Recognition of Sensitive Patterns to the Fault Attack in the Java Card Application

Yahiaoui Chahrazed[1], Lanet Jean-Louis[2], Mezghiche Mohamed[1], and Tamine Karim[2]

[1] LIMOSE Laboratory, University of M'hamed Bouguara Boumerdes, Algeria
`yahiaoui.chehrazed@gmail.com, mohamed.mezghiche@yahoo.fr`
[2] XLIM Laboratory, University of Limoges, France
`jean-louis.lanet@unilim.fr, karim.tamine@unilim.fr`

Abstract. Fault attack represents one of the serious threat against Java Card security. It consists in physical perturbation of chip components that cause an unusual behavior in the execution of Java Card applications. This perturbation allows to introduce faults on Java Card application, with the aim to reveal a secret information that are stored in the card or grant an undesired authorization. This paper presents a methodology to recognize the sensitive code to the fault attack in the Java Card applications. It is based on concept from text categorization and machine learning.

Introduction

Nowadays, smart cards are considered small computers with limited capacity, allowing to run applications and store sensitive and confidential information (such as the PIN [3] and cryptographic keys) in a secure manner. Examples of application that use smart cards are credit card, electronic passport, health insurance card, pay tv, telephony SIM card, etc. Java Card is a kind of smart card that embed a virtual machine (called Java Card Virtual Machine or JCVM), which interpret application byte codes. However, smart cards have been mainly threatened by fault attack (also known fault-injection attacks). The principle of a fault attack is to modify the physical environment of the card in order to provoke an abnormal behavior of the component. It can target either the processor, the data/address bus or even the memory cells. Such physical perturbation can be caused by various tools such as a laser beam or a glitch generator [4]. Fault attacks have been mainly applied in the literature to the implementation of cryptographic algorithms[8]. Nevertheless, such attacks may have an impact on the whole software embedded on the card.

In this paper, our work focuses on the security of applications that run on Java Card platform by analyzing the ability of an application to become hostile, due to laser attack (fault attack with laser beam). Indeed, such attack can cause modification in a Java Card application (also called applet) at byte code level

[3] Personal Identification Number

to obtain an unusual behavior of the latter. Therefore, our work consists in extracting, in a Java Card application, the sensitive patterns (vulnerable code) that could generate mutant at the byte code level. Mutants are codes that have been modified and not detected by the embedded countermeasures. Thus, the idea is to reduce during the development phase the introduction of software pattern known to generate hostile applet. The problem of dangerous patterns recognition can be considered as a supervised classification problem, which is the action to assign patterns to predefined classes (dangerous or not).

The supervised classification is the action to assign an object represented as a vector of characteristics (features) to one of many prespecified classes. More explicitly, the overall process of classification scheme based on supervised learning is divided into two subsequent phases: training and testing. The training phase consists in classifying a set of objects (known as training set) by an expert. These labelled objects (called examples) are used to train the classifier by an appropriate machine learning algorithm. Next, during the testing phase, the trained classifier is tested using a test set (collection of new object that did not appear in the training set) to check the accuracy of the classifier. Thus, it is necessary to know the real class of the objects in the test set in order to compare their real class with the class that was derived by the classifier.

The overall organization of the paper is as follows. After the introduction, we present an overview on fault attack and another on text categorization from which we inspired to classify sensitive patterns. In section 2, we describe our proposition to classify sensitive patterns. The section 3 is devoted to the experimentation. The last section is conclusion and our future work.

1 Background

1.1 Fault Attack

Faults can be induced into a chip using physical perturbations (like: a power spike, the heat, a laser, a clock glitch, etc) in its execution environment. These errors can generate different versions of a program (a mutant version) by changing some instructions, interpreting operands as instructions, branching to invalid labels and so on. To prevent a fault attack to happen, we need to know what are its effects on smart cards. References [3][12] present taxonomy of fault models in detail. In our case, we choose the precise byte error model as the most realistic attack model. When an attacker physically injects energy in a memory cell to change its state and depending of the underlying technology, the memory physically takes the value 0x00 or 0xFF. If memory cells are encrypted the physical value becomes random according to encryption algorithm. Thus, we assume that an attacker can:

- Make a fault injection at a precise clock cycle (he can target any operation he wants),
- Only set or reset a byte to 0x00 or to 0xFF, or change this byte to a random value which cannot been predicted (random fault type),
- Target any memory cells (precise memory cell of a variable or register).

1.2 Defining a Mutant Code and Sensitive Pattern Code

To define a mutant code, we use an example presented in [6], which consists in debit method that belongs to a wallet Java Card applet. The user PIN must be validated prior to the debit operation. Table 1 presents the corresponding byte code representation.

```
private void debit(APDU apdu) {
 if ( pin.isValidated() ) {
    // make the debit operation
 }else {
   ISOException.throwIt(SW_PIN_
   VERIFICATION_REQUIRED);
 }
}
```

Table 1. Byte code representation before attack

Byte	Byte code representation
00 : 18	00 : aload_0
01 : 83 00 04	01 : getfield #4
04 : 8B 00 23	04 : invokevirtual #18
07 : 60 00 3B	07: ifeq 59
10 :	10 : ... ...
59 : 13 63 01	59 : sipush 25345
63 : 8D 00 0D	63 : invokestatic #13
66 : 7A	66 : return

An attacker wants to bypass the PIN test to make the debit operation. A fault on the cell containing the conditional test byte code changes the ifeq instruction (byte 0x60) to a nop instruction (byte 0x00). The resulting byte code is showed in Table 2.

```
private void debit(APDU apdu) {
  // make the debit operation
  ISOException.throwIt (SW_PIN_
  VERIFICATION_REQUIRED);
}
```

Table 2. Byte code representation after attack

Byte	Byte code representation
00 : 18	00 : aload_0
01 : 83 00 04	01 : getfield #4
04 : 8B 00 23	04 : invokevirtual #18
07 : 00	07: nop
08 : 00	08: nop
09 : 3B	09: pop
10 :	10 : ... ...
59 : 13 63 01	59 : sipush 25345
63 : 8D 00 0D	63 : invokestatic #13
66 : 7A	66 : return

When an attack changes an opcode byte, then it may change the number of following bytes used as operands: a shift in the instruction flow occurs which may remain shifted until it eventually recovers its normal flow (interpreting as an opcode, opcode of the original instruction flow after the shift of the flow induced by an attack).

Let us consider an example [2] of an attack targeting an instruction encoded with 3 bytes: X arg1 arg2. The first byte X contains the opcode, the two following arg1 arg2 are operands. Let us assume that X is replaced by Y. The consequences of this change depend on the number of bytes needed by Y:

1. X arg1 arg2 ; $\rightarrow$ Y arg1 ; arg2 if Y needs one operand byte, then arg2 is viewed as an opcode and the instruction flow is shifted (until possibly returning to the original instruction flow).
2. X arg1 arg2 ; $\rightarrow$ Y ; arg1 arg2 if Y has no operand byte then arg1 is viewed as an opcode. Depending on the number of bytes needed by arg1, arg2 is then either an operand or an opcode. The instruction flow has also shifted.
3. X arg1 arg2 ;$\rightarrow$ Y arg1 arg2 ; if Y needs two operand bytes then the whole instruction has changed but the instruction flow has not shifted.

The sequence of instruction from the fault injected until the recovery of the initial code is called mutant. The original code that has mutated is the sensitive pattern code. In our example, the mutant code is the sequence of instruction from instruction 07 to instruction 10 in table 2, and the pattern code is the sequence of instruction from instruction 07 to instruction 10 in table 1.

1.3 Text Categorization

Text categorization TC is the task of assigning a text documents to one or more predefined categories according to the documents' content. We can describe the TC process as consisting of four main phases: document representation, feature selection, classification algorithm and evaluation of results. The first phase in TC is to transform documents into a representation suitable for the classifier. The most widely used document representation is Vector Space Model[9]. A text document is represented as a vector of weighted term. This representation consists first to define the vocabulary ,i.e. the set of all distinct terms that occur in the training documents, and then each term in the vocabulary must be associated with a value (weight) which denotes the importance of this term in a text and its contribution to the semantics of document. The term may be identified either with the words occurring in the document called bag-of-words representation, or with a sequence of character or word called n-grams, that are extracted from a long string in a document. Thus there are several ways of determining the weight [1] such as Boolean weighting, Term frequency weighting (TF), Term Frequency Inverse Document Frequency (TF-IDF) weighting etc. But the major problem of this representation is high dimensionality of the vector space. Hence, there is a need to reduce the size of the vector space in order to improve the accuracy of classifiers. One of the most used technique for dimensionality reduction is feature

selection. The main idea of feature selection is to select a subset of the vocabulary terms. To this end, feature selection attempts to remove the terms that are considered irrelevant for classification. Various feature selection methods, such as document frequency, information gain and X^2 test (CHI) have been commonly used for reducing dimentionality of text document representations[11]. The classification phase needs to use machine learning algorithm to build classifier such as Naive Bayes classifier, decision tree, support vector machines and neural networks. The last phase in TC is to evaluate the effectiveness of a classifier, i.e. its capability of taking the right categorization decisions. An important issue of TC is how to measures the performance of the classifiers. Many measures have been used, like precision and recall, error, accuracy, F-measure, Micro-averaging and Macro-averaging etc. [1] [10]. These evaluation measures are computed by using global contingency table shown in table 3. Let us note that:
TP_i(true positives) is number of test documents correctly classified under c_i;
FP_i(false positives) is number of test documents incorrectly classified under c_i;
FN_i(false negatives) is number of c_i test documents incorrectly classified as non c_i;
TN_i(true negatives) is number of non c_i test documents correctly classified.

Table 3. The Global Contingency Table

Category set, $C = \{c_1, c_2, ..., c_{\|C\|}\}$		True class	
		Yes	No
Predicted class	Yes	$TP = \sum_{i=1}^{\|C\|} TP_i$	$FP = \sum_{i=1}^{\|C\|} FP_i$
	No	$FN = \sum_{i=1}^{\|C\|} FN_i$	$TN = \sum_{i=1}^{\|C\|} TN_i$

In this paper, we use the accuracy, False Positive rate (FP_{rate}) of the class $c_i \in C$ and False Negative rate (FN_{rate}) of the class $c_i \in C$, which are defined as:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

$$FP_{rate_i} = \frac{FP_i}{FP_i + TN_i}$$

$$FN_{rate_i} = \frac{FN_i}{TP_i + FN_i}$$

2 Methods

In this study, we present a methodology for sensitive pattern classification based on concepts from text categorization and supervised machine learning.

2.1 Dataset Creation

For the construction of the data set, we used an analysis tool called SmartCM [5]. The goal of this tool is to analyze the ability of a Java Card application to become hostile due to a laser attack. This tool uses a brute force process to generate all the mutants corresponding to a given application taking into account the model card. The smart card model means the embedded countermeasures (stack underflow, overflow, wrong local variable, wrong expected type etc.) and the nature of the memories (encrypted or not) used by the smart card. To do this, it modifies each opcode value either 0x00 or 0xFF, or from 0x00 to 0xFF according to the kind of memory. Then it evaluate the execution of this new code for each values of the opcode and if an embedded countermeasure detects the deviant behavior the mutant is rejected, else it is stored as a mutant. Thus it classifies the mutants into two categories dangerous or not, according to some security properties. But the major problem of this tool is the high time consumption especially when the card does not have countermeasures or presents weak countermeasures. Hence, the main goal in our study is to overcome this problem by using supervised learning techniques instead brute force process to recognize sensitive code.

The output of SmartCM corresponds to a set of mutants (code after attack) classified as either dangerous or not, taking into account the model of the card used. However, to build our data set we need to classify sensitive patterns code (the original code before attack) instead mutant code. Therefore, we modified the smartCM tool to recover the original code of each mutant generated. Once the patterns are recovered, they must be classified. To do this, we check for each pattern if there is at least one dangerous mutant generated from this pattern, so it is classified dangerous, else it is not.

Let us suppose that this classification is made while using two model cards. However, the classification of patterns as either dangerous or not depends on model card used. Therefore, for the generation of this data set we used a set of Java Card application at byte code level. Each application is used to generate the patterns classified as dangerous or not for each card model. Thus, we have a set of patterns generated from a set of application classified as dangerous or not dangerous for each card model. But for the same application, we have the possibility to assign the same pattern in two different class with different model card. For example, the same pattern is classified as dangerous in the model card 1, and not dangerous in the model card 2 because there are countermeasure detects it. We can distinguish four possible cases.

- Pattern classified dangerous in model card 1 and dangerous in model card 2 ($D_{m1}D_{m2}$)
- Pattern classified dangerous in model card 1 and not dangerous in model card 2 ($D_{m1}ND_{m2}$)
- Pattern classified not dangerous in model card 1 and dangerous in model card 2 ($ND_{m1}D_{m2}$)
- Pattern classified not dangerous in model card 1 and not dangerous in model card 2 ($ND_{m1}ND_{m2}$)

For this, we assume to have four classes instead of two, with the aim to include the information concerning model card.
Our Dataset consists of a set of labelled patterns - the set of patterns together with the respective class label ($D_{m1}D_{m2}$, $D_{m1}ND_{m2}$, $ND_{m1}D_{m2}$, $ND_{m1}ND_{m2}$). Furthermore we split this Dataset into two subsets: the training set and test set.

2.2 Pre-processing and Pattern Representation

To classify the patterns, we had to convert them into a vectorial representation. To do this, first we extracted the vocabulary: all distinct opcodes (op) appear in the entire training set of patterns, disregarding operandes, $V = \{op_1, op_2, ..., op_{|V|}\}$. Later a vector of weighted opcodes terms is created for each pattern. We used two different weight opcodes representation: boolean weighting and term frequency weighting. Let us take *pattern* a vector of weighted opcode terms, where $pattern = \{w_{op_1}, w_{op_2}, ..., w_{op_i}, ..., w_{op_{|V|}}\}$. In the binary representation, w_{op_i} is a binary value (0-1), where the value 1 represents the presence of the opcode op_i in the pattern and its absence is represented by 0. In the term frequency representation, w_{op_i} is the frequency count of ith opcode op_i in the pattern.

2.3 Classification

To classify the patterns, we used two naive Bayes models, Decision Trees with binary representation (B-DT) and Decision Trees with Term Frequency representation (TF-DT). Naive Bayes models are Multivariate Bernoulli Naive Bayes model (MBNB) and the Multinomial Naive Bayes model (MNB). We briefly describe the classification algorithms we used in this study.

Naive Bayes classifiers. The Naive Bayes classifier is a simple probabilistic classifier. The basic idea in Naive Bayes classifier, to assign a document d to one of a set of $|C|$ predefined categories $C = \{c_1, c_2, ..., c_i, ..., c_{|C|}\}$, is first to computes the posterior probability that document d belongs to each particular class c_i by the Bayes theorem as follows:

$$p(c_i|d) = \frac{p(d|c_i)\,p(c_i)}{p(d)} \tag{1}$$

and then assigns the document to the class with the highest probability value (maximum posterior probability). Note that $p(d)$ is constant (is the same for all classes) and can be ignored , thus d can be classified by computing

$$c^*(d) = \underset{c_i \in C}{argmax}\, p(c_i|d) = \underset{c_i \in C}{argmax}\, p(c_i)\,p(d|c_i) \tag{2}$$

The probabilities $p(c_i)$ and $p(d|c_i)$ are estimated from a training set. The category prior probability, $p(c_i)$, can be estimated as follows: $p(c_i) = \frac{N_i}{N}$
where, N_i is the number of training documents in class c_i, and N is the total

number of training documents.
The distribution of documents in each class, $p(d|c_i)$, cannot be estimated directly. In text classification, a document d is generally represented by a vector of $|V|$ terms (features), where $|V|$ denotes the size of the vocabulary V such as $V = \{t_1, t_2, ..., t_{|V|}\}$. Naive Bayes classifier assumes that all features are independent given the context of the class,i.e, the conditional probability of a feature given category is assumed to be independent from the conditional probabilities of other features given that category. This assumption simplifies the computation by reducing (2) to

$$\begin{aligned} c^*(d) &= \underset{c_i \in C}{argmax}\, p(d|c_i)p(c_i) \\ &= \underset{c_i \in C}{argmax}\, p(t_1, t_2, ..., t_{|V|}|c_i)p(c_i) \\ &= \underset{c_i \in C}{argmax} \prod_{j=1}^{|V|} p(t_j|c_i)p(c_i) \end{aligned} \tag{3}$$

There are several Naive Bayes models. The most popular are: the multivariate Bernoulli model (also called binary independence model) and the multinomial models [7]. In the first model, a document is represented by a vector of binary terms indicating which terms occur and do not occur in the document. This is called multivariate Bernoulli model because a document vector can be regarded as the outcome of multiple independent Bernoulli experiments. In the second model, a document is represented by the set of term occurrences from the document. This is called multinomial Naive Bayes model because the probability of a document vector is given by a multinomial distribution.

Decision Trees. Decision trees are the most widely used inductive learning methods. It is composed of three main components: nodes, branches and leaves. Each node denotes a test on a feature, each branch descending from that node corresponds to one of possible values for this feature and each leaf represent class label. The decision tree is constructed during the learning phase, it is then used to predict the classes of new examples. For the construction of the tree, decision tree algorithms like ID3 and C4.5 [14][15], use a top-down (i.e from the root to the leaves) recursive divide and conquer manner. The main idea is to select the best feature that divides the training set. This feature is used as the test at the decision node of the tree. A branch of this decision node is then created for each possible value of this feature. According to the values of this feature, the training set is partitioned. The same process is then repeated on each partitioned subset of the training set by considering all the features except that already selected. The process terminates when all the examples in current subset belongs to the same class and then a leaf node is constituted. For the selection of the best feature, each algorithm uses a feature selection measures such as information gain and gain ratio. An unknown example is classified by starting at the root node and following the tree down the branches until a leaf node representing the

class is reached. Each decision tree represents a rule set. These rules are of type: if condition then conclusion. In addition to the construction and classification phases, most decision tree algorithms include pruning. It consists in removing some branches that are considered useless for improving the performance of the tree in the classification, in order to avoid the problem of over-fitting.
In this study, we used J48, the Weka [13] version of the C4.5 algorithm [15].

2.4 Evaluation Measures

To evaluate the effectiveness of the classifiers, we used accuracy measure, which is the rate of correctly predicted categories. Moreover to known how well the classifier can recognize dangerous patterns, we used False positive rate (FP_{rate}) and False negative rate (FN_{rate}). These measures are described in subsection 1.3.

3 Experimental Results

The Data set consists of 4096 patterns that contain 87 distinct opcodes. Note that we used in this experiment ten Java card applications and two card models: encrypted memory with all countermeasures and non encrypted memory without any countermeasures. SmartCM tool classified these patterns into three classes: $D_{m1}D_{m2}$, $ND_{m1}D_{m2}$, $ND_{m1}ND_{m2}$ and no pattern is classified as $D_{m1}ND_{m2}$. Table 4 shows the number of patterns for each class. This experiment is run on Intel Core i7 CPU 3.4 GHz with 12 GB RAM. When we compared the time taken to classify these patterns which is about 10 hours and the training time of the classifiers(shown in table 5), it seems that SmartCM is the most expensive.

Table 4. Documents number used for each class

Class	Number of patterns
$D_{m1}D_{m2}$	41
$ND_{m1}D_{m2}$	751
$ND_{m1}ND_{m2}$	3304
Total	4096

In this experimental study, we split the data set into two set, two-thirds of the data set represent the training set, and the remaining one-third constitute the test set. Table 5 gives the accuracy results and the training time of each classifier.

Table 5. Classifiers accuracy performance and the training time

Classifier	Accuracy(%)	Training time (seconds)
MBNB	74,23	0,02
B-DT	84,56	3
MNB	63,39	0,03
TF-DT	83,99	4

After analyzing accuracy results, we found that classifiers with Binary representation (MBNB and B-DT) outperformed classifiers with Term Frequency representation (MNB and TF-DT). Thus the decision tree outperformed Naive Bayes models. However, in our case study we are interested to the detection of dangerous patterns. For this, it is important to indicate whether the decision tree classifier is good at classifying the both classes: $D_{m1}D_{m2}$ and $ND_{m1}D_{m2}$. Information of this kind could not be obtained from the accuracy alone, and this is why analyses of FP_{rate} and FN_{rate} results were also included. Figures 1 and 2 show the FP_{rate} and FN_{rate} results of each class. The FP_{rate} is used to indicate the proportion of not dangerous patterns ($ND_{m1}ND_{m2}$) wrongly classified as dangerous patterns ($D_{m1}D_{m2}$ and $ND_{m1}D_{m2}$). The FN_{rate} is the most interesting information to indicate the proportion of dangerous patterns ($D_{m1}D_{m2}$ and $ND_{m1}D_{m2}$) wrongly classified as not dangerous patterns ($ND_{m1}ND_{m2}$).

In figure 1, we note that class $ND_{m1}ND_{m2}$ has the highest FP_{rate} result compared with other classes ($D_{m1}D_{m2}$ and $ND_{m1}D_{m2}$) for all classifiers. This indicate that the $D_{m1}D_{m2}$ and $ND_{m1}D_{m2}$ classes are highly overlapped with $ND_{m1}ND_{m2}$ class and this means that patterns in class $D_{m1}D_{m2}$ and $ND_{m1}D_{m2}$, have been classified as $ND_{m1}ND_{m2}$ class. Thus this overlap is most remarkable in DT which contains the highest FP_{rate} result in class $ND_{m1}ND_{m2}$.

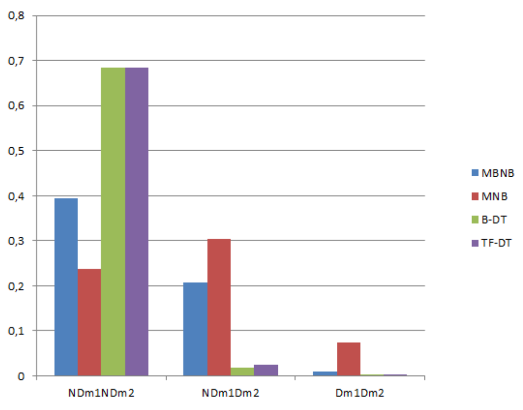

Fig. 1. FP_{rate} results of each class

If we analyze the results of FN_{rate} in figure 2, we find that the two classes $D_{m1}D_{m2}$ and $ND_{m1}D_{m2}$ have the highest FN_{rate} values compared to class $ND_{m1}ND_{m2}$, for all classifiers. This means that the classes $D_{m1}D_{m2}$ and $ND_{m1}D_{m2}$ are poorly recognized compared to class $ND_{m1}ND_{m2}$. Thus, we note that DT contains the highest FN_{rate} values compared to NB for classes $D_{m1}D_{m2}$ and $ND_{m1}D_{m2}$, and the lowest value for other class. DT has well classified $ND_{m1}ND_{m2}$ class compared to NB, but NB has well classified $D_{m1}D_{m2}$ and $ND_{m1}D_{m2}$

classes compared to DT. Since in our problem we are interested in the recognition of the dangerous classes, we can conclude that NB outperform DT. Unfortunately, even the results given by NB are unsatisfactory. We note that about 45% of FN_{rate} for $D_{m1}D_{m2}$ and $ND_{m1}D_{m2}$, which is a highest value. We suggest that this is due to the class imbalance problem also known imbalanced data set in which the distribution of the classes varies (see table 4). It has received considerable attention in areas such as machine learning and pattern recognition. There are many real-world applications that are faced with class imbalance problem such as medical diagnosis. Typically it occurs when there are significantly more examples from one class (majority class) relative to other classes (minority class). The minority classes are usually the most important classes. In such cases the classifier tends to misclassify the examples of the less represented classes.

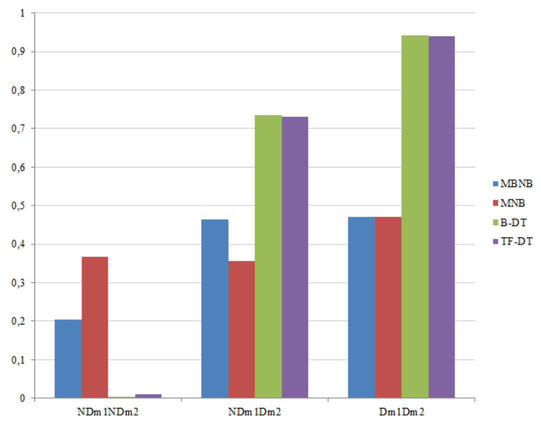

Fig. 2. FN_{rate} results of each class

Conclusion

In this paper, we presented a new methodology for the recognition of dangerous patterns based on text categorization and machine learning techniques. We have used DT and NB classifiers with two modes representation: Binary and TF representation. Our goal is to determine the best term representation and which classifier is more accurate at detecting dangerous patterns. We have used accuracy measure to evaluate performance of the classifiers. The results of this evaluation have shown that Binary representation is better than TF representation and DT outperform NB. Therefore we have used FP_{rate} and FN_{rate} measures to indicate whether the DT classifier is more adept at classifying dangerous patterns. The results have shown that NB has well recognized patterns assigned to dangerous classes, compared to DT classifier. Unfortunately, even the results given by NB are unsatisfactory. This is due to the imbalanced data set.

Our future work will be first to do an experiment in which we investigate the class imbalance problem and also to evaluate the opcode n-gram representation. The Opcode n-gram is a sequence of consecutive n opcodes extracted from each pattern in the training set. Each pattern is represented as a vector of weighted n-gram opcode. We use the TF-IDF weighting term which is widely used in TC. Different experiments have been done to identify the best opcode n-gram size, the feature selection methods (frequency, information gain, mutual information and X^2 test (CHI)) and the classifiers (K-nearest neighbor, Support Vector Machines and Neural Networks).

References

1. Aas, K., Eikvil, L.: Text categorisation: A survey. Technical report, Norwegian Computing Center, 1999.
2. Berthome, P., Heydemann, K., Kauffmann-Tourkestansky, X., Lalande, J-F.: High level model of control flow attacks for smart card functional security. Seventh International Conference on Availability, Reliability and Security, pp. 226, 2012.
3. Blomer, J., Otto, M., Seifert, J.P.: A new CRT-RSA algorithm secure against Bellcore attacks. In: Proceedings of the 10th ACM conference on Computer and communications security, pp. 311-320, 2003.
4. Giraud, C., Thiebeauld, H.: A Survey on Fault Attacks. In: Smart Card Research and Advanced Applications, CARDIS 2004.
5. Machemie, J-B., Mazin, C., Lanet, J-L., Cartigny, J.: SmartCM A Smart Card Fault Injection Simulator. Information Forensics and Security (WIFS), IEEE 2011
6. Machemie, J.-B., Lanet, J.-L., Bouffard, G., Poichotte, J.-Y., Wary, J.-P.: Evaluation of the Ability to Transform SIM Applications into Hostile Applications, CARDIS'11, pp.1-17, Leuven, Belgium, 14-16 September 2011.
7. McCallum, A., Nigam, K.: A Comparison of Event Models for Naive Bayes Text Classification. AAAI/ ICML-98 Workshop on Learning for Text Categorization, 1998.
8. Moulin, H., Bihame, E., Shamir, A.: Differential Fault Analysis of Secret Key Cryptosystem. In: Advances in Cryptology. CRYPTO'97, pp. 513-525, Springer 1997.
9. Salton, G., Wang, A., Yang, C.S.: A Vector Space Model for Automatic Indexing. Communications of the ACM, Vol. 18, pp. 613 -620, 1975.
10. Sebastiani, F.: Machine learning in automated text categorization. ACM Computing Surveys, pp.1-47, 2002.
11. Yang, Y., Pedersen, J.O.: A Comparative Study on Feature Selection in Text Categorization. In: Proceedings of ICML-97, 14th International Conference on Machine Learning, pp. 412-420, Morgan Kaufmann, San Francisco, US, 1997.
12. Wagner, D.: Cryptanalysis of a provably secure CRT-RSA algorithm. In: Proceedings of the 11th ACM conference on Computer and communications security. pp. 92-97, ACM New York, 2004.
13. Witten, I.H., Frank, E.: Data Mining: Practical Machine Learning Tools and Techniques. 2nd edn. Morgan Kaufmann Publishers, Inc., San Francisco, 2005.
14. Quinlan, J. R.: Induction of decision trees. Machine Learning, 1986.
15. Quinlan, J. R.: C4.5: Programs for machine learning. Morgan Kaufmann Publishers, 1993.