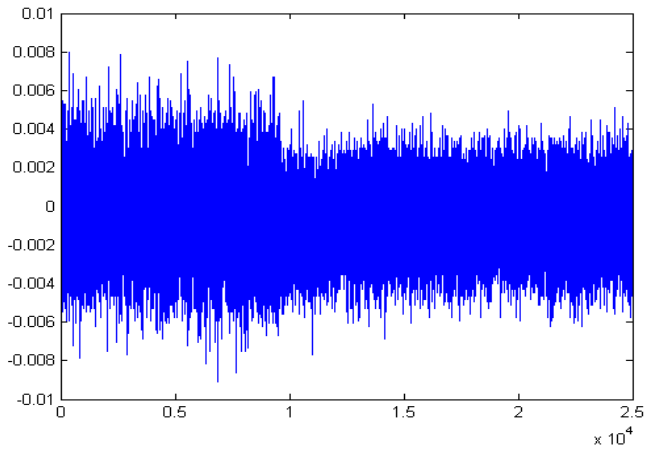

Figure 3. Execution Trace of Aload1 Instruction

As discussed earlier, side channel based reverse engineering attacks consist in two steps: the first step is to build a dictionary of bytecode instructions, the second step is to match instructions of the dictionary with the execution trace of an unknown applet to discover the sequence of instructions being performed.

6. Reverse Engineering Process

6.1. Learning Stage

In order to recognize bytecodes in an unknown applet, each bytecode needs to be represented by a unique pattern, to determine a 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 sequence of instructions into separate parts representing the individual bytecodes.

6.1.1. Extracting the Prefetch-Decode Pattern: The Java bytecodes have all a common part, the 'prefetch-decode pattern', which is identical in time and EM consumption 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 emit the hypothesis that during the execution of NOP bytecode, the only cycle performed of virtual processor is ‘prefetch-decode’. 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. For example, we study a sequence of nine NOP. By averaging multiple acquisitions, we get a signal as depicted in Figure 4.

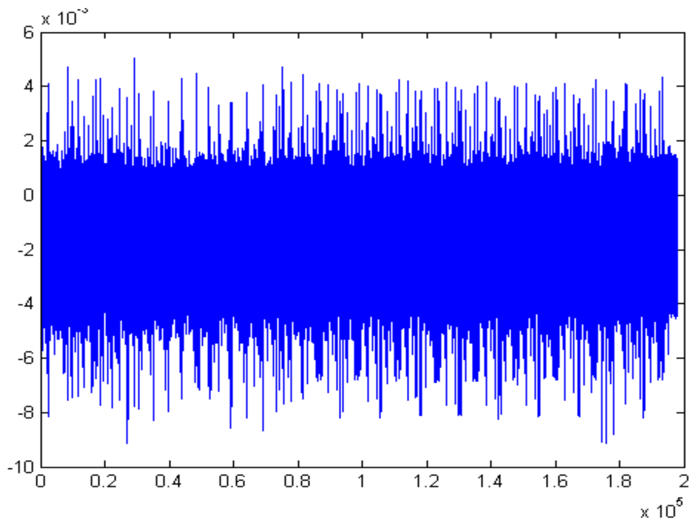

Figure 4. A Sequence of 9 NOP

Then to extract the repeated pattern we use the autocorrelation tool. Figure 5 depicts the autocorrelation curve obtained for the sequence of 9NOP instructions. Detailed information about the autocorrelation function is given in Appendix A (1).

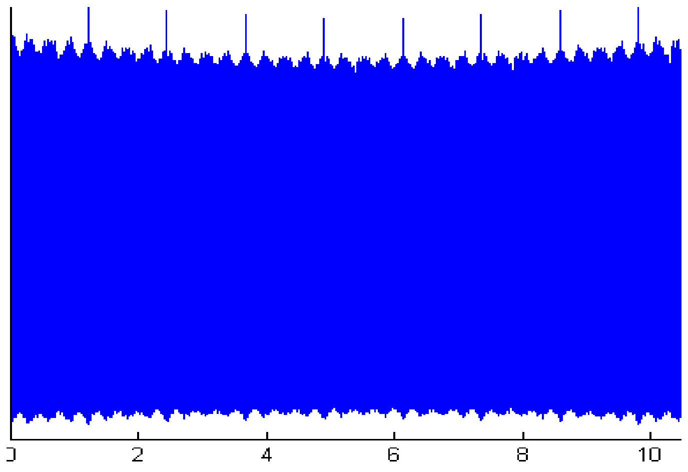

Figure 5. Autocorrelation of NOP Sequence (9 NOP)

The distance between peaks in autocorrelation results is exactly the size of the prefetch-decode pattern. So we can measure size of the prefetch-decode pattern. Then we take a sequence ‘nop nop aload1 nop nop nop nop nop nop’ as 'seq1', and a sequence for nine NOP as ‘seq2’; Next, we do a variance between 'seq1' and 'seq2'. Detailed information about the variance function is given in Appendix A (2).

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

6.1.2. Building Bytecodes Dictionary: In fact, now that we have a template of 'prefetch-decode ', then we can perform a template matching using the method of correlation between this 'prefetch-decode' pattern and the curves of each bytecode sequence. Figure 6 shows the curve obtained by performing this pattern matching with an iload1 sequence (nop nop iload1 nop nop nop nop nop nop).

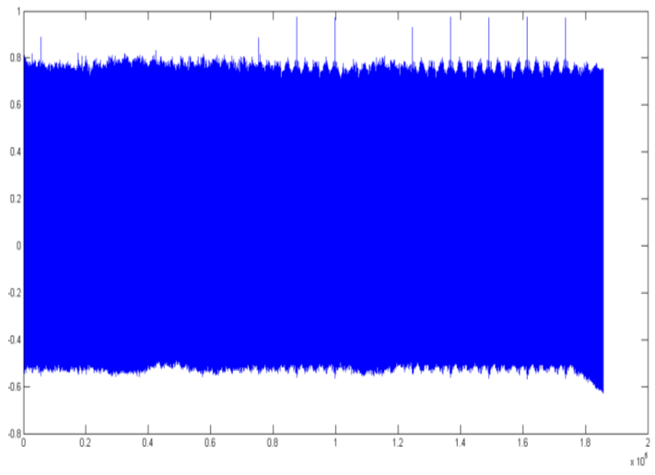

Figure 6. Correlation of a Sequence iload1 with the ‘Prefetch’ Pattern.

The correlation result peaks correspond 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 families per their size (number of points). For example, in the case of Figure 6, we extract pattern for bytecode ‘iload1’.

6.2. Matching Stage

The patterns determined in the building phase can be used to parse an unknown sequence of bytecodes corresponding to an unknown Applet. The implemented algorithm automatically matches n patterns against an average Electromagnetic trace by using the Pearson's correlation coefficient. Detailed information about the correlation function is described in Appendix A: (3), (4), (5), and (6).

The example used for test: we take the trace corresponding to a sequence of bipush bytecode (nop nop bipush nop nop nop nop) that we consider as an unknown sequence to recognize, and we do its correlation with the ‘prefetch-deode’ pattern (Figure 6), and with bipush pattern (Figure 7). From peaks in Figure 6, it can be concluded that the pattern of ‘prefetch-decode’ cycle or ‘nop’ bytecode is executed eight times and from peak in Figure 7 we see that the bytecode ‘iload1’ 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; time’ we can deduce the order of bytecodes as follow: “nop nop ilaod1 nop nop nop nop nop”.

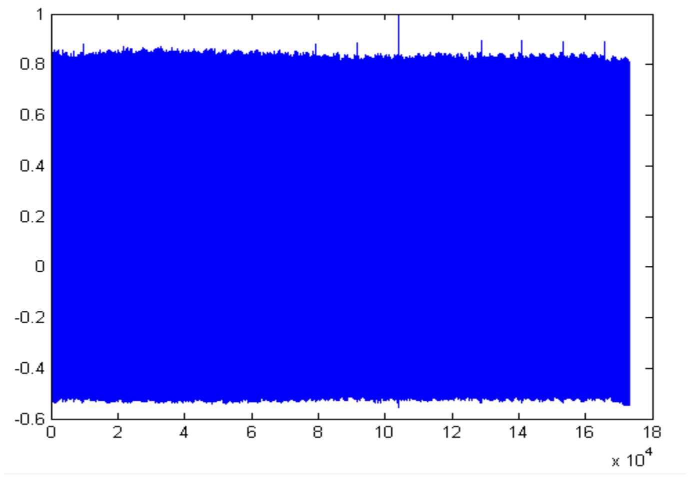

Figure 7. Correlation of iload1 Sequence with the "Execute" Pattern of iload1.

6.3. New Idea for the Reverse Engineering Process

In the JCVM interpretation loop, as described in Figure 2, 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.

Our innovative idea in the process of reverse engineering 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 BC 'execute' cycle, we will get a redundancy to confirm or deny the information obtained with the 'prefetch-decode' cycle.

6.4. Find the Encryption Key

In the first time, we load an applet that contains a sequence of known bytecodes, then at the load step each bytecode will be encrypted with a key 'k' (a xor k = a'), in the runtime we decrypt the bytecode before execution (a' xor k = a), thus if we have the clear bytecode 'BC loaded in a known sequence and the encrypted one BC' obtained by the reverse process, we can deduce the encryption key used for the obfuscation process .

7. Conclusion and Perspectives

In this paper, we proposed a methodology to bypass the JCVM obfuscation process using the EMA technique that can be used to perform a reverse engineering on a scrambled Java Card applet bytecode. In this paper, we also presented techniques and methodology to reverse a JCVM bytecode applet using "Pattern Matching Attack" based on signal processing techniques like correlation. These techniques are performed in two stages: (i) the building stage with an open training device and (ii) the matching stage on the device under attack. In addition, we proposed a new idea in the process of reverse engineering which consists of using the virtual processor cycles 'prefetch', 'decode' and 'execute' as three signatures of the same instruction in the goal to recognize more bytecodes with a high percent of accuracy. This is an ongoing research project and we are following our work by tests and comparison between our methodology and others like

"Template Attacks" technique for reverse engineering in the aim to obtain more specific results.

Appendix A

Average: A single curve is noisy. Taking the arithmetic mean of a set of traces is a simple but an effective technique to remove noise.

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 output of an autocorrelation is the correlation amount as a function of 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. In other words, autocorrelation is based on the idea that a quasi-periodic signal will resemble itself in the time domain when time-shifted by duration equal to the period.
The autocorrelation coefficient at lag k of a series x_0 , x_1,x_2,....x_{n-1} is normally given as :

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

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. If the correlation coefficient is calculated for all lags k=0,1,2...n-1 the resulting series is called the autocorrelation series or the correlogram.

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:

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

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 byecode patterns in a EM emanation trace.

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.

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

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

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

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

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

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

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] GlobalPlatform: Card Specification. In: GlobalPlatform, 2.2.1 edn. GlobalPlatform Inc. **(2011).**

[3] D. Vermoen, M. Witteman, G. N. Gaydadjiev, "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] F.X. Aranda, J. L. Lanet, "Smart card reverse-engineering binary code execution using side channel analysis", Proceedings of the Number Theory, Codes, Cryptography and Communication System (NTCCCS) Morocco, Oujda, **(2012)** April 26-28.

[5] T. Eisenbarth ,C. Paar, B. Weghenkel, "Building a side channel based disassembler", Journal of Transactions on Computational Science X. LNCS, vol. 6340, **(2010)** pp. 78–99.

[6] K. Gandolfi, C. Mourtel, F. Olivier, "Electromagnetic analysis concrete results". Edited K.K.Çetin, D. Naccache, C.Paar , Cryptographic Hardware and Embedded Systems, Lecture Notes in Computer Science, vol. 2162, Berlin, Heidelberg **(2001)**, pp. 251–261.

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

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

[9] F. X. Aranda, J. L. Lanet, "Automated reverse engineering method on embedded systems", Proceedings of the Congress of CRYPTO'PUCES 2011, Limoges, France, **(2011)** May 8-12.

[10] S. Chari, J. R. Rao, P. Rohatgi, "Template Attacks", In proceeding of the 4th International Workshop on Cryptographic Hardware and Embedded Systems, CHES 2002, volume 2523 of LNCS, Springer-Verlag, **(2002)**, pp 13-28

[11] Goldack, Martin, C. Paar. "Side-channel based reverse engineering for microcontrollers", Master's thesis, Ruhr-Universität Bochum, Germany **(2008)**.

[12] C. RECHBERGER, et E. OSWALD, "Practical template attacks", Journal of Information Security Applications. Springer Berlin Heidelberg, **(2005)**. pp. 440-456.

[13] J. Iguchi-Cartigny, J.L.Lanet, "Developing a trojan applets in a Smart Card", Volume 6, Issue 4, **(2010).** pp 343-351.

[14] G. BARBU, G. Duc, Hoogvorst, Philippe, "Java card operand stack: fault attacks, combined attacks and countermeasures", proceeding of international conference of Smart Card Research and Advanced Applications. Springer Berlin Heidelberg, **(2011)**, pp. 297-313.

[15] G. Bouffard, J. Iguchi-Cartigny, J. L. Lanet, "Combined software and hardware attacks on the Java Card control flow", proceeding of international conference of Smart Card Research and Advanced Applications, Lecture Notes in Computer Science, vol. 7079 Springer,Berlin Heidelberg **(2011)** , pp. 283–296.

[16] G. Bouffard, J. L. Lanet, "Reversing the operating system of a Java based smart card", Journal of Computer Virology and Hacking Techniques, Volume 10, Issue 4, **(2014)**, pp 239-253.

[17] G. Barbu. "On the security of Java Card™ platforms against hardware attacks", PhD thesis.Grant-funded with Oberthur Technologies and TélécomParisTech, **(2012)**.

[18] G. Bouffard. "A Generic Approach for Protecting Java CardTM Smart Card Against Software Attacks", PhD thesis in Electronics.University of Limoges, **(2014).**