International Journal of Security and Its Applications
Vol.8, No.1 (2014), pp.373-388
http://dx.doi.org/10.14257/ijsia.2014.8.1.35

# Security Vulnerabilities Tests Generation from SysML and Event-B Models for EMV Cards

Noura Ouerdi[1], Mostafa Azizi[2], M'Hammed Ziane[3], Abdelmalek Azizi[4], Jean-louis Lanet[5] and Aymerick Savary[6]

*[1,3,4]Department Mathematical and Computer, Lab. ACSA, Faculty of Sciences, Mohammed First University, Oujda 60000, Morocco*
*[2]Department Computer, Lab. MATSI, ESTO, Mohammed First University, Oujda 60000, Morocco*
*[5]XLIM/DMI/SSD,87 street of Isles, 87000 Limoges, France*
*[6]GRIL, IT department, Sherbrook University*, Quebec, Canada
*noura.ouerdi@gmail.com[1], azizi.mos@gmail.com[2], ziane12001@yahoo.fr[3], abdelmalekazizi@yahoo.fr[4], jean-louis.lanet@unilim.fr[5], aymerick.savary@gmail.com[6]*

***Abstract***

*The Model Based Testing (MBT) is an original approach where test cases are automatically generated from the specifications of the system under tests. These specifications take the form of a behavioral model allowing the test generator to determine, on the one hand, the possible and relevant execution contexts. On the other hand, to predict the effects of these executions on the system. This paper proposes new methodology to generate vulnerability test cases based on SysML model of Europay-Mastercard and Visa (EMV) specifications. Our main aim is to ensure that not only the features described by the EMV specifications are met, but also that there is no vulnerability in the system. To meet these two objectives, we automatically generated concrete tests basing on SysML models. Indeed, this paper highlights the importance of modeling EMV specifications. We opted for the choice of SysML modeling language due to its ability to model Embedded Systems through several types of diagrams. In our work we used state machine diagram to generate vulnerability test cases for a secure and robust system.*

***Keywords:*** *EMV, Model-Based Testing, Event-B, SysML, Smart card, Vulnerability*

## 1. Introduction

Although the magnetic stripe technology provides safe security, the use of a smart card into a terminal with a smart card reader helps to increase the security of operating system already safe, since it validates both the smart card and the identity of its holder. The cardholder authentication is assured by the Personal Identification Number (PIN) for the approval of transactions. So, according to the smart card issuers, it is virtually impossible to clone a smart card, unlike magnetic stripe cards. However, a current research [7] has shown that it is possible to intercept information transmitted between smart card and terminal, which make cloning possible. Therefore and given the widespread use of smart cards dedicated to payment (EMV cards), it is crucial to secure them. Our contribution is around this context.

The adoption of traditional or mobile payment system with smart card is based on the confidence we have in the underlying computer system. Some recent attacks, like *Chip and PIN is broken* [8], showed a misconfiguration could lead to make unauthorized transaction. This attack allows criminals to use a genuine card to make payment without knowing the card's PIN and remains undetected over the banking network. In order to avoid such attack, we would try to ensure that each element of the system is correct. This guarantee has generally high cost. That's why we propose, in this paper, a new methodology that automates this process to increase quality with a reasonable cost. Our goal was to ensure security of EMV card earlier, that means that we can verify the system in the design phase before moving to the implementation phase. The verification process did not cover only the satisfaction of the EMV specifications but also the verification that the system has no flaw or vulnerability that could lead to a possible attack. To do this we adopted a MBT approach. We conducted therefore this verification through the generation of vulnerability tests based on SysML models. Neglected long enough, this approach is now the subject of serious studies and investigations including our own. Unfortunately, most of researches on tests generation based on models have been conducted only to verify that authorized behavior by a model is also authorized by its implementation. In this paper, we aim to generate a set of tests corresponding not only to conformance behavior, but also to prohibited behavior by the specifications presenting vulnerability tests. These tests should be rejected by the EMV card. The contribution of our research is therefore to identify possible vulnerabilities and not only to ensure the compliance of the system.

The paper is organized as follows: the second section presents the state of the art of the MBT approach. The third section highlights the application domain which is the EMV infrastructure and provides a general context of Europay-Smartcard-Visa specifications. The fourth section lists the various tools, language and plugins used to ensure our new approach. In the fifth section, we explain our new methodology of vulnerability tests generation. The sixth section presents in detail all of the steps to ensure our approach of tests generation based on SysML models.

## 2. State of the Art

### 2.1. Model-Based Testing

The principle of Model-Based Testing (MBT) is to compare the behavior of an explicit model with that of a system under test (SUT). This assumes that the model is valid. This task is not difficult in comparison to validation of the implementation itself which is, in the most of the time, a consuming task and requires a real cost. Once the model is considered valid, it is used for automatic test cases generation. According to [20], MBT process is divided into five stages:

- Modeling: definition of an abstract model for the test system,
- Production: generation of test cases from the defined abstract model,
- Realization: Transforming abstract test cases into executable test cases in the SUT,
- Running: run the tests on the system,
- Analysis: study of results.

There are several solutions around the MBT approach. These solutions are mainly distinguished by the used tools for each of the five phases listed above and the nature of the SUT.

In 2012, Roberto [21] published his experiences about creating models of EMV standard using B-Method, a formal method developed by Jean Raymond Abrial and testing its models using JTorX language. He created four models with B-Method. For example, in the first model, the messages exchanged between the card and terminal were treated as abstract messages in which only the type of the request is considered, *i.e.*, if the message is sent from the terminal or from the card. He also left the tests with JTorX for future work.

Currently, several researches focus on EMV modeling including Joeri's model [1]. Indeed, he presented a formal model of EMV protocol suite in F# language which can be analyzed using the protocol verification tool ProVerif [10] with FS2PV [11]. Although that F# model details the EMV transaction, it is not yet confirmed that this model would be useful for MBT of terminals and cards. In the same context, Aarts's team [9] has worked on formal models of bank cards. Several types of banks and cards are considered; Dutch bank, German bank, Swedish bank and MasterCard credit cards in order to derive the differences between them. It is possible that specificities of each bank could influence safety. However, this is not the goal of our work which is articulated around the EMV specifications regardless of the type of bank.

There are other tests techniques for security vulnerabilities as fuzzing technique [12]. We will study this technique as future work in order to compare it with our new methodology described in this paper. Moreover, there are research works that use this technique as Lancia's work [18]. Lancia guided his technical choice to a fuzzing framework. This testing technique in black box is traditionally used for finding vulnerabilities in implementations of network protocols.

## 2.2. Limitations

Generally, the MBT method for old works deals with SUT testing functionality as a black box. In the EMV field, a few works focused on the tests generation from models for EMV protocol, we extracted their limitations as follows:

- Incomplete work as is the case for Roberto's work [21] that has not yet been able to enter the testing phase with JTorX tool,
- Very few public works that focused on the EMV protocol, possibly due to its complexity and large specification and most of them are more interested in formal modeling of a portion of EMV specifications. This is the case for Joeri's work[1],
- Regarding the research work for vulnerabilities detection using fuzzing techniques, we cited Lancia's work [18]. Lancia realized that for security vulnerabilities, fuzzing techniques are not inherently more efficient than conventional testing techniques. Therefore, fuzzing tests appear complementary to conventional tests approaches.

For All these reasons, we thought to propose a new methodology that starts from a more abstract SysML model whose goal is to achieve, not only functional tests, but also vulnerabilities tests of an EMV card. We limited the work field within an EMV transaction between terminal and card in order to properly master our test context.

## 3. Application Domain: The EMV Infrastructure

EMV stands for Europay, MasterCard and Visa is a standard for inter-operation of integrated circuit card (ICC) for credit and debit transactions. There are other standards like ISO/IEC 7816 [5] for contact cards which is followed by all smartcards. The major benefits of moving to smartcards based on credit card payment systems are to improve security and to allow for multiple applications, namely credit and debit card application, to be held on a card. Today, more than 36% of total cards and 65% of deployed terminals are based on the EMV standard [8].

It is true that the EMV specifications are complex and stalls on four large books [6] which are: Application Independent ICC to terminal (book1), Security and Key Management (book2), Application Specification (book3) and Cardholder, Attendant, and Acquirer (book4).

In our work, we are interested more in book2 and book3. Indeed, we thought about modeling EMV specifications in terms of EMV transaction between a card and terminal. The strength of our model is that it will be confirmed with the generation of vulnerability test suites. So, before moving to modeling process, we should explain the steps of an EMV transaction between an EMV card and terminal.

### 3.1. APDU Command

An APDU command consists of a class byte (CLA), instruction byte(INS) of the Command Message, parameter 1 (P1), parameter 2 (P2), length of command data field (Lc) and length of expected data (Le).

The table 1 shows the Command APDU structure:

**Table 1. Command APDU Structure [6]**

| | | | | | | |
|---|---|---|---|---|---|---|
| CLA | INS | P1 | P2 | Lc | Data | Le |
| Mandatory Header | | | | Conditional Body | | |

While an APDU Response is composed of Data and status words SW1 SW2. The Status Words SW1 and SW2 indicate the success or failure of the command APDU execution. In the case of success, the returned SW1SW2 is equal to “9000” i.e, SW1=”90” and SW2=”00”. The APDU response components are described by the table 2.

**Table 2. Response APDU Structure**

| | | |
|---|---|---|
| Data | SW1 | SW2 |
| Body | Trailer | |

### 3.2. EMV Transaction

To perform a successful and complete EMV Transaction, there are three main phases. The first one is devoted to initialization of payment transaction. In this step, the task of terminal includes:

- Application Selection: the terminal sends SELECT APPLICATION command to select the wanted applet present in the card using its AID.

- Transaction Initialization using the GET PROCESSING OPTIONS command. It makes the card ready to exchange data with the terminal.
- Records Reading by sending a sequence of READ RECORD or GET DATA commands in order to read the public information of the card. This step is required to complete the transaction and store the resulting data for future use.

In the second phase, the terminal performs the card and cardholder authentications as follows:

- Card Authentication: There are two types of card authentication: Static authentication and dynamic authentication. For static authentication, the terminal signs static data and put them into the card using a public key to prove the card legitimacy. While dynamic authentication allows terminal to authenticate the card dynamically. So, the card must renew the signature for each transaction. Then, the signature remains unique for one transaction.
- Cardholder authentication: the cardholder enters his Personal Identification Number (PIN) through the terminal which sends this PIN code to the card using VERIFY command.

The third phase represents the decision provided by the terminal in terms of cryptogram generation:

- Cryptogram generation: since authentication of both of the card and cardholder are performed, the necessary parameters are initialized. The terminal decides to send GENERATE APPLICATION CRYPTOGRAM to the card. This command presents the heart of EMV transaction. The data field of GENERATE AC command contains Transaction’s amount, currency and terminal capabilities. The card informs the terminal of its decision concerning the transaction acceptance: online or offline acceptance or rejection of the transaction.
- Transaction completion: the terminal sends the second GENERATE AC command, and the card returns the final decision: accept or abort the transaction.

## 4. Tools and Method

**SysML** Language: Our work begins with the modeling of EMV specifications. We chose the SysML language [13] because it offers the possibility to model system requirements which will help us to design the system. In addition, SysML is more appropriate for Embedded Systems such as smart card, it allows us to model both the hardware and the software of the system and contains less complex theoretical concepts which make the models more understandable for a system engineer.

**Rodin Platform** is an Eclipse-based IDE for Event-B which provides effective support for refinement and mathematical proof.

**UML-B** [14] is a plug-in for RODIN toolkits implemented by Eclipse Modeling Framework (EMF)[ Abrial, 1996]. UML-B has its own meta-model and provides tool support, including drawing tools based on UML and a translator to generate Event-B models [15]. When the UML-B drawing is saved and has no error, the translator generates automatically the corresponding Event-B model.

**ProB** [2] is a plugin for RODIN representing an animator and model checker for the Event-B method. It allows an automatic animation of Event-B specifications. The animation's facilities allow users to gain confidence in their specifications. ProB supports also Event-B method and it can be integrated in Rodin platform to animate models and easily generate domain specific graphical visualizations. Its model checker and constraint-based checker help to detect various errors in Event-B specifications.

**Event-B** method is derived from B method. It keeps the B-method concepts and adds the event concept. Event-B models present well-defined syntax, concepts and semantics and it is possible to test them by proving that transitions made during the software process are correct.

**JDOM** is an API for XML documents. It has been created specifically for the Java programming language because it has been optimized for Java.

**Opal** [16] is an open-source Java library that complies with the Global Platform specification. It provides the support of Java Card applets and smart card management. This library presents a good tool to exchange data with Java Card. During our tests, we used this library to simulate a terminal and communicate with smartcard via APDU Commands.

## 5. Our new Methodology of vulnerabilities detection

### 5.1. Practical vulnerabilities of EMV Cards

The first vulnerability related to EMV protocol concerns the SDA method. SDA cards are vulnerable because of the lack of a mechanism to update signed data for each transaction, which means that this data can be copied. During an SDA transaction the card encrypts, using its key shared with the issuer, some transaction data. This signed message is forwarded to the issuer for authentication. As a result, cloned card could be identified whenever the card is presented in an on-line terminal. Since terminals do not have access to this key, they will simply forward messages to the issuer for verification. If the terminal is off-line, then the message cannot reach the issuer and the false transaction might be accepted [4].

Among the most famous stages of EMV transaction authorization is the PIN verification. An attacker, who can modify the communication between cards and terminal, can trick into believing that the PIN verification has been performed successfully by responding with the status word value "9000" to the VERIFY command APDU without sending the PIN to the card. The attack allows any PIN code to be entered and then accepted. Since the inserted dummy PIN never gets to the card, this card will believe that the terminal did not support PIN verification step. As a result, it has skipped cardholder verification step or used a signature instead [8].

For the first vulnerability, the potential solution would be to eliminate SDA cards and use only DDA cards especially since DDA cards are becoming cheaper. If DDA smart card is used in a transaction, it will be asked to sign uniquely each transaction details. For the second cited vulnerability, the *Chip and PIN attack* [8] could cause skipping of a very important security step which is cardholder verification. To avoid such vulnerability, we propose new methodology for generating vulnerabilities tests in terms of APDU commands execution order based on SysML model.

### 5.1. Our New Methodology

In our research, we are interested in generation of security and vulnerability tests which aim to ensure that the behavior rejected by the model is also rejected by its implementation. Our goal is to test using an abstract model that an EMV card implementation has no vulnerability. In our approach, we have only EMV specifications, we tried to identify security vulnerabilities which can be corrected later. To do this, we started by modeling EMV transaction using an informal modeling language namely SysML language. We chose SysML language because it provides concepts and tools that make easier expression of requirements thanks to its diagrams. Indeed, starting with formal models is usually a complex task and requires learning and mastering of the formal method. Therefore, we modeled, in the first time, the EMV transaction using state machine diagram, then, we proceeded to the generation and validation of Event-B model. Then, using this model, we generated a series of xml files containing invalid abstract tests of modeled EMV specifications using VTG tool [17]. We send these invalid tests to an EMV card and save the APDU responses on the card in terms of status words SW1SW2. The acceptance on an invalid test, otherwise SW1SW2 equals to "9000", means that vulnerability had been detected. Whereas if status word SW1SW2 is not equal to "9000", the card behavior, in this case, is normal and therefore there is no security vulnerability related to the test. Finally, the results in terms of APDU commands and APDU responses applied to the EMV card during real tests executions are saved in XML files which present the vulnerability tests.

## 6. Results and Discussions

### 6.1. Modeling EMV Transaction using Machine State Diagram

In order to build our machine state diagram, we need set of formal definitions. A state machine is a 6-tuple M = (S, E, A, d, dtr, dst); where S is a set of states, E is a set of events, A is a set of actions, and d is a transition relation $d : S \times E \rightarrow S$. A transition is labeled by the event causing it. For example, the transition d (s1, e) = S2 means that s1 is the source state and s2 is the target state. A state machine contains relations dtr and dst connecting respectively actions to the transitions and states. The diagrammatic presentation of a state machine is a statechart diagram [3]. A guard condition can be attached to a transition. It represents a Boolean expression which defines conditions under which the transition is able to fire. In our work, we studied EMV specifications in terms of EMV transaction. Then, we tried to determine the mainly states and transitions for our machine state diagram. These states and transitions are explained in details in the section 6.1 of this paper.

After a detailed study of EMV specification [6], especially the specifications of an EMV transaction which are described briefly in Section 3, we modeled these specifications using SysML models including the machine state diagram. States represent the system status of the Card-Terminal system during an EMV transaction. While transitions are the APDU commands sent by the terminal to change the status of our system. We specified for each transition (APDU Command) the appropriate parameters, guards and actions. For example, the transition from the *Initialization* state to *Application_Selected* state requires that the terminal sends the SELECT_APPLICATION APDU command successfully to the card. The machine state diagram defined in figure 1 designs an EMV transaction.

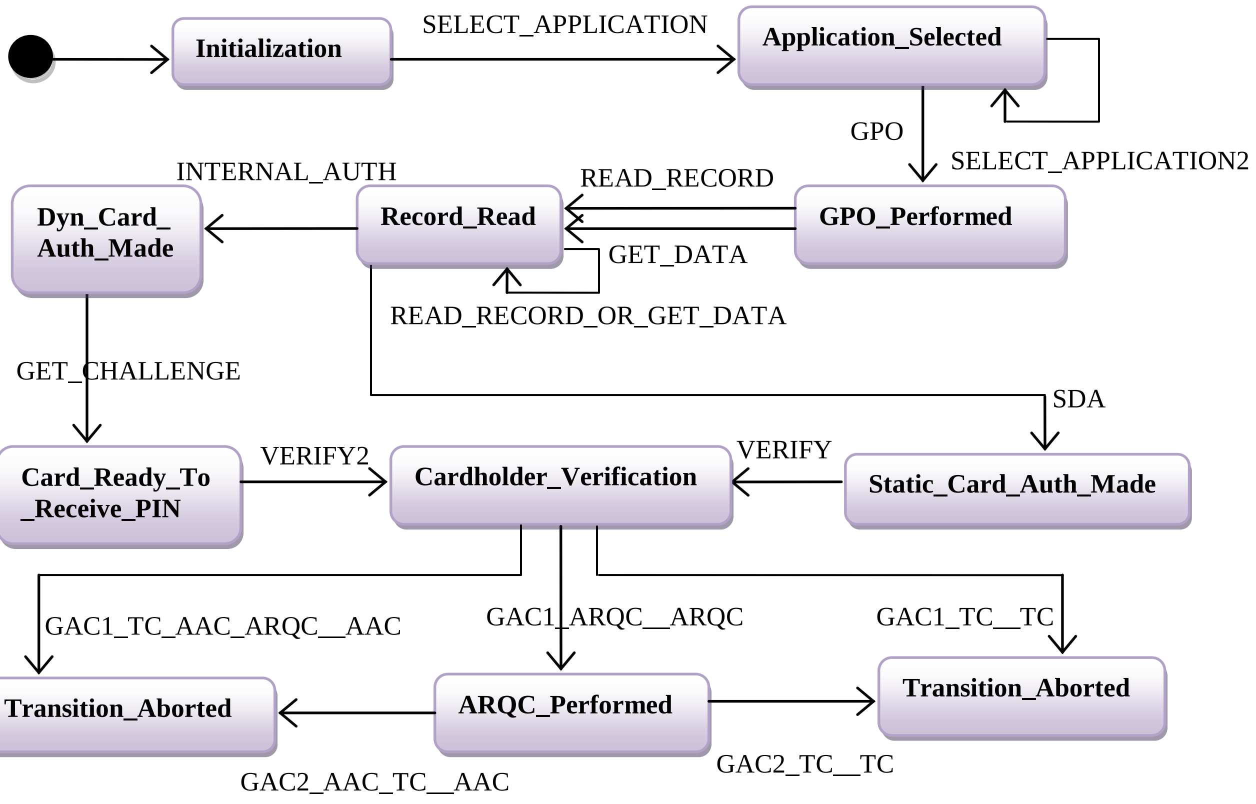


**Figure 1. Machine State Diagram to model EMV Transaction**

The following transitions occur in the diagram of the figure 1:

- ***SELECT_APPLICATION*** and ***SELECT_APPLICATION2*** present the same APDU command of application selection except that the source states are not the same; the source state of SELECT_APPLICATION is the *Initialization* state while the source state of SELECT_APPLICATION2 transition is *Application_Selected.*

- ***GPO*** transition means GET PROCESSING OPTIONS APDU command.

- ***READ_RECORD*** and ***GET_DATA*** transitions represent the READ RECORD and GET DATA APDU commands respectively.

- ***READ_RECORD_OR_GET_DATA*** transition replaces two transitions that start from the Record_Read state namely READ_RECORD and GET_DATA transitions.

- ***INTERNAL_AUTH*** transition is associated to *INTERNAL AUTHENTICATE* APDU command used when the card authentication is dynamic. *SDA* transition presents the case of static authentication.

- ***GET_CHALLENGE*** transition means the GET CHALLENGE APDU command to request the random number

- ***VERIFY*** and ***VERIFY2*** transitions refer to VERIFY APDU command. The first transition is intended for plaintext PIN verification while the second one is related to enciphered PIN verification because it is preceded by GET_CHALLENGE transition.

- ***GAC1_TC_AAC_ARQC__AAC***: GAC1 means the first call of GENERATE AC command. It takes as parameter either TC, AAC or ARQC cryptogram and returns AAC cryptogram

- ***GAC1_ARQC__ARQC***: the first call of GENERATE AC command with ARQC cryptogram argument. It returns ARQC cryptogram.

- ***GAC1_TC__TC***: the first call of GENERATE AC command. TC cryptogram as argument and as return value TC.

- ***GAC2_TC__TC***: the second call of GENERATE AC command. TC cryptogram as argument and as return value TC.

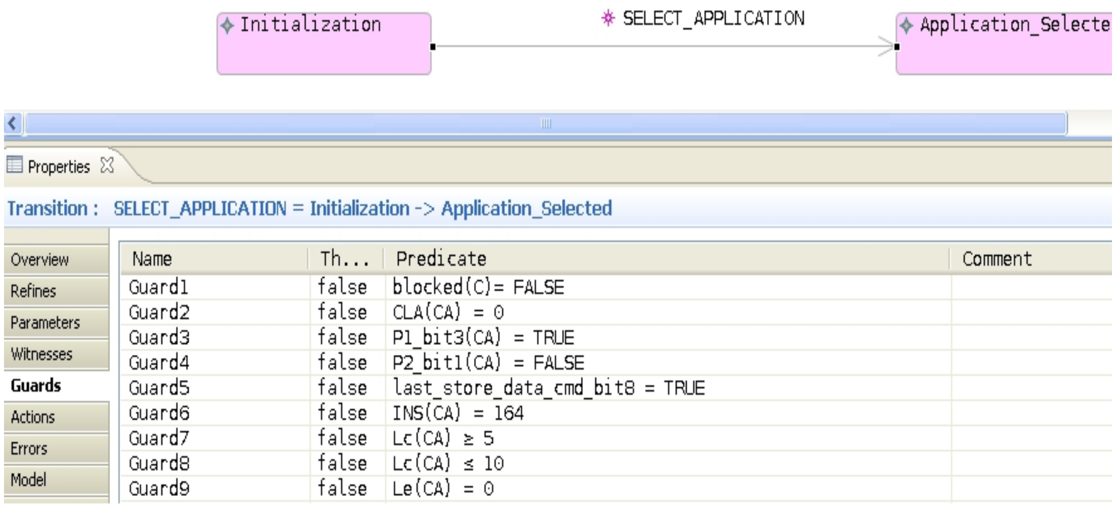
- ***GAC2_AAC_TC__AAC***: the second call of GENERATE AC command. It takes as parameter AAC or TC cryptogram and returns AAC cryptogram

## 6.2. Generation of Event-B Model

Once the behavior of the EMV card is already modeled through the machine state diagram, we generate the related Event-B model. So, how can we generate the Event-B model from our state machine? To answer this question, we should know that a state transition in the machine state diagram represents an event with behavior associated with the change of states, from a source state to a target one. Each transition is generated as an event. Additional guards and actions can be attached to the transition in the *Properties* tab of UML-B interface to describe the events' behaviors.

Using UML-B plugin provided by Rodin platform, we had the possibility to generate Event-B model from the machine state diagram by transforming the several transitions of the machine state diagram to events, keeping all specified guards and actions. By this operation, we generated Event-B model which is equivalent to full specifications modeled with our SysML diagram.

The generated Event-B model is spread over 500 lines of code including parameters, guards and actions that is why we could not put it in this paper. These parameters, guards and actions are specified for each transition of the machine state diagram under Rodin platform.To explain more how to generate the Event-B model, we take the example of the SELECT_APPLICATION transition. Using the UML-B interface of Rodin platform, we specify the parameters C and CA instances respectively of Card and Command_APDU classes. Then we specified the guards such as 'blocked (C)= FALSE' to require that the card is not blocked. There are also other guards that cover the elements constituting the structure of the SELECT_APPLICATION command as 'CLA(CA)=0', 'INS(CA)=164', *etc*. The Figure 2 presents how the guards are associated to SELECT_APPLICATION transition.



Transition : SELECT_APPLICATION = Initialization -> Application_Selected

| Name | Th... | Predicate | Comment |
|---|---|---|---|
| Guard1 | false | blocked(C)= FALSE | |
| Guard2 | false | CLA(CA) = 0 | |
| Guard3 | false | P1_bit3(CA) = TRUE | |
| Guard4 | false | P2_bit1(CA) = FALSE | |
| Guard5 | false | last_store_data_cmd_bit8 = TRUE | |
| Guard6 | false | INS(CA) = 164 | |
| Guard7 | false | Lc(CA) ≥ 5 | |
| Guard8 | false | Lc(CA) ≤ 10 | |
| Guard9 | false | Le(CA) = 0 | |

**Figure 2. Guards Associated to SELECT_APPLICATION Transition**

The generated Event-B model for SELECT_APPLICATION transition is as follows:

```
event SELECT_APPLICATION
    any C CA
    where
      @C.type C ∈ Card
      @CA.type CA ∈ Commande_APDU
      @Statemachine1_isin_Initialisation Statemachine1=
      Initialisation
      @SELECT_APPLICATION.Guard1 blocked(C)= FALSE
      @SELECT_APPLICATION.Guard2 CLA(CA) = 0
      @SELECT_APPLICATION.Guard3 P1_bit3(CA) = TRUE
      @SELECT_APPLICATION.Guard4 P2_bit1(CA) = FALSE
      @SELECT_APPLICATION.Guard5 last_store_data_cmd_bit8 = TRUE
      @SELECT_APPLICATION.Guard6 INS(CA) = 164
      @SELECT_APPLICATION.Guard7 Lc(CA) ≥ 5
      @SELECT_APPLICATION.Guard8 Lc(CA) ≤ 10
      @SELECT_APPLICATION.Guard9 Le(CA) = 0
    then
      @Statemachine1_enterState_Application_Selected Statemachine1 ≔
      Application_Selected
  End
```

### 6.3. Validation of Event-B model

Since the formal model is successfully generated, we verified guards, conditions, parameters and context of this model. We moved to animating model with ProB. Here, ProB plays the role of an animator of Event-B specifications. It generates and displays examples when it discovers a violation of an invariant. Since ProB supports automated consistency checking of Event-B machines via model checking, we used it to traverse all the reachable states of our machine. So, ProB helps us to ensure sophisticated debugging, testing and validating of our Event-B model.

## 6.3. Generation of Abstract Tests

To generate abstract test cases, we used the Vulnerability Tests Generator (VTG) that we implemented. VTG tool is an implementation of the process proposed in [17]. This tool is flexible; it takes as input a valid Event-B model and generates associated abstract test cases. VTG uses Rodin for the manipulation of models and ProB for the extraction of abstract tests as XML files. The principle of this tool is based on the negation of all conditions and guards in order to be sure to cover all cases prohibited by the modeled specifications. In this paper, we interested in the receipt order of APDU commands by EMV card. Otherwise, our objective is to detect system vulnerabilities exploited by a potential attacker who wants to skip a step in the EMV transaction. For example, according to the EMV specifications, the cardholder authentication should be preceded by other steps such as application selection, transaction initialization, data read and card authentication. Our vulnerability test consists in the negation of this specification, i.e, calling APDU command associated to cardholder authentication without doing the previous steps. If the test is accepted by the EMV card, this means that vulnerability has been detected, otherwise the behavior is normal.

VTG process [17] consists of two main parts as shown in Figure 2. First, the model derivation generates derivative faulty models from a functional model. This derivation is fully automated. Secondly, from these new models, abstract tests are extracted.

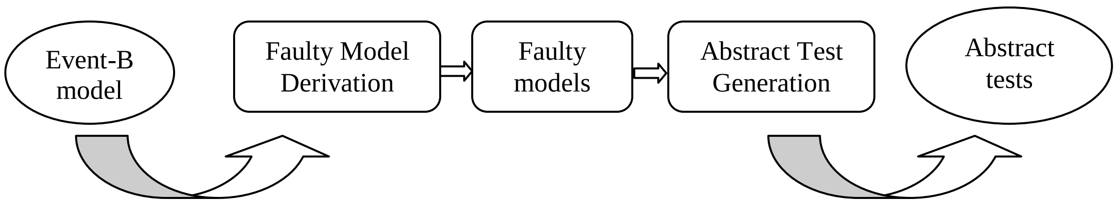


**Figure 2. VTG Processes**

The principle of the VTG tool can be explained through the following part of Event-B code:

```
event SELECT_APPLICATION
    where
      @Statemachine1_isin_Initialisation_t Statemachine1=
      Initialisation
    then
      @Statemachine1_enterState_Application_Selected Statemachine1 ≔
      Application_Selected
End
```

The associated generated Faulty model is expressed by taking the negation of test guards like what is shown below:

```
event SELECT_APPLICATION
    where
      @Statemachine1_isin_Initialisation_t ┐(Statemachine1=
      Initialisation)
    then
      @Statemachine1_enterState_Application_Selected Statemachine1 ≔
      Application_Selected
  End
```

The principle is generalized for all tests guards of the generated Event-B model to generate Faulty models, then, VTG tool generated abstract tests related to the test condition.

Finally, the generated abstract tests are presented in the form of a set of XML files; each one is associated to an APDU command test. As a result, the number of generated XML files containing the abstract tests is equal to the number of APDU commands. This number is also equal to the number of events described in the Event-B models.

As a simple example of vulnerability test, we took the following sequence to test VEFIRY APDU command:

**SELECT_APPLICATION → VERIFY**

The general structure of generated abstract tests takes the form of XML file. The root is <extended_test_suite> tag containing the test cases. Each test case contains all the steps to produce the vulnerability test.

Below, the generated abstract tests for the VERIFY command using the sequence. The first step namely SELECT_APPLICATION represents a correct APDU command according to EMV Specifications. The security test is presented by the last step <VERIFY>. This is a vulnerability test because the cardholder verification step is performed without doing the card authentication which contradicts the EMV specifications. If the test is accepted by the card, this means that the card is vulnerable at this level. Otherwise, it means that the test is rejected.

```
<test_case>
  <step name="SELECT_APPLICATION">
    <modified name="Statemachine1">Application_Selected</modified>
  </step>
  <step name="VERIFY">
    <modified name="Statemachine1">Transaction_Aborted</modified>
  </step>
 </test_case>
```

## 6.5. Our Automatic Vulnerability Tests Generator

In this step, we based on the EMV applet already implemented by OpenEMV project [22]. To do this, we took a smartcard JCOP 21. We loaded the chip with the Java Card Applet named “OpenEMV”. This applet has played the role of EMV application in accordance with EMV specifications [6] describing an EMV transaction. On the other hand, we simulated the terminal by a set of Java classes, that we implemented, through which we communicate with the card. The Figure 3 presents a simplified diagram of our architecture.

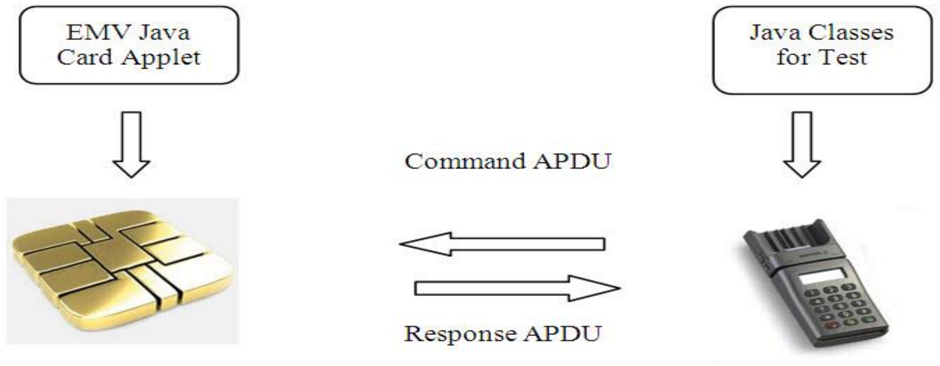


**Figure 3. Diagram showing the Proposed Test Architecture**

To generate vulnerability tests, it was necessary to create our tests generator which takes as input the abstract tests already generated in the section 6.4. Our tests generator explores the XML files relating to abstract tests and converts them into sequences of concrete APDU commands to send to the card. To do this, we based on the structure of APDU commands. For example, the SELECT_APPLICATION APDU command is defined as a method which takes as parameters showed in table1: CLA=00, INS=A4, P1=04, P2=00 and data representing the Applet identifier.

Once the request of each test is structured as series of APDU commands understandable by the card, the request is sent to the card for processing. Our generator retrieves the response returned by the card and save it in a XML file presenting physical tests of vulnerabilities. The whole process of security tests generation is performed automatically once and for all.

To implement our automatic tests generator, we used OPAL and JDOM libraries: OPAL library to communicate with the EMV card and JDOM to manipulate XML files including abstract test cases files.

Below, a small part of concrete test case generated for VERIFY command which describes the structure of vulnerabilities tests:

```
<concrete_test_case>
    <step name="evt_VERIFY2_8_5_EUT">
      <request>
        <cla>00</cla>
        <ins>20</ins>
        <p1>00</p1>
        <p2>80</p2>
        <data>0201010101010101</data>
      </request>
      <response>
        <sw1>69</sw1>
        <sw2>83</sw2>
        <data />
      </response>
    </step>
  </concrete_test_case>
```

If we start by checking the cardholder’s PIN code without executing one of the previous commands APDU namely SELECT APPLICATION, GET PROCESSING OPTIONS, READ RECORD and INTERNAL AUTHENTICATE, this test, in this case, present vulnerability test which is generated by our vulnerability tests generator. We had as generated test case the XML code given in the previous example.

The returned status word SW1SW2 equals to “6983” means that VERIFY command APDU is not allowed and the authentication method is blocked which means that the false test is rejected. According to EMV specification, this is normal behavior of the card, *i.e.*, the cardholder verification operation should be stopped if the necessary previous steps are skipped.

### 6.5. Synthesis

As synthesis of our approach of generating vulnerability test suites using MBT approach, each APDU command is considered as a transition of our Machine state diagram, thus as an event of our Event-B model, with guards presenting potential conditions. In this work, we are interested more in the execution order of APDU commands in an EMV transaction, so in the terminal behaviour. Any lack of an APDU command execution may present a vulnerability for our EMV card.

We know that the majority of old research works, followed MBT approach, focused only on the generation of conformance tests of the system in order to be sure that all of the functional specifications have been meet. They do not take into account the vulnerabilities identification. There are few researches, as Lancia’s work [18] that focused on the identification of vulnerabilities of system implementations based on fuzzing techniques for smartcards. The fuzzing revealed to Lancia a number of deviations from specifications of tested implementations, which can be explained by the complexity of dedicated smart card protocols. As consequence, the fuzzing test for smart cards is complementary to conventional test approaches. For all these reasons and given the importance of security in the smart card field, we have developed our own generator of both functional and vulnerability tests for EMV protocol. The main aim was to ensure that the system is functional and has no vulnerability.

Regarding modeling, we opted for the choice of SysML language which is based on the minimal subset of UML that satisfies the needs of systems engineers. SysML provides also the possibility to represent requirements and relating them to the model of a system, the actual design and test cases. Comparing to other modeling languages, it contains everything it needs a systems engineer to model an Embedded System like smart card, but with less theoretical concepts. The major advantage is that it allows the modeling of the requirements, the system, and the traceability between them. It is true that we did not use requirement diagram until now, however, we strive very later the use of the requirement diagram with MBT approach and compare it with our actual methodology.

## 7. Conclusion

Today, the smart cards offer a considerable computing power and embed advanced security mechanisms especially EMV cards. Indeed, the EMV standard requires, for example, the use of payment cards containing integrated circuits able to perform specific cryptographic functions related to safety and security of card. However, this potential security provided by the hardware and software platform can be destroyed through design or implementation vulnerabilities. If the vulnerability is detected after the manufacture of the card and its implementation, it will be too late and it requires a high cost to correct it. The use of models obtained in the design phase is cheaper even if it adds another time devoted to the models construction and tests generation based on these models

EMV specifications are presented in a complex way. The proposed state machine diagram is easy to understand for anyone familiar with EMV standard and clearly shows the steps of real EMV transaction between terminal and card. Usually, terminal-card specifications are spread over the four books [6]. So, we proposed simple model that gathers the specifications in term of EMV transaction between Terminal-Card. Then, we generated formal models with well-defined concepts and semantics using Event-B method. The formal models are animated with ProB to ensure their correctness. Next, based on the Event-B model, we generated abstract tests using VTG tool [17]. Since, the abstract tests are available; we created our own automatic vulnerability Tests Generator which takes as input the directory path in which the abstract tests are saved. Then, it builds the sequences of APDU commands associated to each abstract test case to send to the card. Once APDU commands are sent to the card, we get the responses of the chip. Both of APDU commands and associated responses are stored in XML files presenting the vulnerability tests. To implement our tests generator, we used OPAL to communicate with an EMV card by sending APDU commands and returning APDU responses.

As result, our MBT approach allows us to generate not only the functional tests, but also the vulnerability ones. For functional tests, we simply translate the model into a list of test cases for system compliance. While in the vulnerability tests, we had to take the negation of various conditions presented by the model. If all of vulnerability tests are rejected by the card, it means that there is no vulnerability and the system is safe and not vulnerable. This is the strong point of our contribution.

It is true that, in this paper, we could not test all possible guards, otherwise we will have a combinatorial explosion. Indeed, some APDU commands values can take a wide range of values. The exploration of these values is tedious and exhaustive test of all possible combinations of values leads to a combinatorial explosion.

Regarding the difficulties, we could not test the true EMV payment cards because vulnerability test may at any time block it. However, we tested a public OpenEMV application[22] after loaded it onto a smart card, we found some vulnerabilities in the order of ten because the application was not yet complete.

For future work, we aim, on the one hand, continue the tests generation for the various guards of each event of our Event-B model by exploiting fuzzing tests in the same context of EMV transaction. On the other hand, we would complement other EMV specifications not taken into account by our model, we hope to create requirements diagram provided by SysML language. Moreover, this justifies our choice of SysML which presents several ways to model a complex system as EMV card. The tests generation based on the requirements diagram became possible because several studies already exist around the conversion of SysML models to Event-B models. We cite as an example the generating High-Level Event-B System models from KAOS Requirements Models [19].

## References

[1] J. Ruiter and E. Poll, “Formal Analysis of the EMV Protocol Suite”, Theory of Security and Applications, lecture notes in Computer Science, vol. 6993, **(2012)**, pp. 113-129.

[2] M. Leuschel and M. Butler, “ProB: A Model Checker for B. FME 2003: Formal methods”, Lecture Notes in Computer Science, vol. 2805, **(2003)**, pp. 855-874.

[3] T. Systa, K. Koskimies and E. Makinen, “Automated compression of state machines using UML statechart diagram notation”, Information and Software Technology, vol. 44, **(2002)**, pp. 565-578.

[4] K. Markantonakis, M. Tunstall, G. Hancke, I. Askoxylakis and K. Mayes, “Attacking smart card system: Theory and practice”, Information security technical report, vol. 14, **(2009)**, pp. 46-56.

[5] ISO/IEC, “ISO/IEC 7816: Identification cards – Integrated circuit cards, **(2005)**.

[6] EMV. Book 1-2-3-- Application independent ICC to Terminal Interface requirements, 4.3 edition, November **(2011)**.

[7] M. Bond, O. Choudary, S. J. Murdoch, S. Skorobogatov and R. Anderson, “Chip and Skim: cloning EMV cards with the pre-play attack. arXiv: 1209.2531v1 [cs.CY]”, **(2012)**.

[8] S.-J. Murdoch, S. Drimer, R. Anderson and M. Bond, “Chip and PIN is Broken”, IEEE Symposium on Security and Privacy, **(2010)**.

[9] F. Aarts, J. Ruiter and E. Poll, “Formal models of bank cards for free”, SECTEST’ 13, to appear, **(2013)**.

[10] B. Blachet, “An efficient cryptographic protocol verifier based on prolog rules”, Computer Security Foundations Workshop (CSFW), IEEE, **(2001)**, pp. 82-96.

[11] K. Bhargavan, C. Fournet and S. Tse, “Verified interoperable implementations of security protocols”, computer Security Foundations Workshop (CSFW), IEEE, **(2006)**, pp. 139-152.

[12] M. Felderer, B. Agreiter, P. Zech and R. Breu, “A classification for model-based security testing”, Advances in System Testing and Validation Lifecycle (VALID 2011), **(2011)**, pp. 109-114.

[13] Y. Vanderperren and W. Dehaene, “UML 2 and SysML: an Approach to Deal with Complexity in SoC/NoC Design”, Proceedings of the Conference on Design, Automation and Test in Europe (DATE’05), Munich, Germany, IEEE Computer Society, **(2005)**.

[14] C. Snook and M. Butler, “UML-B: A plug-in for the Event-B tool set”, Abstract State Machine, B and Z, First International Conference ABZ, **(2008)**.

[15] J.-R Abrial and T.S. Hoang, “Using Design Patterns in Formal Methods: an Event-B Approach”, Proceedings of the 5th International Colloquium: Theoretical Aspects of Computing (ICTAC 2008), Istanbul, Turkey, Springer-Verlag, **(2008)**.

[16] A. Bkakria, G. Bouffard, J. Iguchy-Cartigny and J.-L. Lanet, “OPAL: an open-source global platform Java library which includes the remote application management over HTTP”, National conference e-smart 2011, Nice (France), **(2011)** September.

[17] A. Savary, M. Frappier and J.-L. Lanet, “Automatic Generation of Vulnerability Tests for the Java Card Byte Code verifier”, Network and Information Systems Security (SAR-SSI) conference, 2011. DOI:10.1109/SAR-SSI.2011.5931379, **(2011)** May.

[18] J. Lancia, “A fuzzing framework for smart cards: EMV protocol application”, SSTIC, **(2011)**.

[19] C. Ponsard and X. Devroey, “Generating High-Level Event-B System Models from KAOS Requirements Models”, INFORSID 2011, Lille (France), **(2011)** May 26-26.

[20] M. Utting and B. Legeard, “Practical Model-Based Testing: A Tools Approach”, Morgan Kaufmann, 1ère edition, **(2007)**.

[21] R. Alves de Almeida Junior, “Model-based Testing with a B Model of the EMV Standard”, Thesis http://www.cs.ru.nl/bachelorscripties/, **(2012)**.

[22] E. Poll, J. Ruiter. OpenEMV: A Java Card implementation of the EMV standard. http://sourceforge.net/projects/openemv/, **(2012)**.