HAL
archives-ouvertes.fr

Analysis of Embedded Applications By Evolutionary Fuzzing

Vincent Alimi, Sylvain Vernois, Christophe Rosenberger

► **To cite this version:**

Vincent Alimi, Sylvain Vernois, Christophe Rosenberger. Analysis of Embedded Applications By Evolutionary Fuzzing. Workshop on Security and High Performance Computing Systems, International Conference on High Performance Computing & Simulation (HPCS), Jul 2014, Rome, Italy. pp.7. hal-01019978

HAL Id: hal-01019978

https://hal.archives-ouvertes.fr/hal-01019978

Submitted on 7 Oct 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Analysis of Embedded Applications By Evolutionary Fuzzing

V. Alimi, S. Vernois, C. Rosenberger

Normandie Univ, France;
UNICAEN, GREYC F-14032 Caen, France;
ENSICAEN, GREYC, F-14032 Caen, France;
CNRS, UMR 6072, F-14032 Caen, France
christophe.rosenberger@ensicaen.fr

***Abstract*—In this paper, we propose to use fuzzing techniques to discover vulnerabilities in programs hosted into smart cards used for telecommunications or banking purposes (SIM cards, credit cards, secure element into NFC mobile devices...). Those programs – called *applets* – usually host sensitive applications and manipulate sensitive data. A flaw by design or by implementation in one of those applet could have disastrous consequences. The proposed approach uses a genetic algorithm to optimize the vulnerabilities search. We illustrate the benefit of the proposed method on a MasterCard M/Chip applet through experimental results.**

I. Introduction

With the development of Internet and digital networks, new transactions have emerged. These electronic transactions govern our daily lives such as access control to a building with a badge, internet payment, payment with a mobile phone, etc. As for example, here are some figures in France in 2010: E-commerce (online shopping) concerned an amount of $31 billion, it has been recorded $7 billions of credit card payments for an amount of $336 billion and 1.5 billion withdrawals on ATMs for $115 billion.

Electronic transactions have opened the door to a multitude of opportunities declined in various forms : the internet portal to check your bank accounts, make transfers or place orders in scholarships, a smart card to open a door or validate his title transit, a downloaded to a computer or a mobile device like a PDA or a mobile phone application. This latter category of mobile equipment is extremely powerful in terms of service offerings. Indeed, a mobile phone (which will be called mobile thereafter) has the following characteristics :

- Nomad: it is characterized by its mobility which allows him to become an indispensable tool of everyday life ;
- Online: with the phone 3th and now 4th generation, the mobile has a roaming internet access broadband. This allows, among other things, the consultation emails, access to web services, ...;
- Powerful: current mobile called *smartphones* ship processors more powerful, of memory increasingly important and operating more efficient system.

By nature and as for example, contactless transactions of mobile services are more prone to attacks such as *man-in-the-middle*, *eavesdropping and relay*. The discovery of new vulnerabilities and new types of attacks are often defined through the generalization of existing ones from the state of the art. When developing new applications, most of implementation errors are mostly detected. It may nevertheless be possible to have undetected errors which could have disastrous consequences.

Vulnerabilities can be classified in different ways. In [1], Dowd et al. propose a classification into three classes:

- Vulnerabilities due to the design: introduced during the transcription of specifications into functional specifications themselves transcribed into technical specifications;
- Vulnerabilities due to the implementation: introduced when developing the program code or server ;
- Vulnerabilities due to the use: observed during the deployment and operation of the program or server.

The implementation errors are mostly detected by error or warning messages from the compiler, unit testing or integration testing. It may nevertheless happen that some errors are not detected, either by lack of resources to allocate to the test phase or because the test phase focuses primarily on compliance with the functional requirements. As for example, the attack on the Mifare Classic cards made by Koning *et al.* published in [2] exploits a weakness in the pseudo-random generator involved in the encryption of the communication between the card and the player. In the context of mobile contactless services, implementation errors can have disastrous consequences. Consider as an example the case of a mobile proximity payment service on the Secure Element as an applet. This application was developed considering defined specifications by an actor such as Visa, MasterCard, American Express or by a so-called proprietary networks. To overcome the difficulty of conducting extensive testing, fuzzing techniques can be very useful. The fuzzing is a technique to discover vulnerabilities in a program or a system. The principle is to inject malformed or the limits of their values.

Barton Miller, professor at the University of Wisconsin, is the originator of the field of fuzzing. He introduced the concept of *fuzz program* in 1988 in a class project [3] whose first findings were published in [4]. The class project consisted in sending some random character chains as input into some UNIX processes in order to make them *crash*. In [5], Clarke enunciates the common features to fuzzing programs:

- data generation (creating data to be passed to the target);
- data transmission (getting the data to the target);
- target monitoring and logging (observing and recording the reaction of the target), and;
- automation (reducing, as much as possible, the amount of direct user-interaction required to carry out the testing regime).

The last two may be considered optional or can be implemented in an external module fuzzer. Three methods for the generation of data [6] can be distinguished: Random, data mutation and protocol analysis. The random generation involves a generator that produces a set of test data. This approach minimizes the effort and time required, but turns out to be less effective for detecting vulnerabilities because they are not necessarily known in advance. The mutation data combines two techniques: data capture and selective mutation. Starting from a valid input data is carried mutations in order to obtain a set of test data which is very close to the structure of a valid data. Generating data based on protocol analysis is based on the principle that the input data meet very often a protocol or a pre-defined format. A fuzzer of this type is implemented by defining a model of this protocol so that it can create valid input but whose data is random data.

The paper is organized as follows. in section II, a brief state of the art is given related to existing analysis method for embedded applications such as the ones in a Secure Element. Section III describes the proposed method based on a Fuzzing approach exploiting a genetic algorithm to optimize the space search. Illustrations of the proposed method are given in section IV on a real JavaCard payment application. Section V gives the conclusion and perspectives of this study.

II. State of the art

In [7], Guyot illustrates the use of fuzzing on applets. He demonstrates how easy it is to accurately determine the commands (i.e. the instruction codes) that are accepted by the application. The commands – also called Application Data Units (APDU) – are coded in accordance with the ISO 7816 standard [8]. A command that is recognized induces an action and the return of data or of status words indicating an internal error. For instance, the applet can return the status words `6F 00` if the input data are not conform and provoked a treatment error in the applet. On the other hand, it the command is not recognized the standard status words `6D 00` are returned. Then, Guyot uses the results obtained to make the application (a student card application) *fuzzing-proof*. Instead of using a different instruction code per supported command, he uses a single instruction code and differentiates the commands by using the reference parameters (`P1` and `P2`). The result is an increase of the complexity for an attacker to find out the way the application works because the use of the reference parameters is not standard.

In [9], Barreaud *et al.* expose a method to analyse the vulnerabilities on a smard card embedding a web server. Their approach consists in *fuzzing* the BIP protocol (Bearer Independant Protocol) responsible for the communication between the application processor and the SIM card of a mobile phone. They use the fuzzing framework *Peach* that they extend with the Pyscard library in order to communicate with the SIM card. They modelize the BIP protocol with an XML file. They also add to the file the two markups $< Expected >$ and $< Response >$ in order to monitor the target. They find out that some of the tested cards do not properly implement the protocol as defined by ETSI (European Telecommunications Standards Institute) and that some have implementation flaws allowing to create a Denial of Service attack.

In [10], Lancia publishes one of the only known fuzzing methodologies aiming at discovering vulnerabilities in EMV (Eurocard Mastercard Visa) banking applications. To realize this attack, he uses the block fuzzing framework *Sulley* [11]. Sulley is written in Python and is recognized as one the most complete and efficient fuzzing framework. In order to transmit data, Lancia has integrated the *Triton* library, also written in Python, allowing to communicate with smart cards through the standard PC/SC API. For monitoring the target, Lancia developed a reference implementation of the EMV applications he targeted. The same commands are simultaneously sent to the real card and the reference implementation. An anomaly is detected when the results returned by the real card differs from the one returned by the reference implementation. All the protocol commands are modelized then linked together in order to create the protocol graph (cf. figure 1).

To test a particular command in the graph, all the preceding commands are sent with their default value. For that particular command, the framework Sulley generates the data from the model of the protocol. Thanks to this methodology, Lancia has brought out some functional differences with the EMV specifications and some security flaws on implementations of the Visa and Mastercard specifications. For instance, he noticed that a particular combination of data could generate the reset of the offline counters.

Lancia's approach has proven efficient by showing some functional differences and security flaws on real cards that have been certified in accredited certification labs. This efficiency is the result of a precise description of the data model and a thoroughness in the development of the reference implementation. However, the approach has gaps that we propose to fill in.

In this paper, we propose an improvement of Lancia's approach for the testing of payment applications in black box

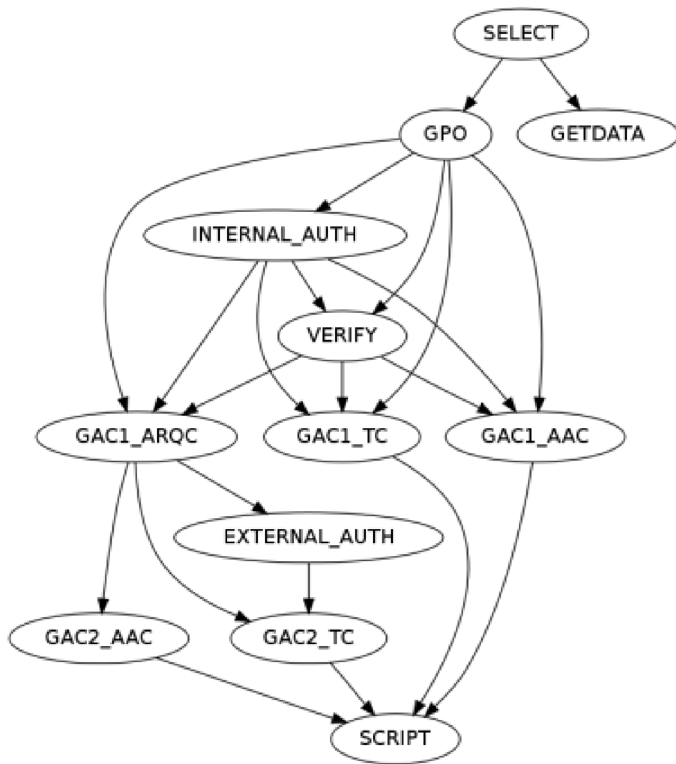

Fig. 1. EMV protocol model according to Lancia in [10]

by using evolutionary algorithms.

III. Proposed method

A. Principles

Instinctively, we thought that an ideal testing framework would be a framework capable of assessing the quality of the data sent to the card function of the effect produced. In other words, the ideal framework would be capable of modifying in real-time the commands sent to the card function of the data returned in order to discover vulnerabilities.

A fuzzer based on a reference implementation does not match our initial need to adapt the data generated function of the results it produces. Indeed, this kind of fuzzer makes a static comparison and raises a warning when the results returned by the target and the reference implementation differ. Instead, we prefer using an algorithm capable of iterating over a new round of data based on the evaluation of the preceding rounds data, such as evolutionary algorithms.

In this paper, we propose a fuzzing methodology for smart card applications using the genetic algorithm (GAs) to generate the data sent to the card and evaluate the quality of this data.

B. WSCT Framework

WinSCard Tools, alias WSCT, is a framework foremost developed for Windows using C# programming language. In the same way as Java is based on the execution of byte code independent of the machine thanks to the Java Virtual Machine (JVM), C# compiler generates a byte code intended to be executed by a Common Language Runtime (CLR). Hopefully CLR implementations exist on most common systems: .Net framework by Microsoft is dedicated to Windows operating systems starting with Windows XP and Mono platform by Xamarin is dedicated to Linux based operating systems up to iOS and Android, even if Windows is also supported.

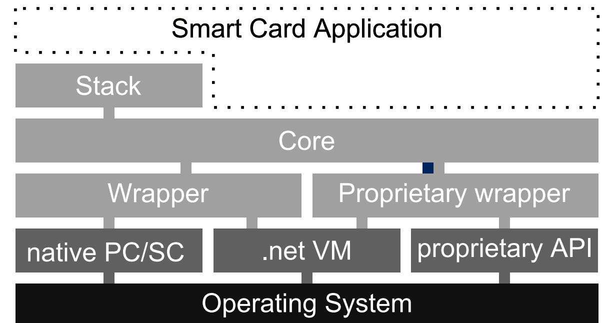

Fig. 2. WSCT framework overview

WSCT framework itself is mainly composed of five modules (figure 2) :

- Wrapper: this module publishes a first basic API allowing to access to the concrete PC/SC resource manager hosted on the machine. It is the most machine dependent module as it has to be adapted for each operating system. Standard implementation allows transparent access to PC/SC smartcard readers on Windows, Linux and MacOS by providing one binary for all. By overriding this module, the entire framework can self adapt to other systems (Android NFC reader is a work in progress for example) or specific readers and probes.
- Core: this module aims at providing an higher level API allowing the communication with the wrapped readers and inserted smartcard. It provides useful interfaces and objects allowing observability of communication between the card and the caller. It's the foundation of genericness and re-usability of tools developed upon WSCT.
- Stack: it publishes a mechanism allowing the chaining of layers able to intercept and transform data exchange between the caller and the card.
- ISO 7816 library: it provides mainly a partial implementation of common ISO7816-4 normalized objects, such as C-APDU and R-APDU (normalized formats of command and response) or SELECT instruction.
- Helpers: it publishes a set of useful objects often needed when working with smart cards. For example TLV format (tag length value), often used with smart cards, is defined there and can be used everywhere.

These items have finally been made public [12]. A graphic user interface is also available to help creating demonstrators. Several libraries have been built on top of WSCT to ease the work on concrete card. The most used is the EMV library

that implements main parts of EMV specification and allows sending, observability and interpretation of exchanges relative to EMV payment. The sources of these libraries are not published to prevent public misuse.

The added value of this framework compared to other existing API dedicated to smartcard communication, whatever the language, is certainly the passive observation of the communication that is natively provided, allowing the interpretation of transaction exchanges to be kept separated and independent from the functional cinematic, as illustrated by figure 3. This is why the fuzzing experimentations were realized using this framework.

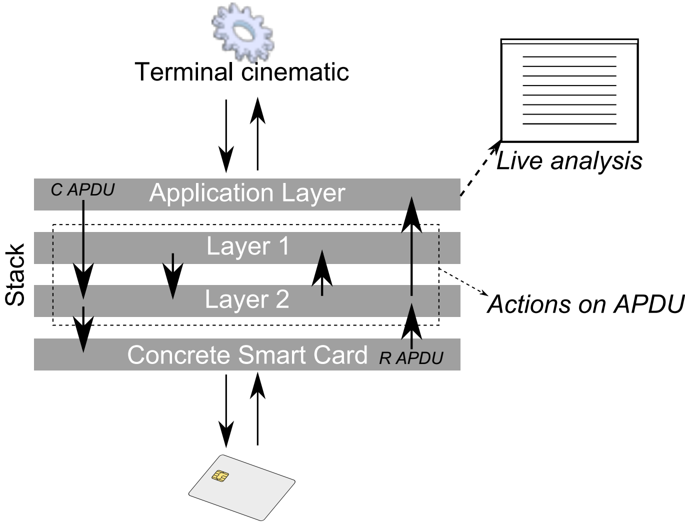

Fig. 3. WSCT observability (Core) and interception (Stack)

C. Genetic algorithm

Genetic algorithms determine the optimal value of a criterion by simulating the evolution of a population until survival of best fitted individuals [13], [14]. The survivors are individuals obtained by crossing-over, mutation and selection of individuals from the previous generation. We think that GA is a good candidate to find out the optimal combination of segmentation results for two main reasons. The first one is due to the fact an evaluation criterion is not very easy to differentiate. GA is an optimization method that does not necessitate to differentiate the fitness function but only to evaluate it. Second, if the population is enough important considering the size of the search space, we have good guarantees that we will reach the optimal value of the fitness.

A genetic algorithm is defined by considering five essential data:

1) *genotype*: It is composed of the candidate solution for the command resulting to a vulnerability. In our case, we focus on the GENERATE AC command on a payment applet (see Figure 4)

2) *initial population*: a set of individuals characterized by their genotypes,

3) *fitness function*: this function enables us to quantify the fitness of an individual to the environment by considering its genotype. In our case, it corresponds to the evaluation of the GENERATE AC command. For more details, see section IV-B.

4) *operators on genotypes*: they define alterations on genotypes in order to make the population evolve during generations. Three types of operators are used:
   - individual mutation: individual's genes are modified in order to be better adapted to the environment. We use the non-uniform mutation process which randomly selects one chromosome x_i, and sets it as equal to a non-uniform random number:
   $$x_i' = \begin{cases} x_i + (b_i - x_i)f(G) & if\ r_1 < 0.5 \\ x_i - (x_i + a_i)f(G) & if\ r_1 \geq 0.5 \end{cases} \quad (1)$$
   where $f(G) = (r_2(1 - \frac{G}{G_{max}}))^b$

     The values r_1, r_2 are numbers in the interval [0,1]. The values a_i and b_i are the lower and upper bound of chromosome x_i. G is the current generation, G_{max} is the maximum number of generations and b is a shape parameter
   - selection of an individual: individuals that are not adapted to the environment do not survive to the next generation. We used the normalized geometric ranking selection method which defines a probability P_i for each individual i to be selected as following:
   $$P_i = \frac{q(1-q)^{r-1}}{1-(1-q)^n} \quad (2)$$
   where q is the probability of selecting the best individual, r is the rank of individual (1 is the best) and n is the size of the population.
   - crossing-over: two individuals can reproduce by combining their genes. We use the arithmetic crossover which produces two complementary linear combinations of the parents:
   $$\begin{aligned} X' &= aX + (1-a)Y \\ Y' &= (1-a)X + aY \end{aligned} \quad (3)$$
   where X and Y are the genotype of parents, a is a number in the interval [0,1] and X' and Y' are the genotype of the linear combinations of the parents.

5) *stopping criterion* : this criterion allows to stop the evolution of the population. We can consider the stability of the standard deviation of the evaluation criterion of the population or set a maximal number

of iterations (we used the second one with the number of iterations equals to 2000).

Given these five informations, the execution of the genetic algorithm is carried out in four steps:

1) definition of the initial population (segmentation results) and computation of the fitness function (evaluation criterion) of each individual,
2) mutation and crossing-over of individuals,
3) selection of individuals,
4) evaluation of individuals in the population,
5) back to step 2 if the stopping criterion is not satisfied.

IV. ILLUSTRATIONS

A. Experimental protocol

Our GA-based testing framework is based on the architecture described in section III-B. We're adding to the logic responsible for the fuzzing the library AForge.NET [15] offering an open source implementation of genetic algorithms. We're loading, installing and personalizing an application that we developed onto a smart card. We implemented in this applet a part of the MasterCard M/Chip [16] specifications whose transaction flow complies with the EMV standard. We simplified the development by keeping only the piece of code strict necessary to perform a transaction. Table I summarizes the parts of the experiment.

Fuzzing framework	WinSCard Tools
Framework language	C#
Smart card used	JCOP 2.4.1 simulator
Payment application	MasterCard MChip 4
GA library	AForge.NET
Population size	10000
Number of iterations	5000
Selection of best individuals	Elitist selection
Selection method	Permutation of two genes randomly selected
Mutation method	Crossover of two genes randomly picked)
Data represented by individuals	Data of the command GENERATE AC (cf. Figure 4)
Fitness function	Evaluation of the response to GENERATE AC (CID, CVR ...) (cf. **??**)
Coefficient α, score multiplier	$\alpha = 1$ if the cryptogram is an AAC $\alpha = 3$ if the cryptogram is an ARQC $\alpha = 5$ if the cryptogram is an TC

TABLE I. SUMMARY OF THE EXPERIMENT

The following section details the implementation of our approach on this particular applet.

B. Experiment procedure

In our GA-based approach, we follow the flow of a payment transaction and use GAs to generate the data field of the command texttGENERATE AC. Hence, the genome of the individuals represents the data field of this command, *i.e.* the data related to `CDOL 1` or `CDOL 2` depending on the case. The CDOL 1 related data of the MasterCard M/Chip application is given in figure 4. The fitness evaluation of the command `GENERATE AC` is a combination of the type of cryptogram returned by the application indicated by the data element *Cryptogram Information Data* (CID) and the checks performed by the application gathered in the data element *Card Verification Results* (CVR). Both CID and CVR are returned by the card in the response to the command `GENERATE AC`.

Data Element	Tag	Length
Amount, Authorized (Numeric)	'9F02'	6
Amount, Other (Numeric)	'9F03'	6
Terminal Country Code	'9F1A'	2
Terminal Verification Results	'95'	5
Transaction Currency Code	'5F2A'	2
Transaction Date	'9A'	3
Transaction Type	'9C'	1
Unpredictable Number	'9F37'	4
Terminal Type	'9F35'	1
Data Authentication Code	'9F45'	2
ICC Dynamic Number	'9F4C'	8
CVM Results	'9F34'	3

Fig. 4. MasterCard M/Chip CDOL 1 related data

MasterCard M/Chip CVR comprises six bytes: the first three bytes are used for information only while the last three bytes are used for the decision making. We detail in figure 5 the bits used by the fitness function. The fitness score is incremented by one if one of the bits flagged in bytes 1 to 3 is set to 0, and also incremented by one if one the bits flagged in bytes 4 to 6 are set to 1. Then this score multiplied by a coefficient α whose value is function of CID value, *i.e.* of the cryptogram returned: 1 if the cryptogram is an AAC, 3 if the cryptogram is an ARQC and 5 if the cryptogram is a TC.

On the fuzzing framework side, we enriched the library AForge.NET with a new object type `Chromosome` capable of generating the data field of the command `GENERATE AC` based on the `CDOL 1` or `CDOL 2` and to performs crossovers and mutations. We also developed an evaluation function and a fitness function dedicated to this type of chromosome. For this fuzzing session, we had set a simple goal to reach in order to validate our approach: fuzz the command `GENERATE AC` responsible for the approval of the transaction by the application. The experiment procedure is the following:

1) Definition of the coefficients used for the score evaluation: $\alpha = 1$, $\beta = 3$, $\gamma = 5$.
2) Creation of a population comprising n individuals.
3) For each iteration i, perform a payment transaction with the n individuals :
   a) send the commands `SELECT`, `GET PROCESSING OPTIONS`, `READ RECORD` and `GET DATA` with their default values,
   b) randomly, send the PIN code with the command `VERIFY` (requires the prior knowledge of the PIN code),
   c) the first command `GENERATE AC` is sent to request a `TC` with the data of the individual coding the `CDOL 1` related data. If an `ARQC` is returned by the application, the second `GENERATE AC` is sent requesting a `TC` with

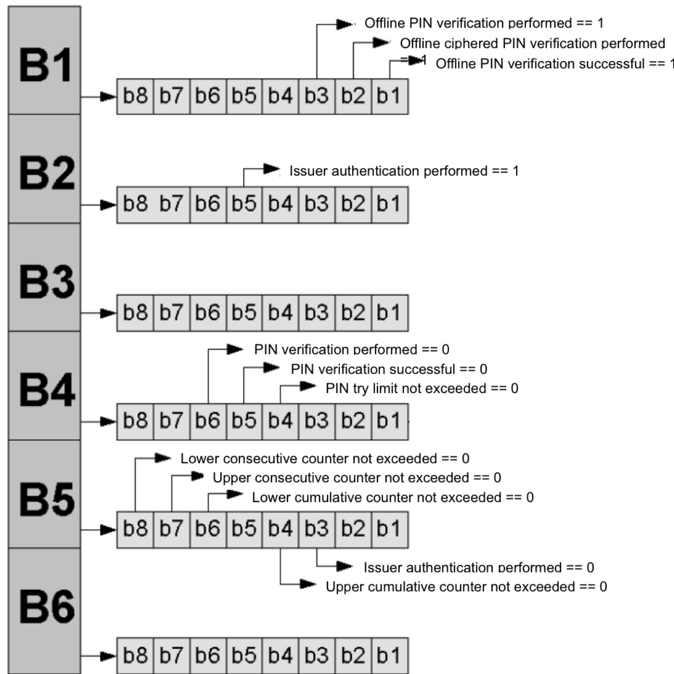

Fig. 5. Data associated to the `Card Verification Results` from the MasterCard M/Chip application used for the fitness function of the genetic algorithm

the data of the individual coding the `CDOL 2` related data,

d) based on the data returned by the application, each indivual is evaluated by the fitness function as explained above,

e) the best individual is selected to create the next generation of individuals.

All the transactions are recorded and when a reset of the offline counters occurs or when a transaction is approved above the limit of offline consecutive transactions, this transactions is also added to the abnormal transactions records. The reset of the offline transactions counters can be easily detected as the application returns them in the response to the command `GENERATE AC` into the data element *Issuer Application Data*.

C. Results

The first conclusion we came to was that the smart cards we used to perform our fuzzing sessions appeared inadequate for this kind of stressful treatment. Indeed, we made our set of smart cards unusable after 25,000 consecutive transactions. We then decided not to perform our fuzzing sessions on real cards but to load the applications into a simulator (*cf.* figure 6). After obtaining a license of the NXP tool *JCOP*, we integrated it to WinSCard Tool. To do so, we used the JCOP simulator executable file and ported the JCOP offcard library written in Java to C#. Then, we developed an interface in order to seamlessly send and receive APDU commands to the JCOP simulator from the our testing framework. While this dematerialization introduces a bias into the experiment as some vulnerabilities could have a hardware origin, this allowed us to dramatically increase the framework execution speed. For instance, we achieved to perform 5,000 EMV payment transactions per minute on a powerful computer.

After many fuzzing sessions – dozens of hours and millions of transactions – the proposed approach did not allow us to observe any rest of the offline counters on our Mastercard M/Chip implementation, but we observed the approval of many transactions above the limit of consecutive offline transactions. This proves the presence of an anomaly in our implementation of the payment application which was the goal of our fuzzing framework and of our experiment. Those illegitimate transactions are recorded so that we have all the necessary elements for further analysis. However, it is quite difficult to go back to the sequence of commands that lead to this anomaly. For instance, we have detected during a fuzzing session that some transactions got approved offline beyond the 10,000th transaction Hence, it is very difficult to know exactly the context of the anomaly and how the preceding transactions had influence on the result.

V. Conclusion and perspectives

We proposed in this paper a new fuzzing technique to detect vulnerabilities or problem of conformance to specifications based on a genetic algorithm. This technique allows us to optimize the search of commands that result to a problem. Experimental results on a real applet showed some interesting results such as the observation of different illegitimate transactions. We observe that it is difficult to identify exactly the commands resulting to a problem but vulnerabilities are correctly identified.

Perspectives of this study concern the definition of properties on a transaction in order to better understand the situations resulting to a vulnerability or a problem of conformance to specifications.

References

[1] Mark Dowd, John McDonald, and Justin Schuh. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. Addison-Wesley Professional, 2006.

[2] Gerhard Koning Gans, Jaap-Henk Hoepman, and Flavio D. Garcia. A practical attack on the mifare classic. In *Proceedings of the 8th IFIP WG 8.8/11.2 international conference on Smart Card Research and Advanced Applications*, CARDIS '08, pages 267–282, Berlin, Heidelberg, 2008. Springer-Verlag.

[3] Barton Miller. Cs 736, fall 1988, project list. http://pages.cs.wisc.edu/ bart/fuzz/CS736-Projects-f1988.pdf, 1988.

[4] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, December 1990.

[5] Toby Clarke. Fuzzing for software vulnerability discovery. Technical report, Royal Holloway University of London, 2009.

[6] Jason Crampton Toby Clarke. Fuzzing – or how to help computers cope with the unexpected. Technical report, Royal Holloway University of London.

[7] Vincent Guyot. Smart card, the invisible wallet. In *Proceedings of the 9th European Conference on Information Warfare and Security*, 2010.

[8] ISO/IEC, http://www.iso.org. *ISO/IEC 7816-1 to 15: Identification cards – Integrated circuit(s) cards with contacts(Parts 1 to 15).*

[9] Nassima Kamel Matthieu Barreaud, Guillaume Bouffard and Jean-Louis Lanet. Fuzzing on the http protocol implementation in mobile embedded web server. In *Proceeding of C&ESAR 2011*, 2011.

```
Status: No Error
 => 80 CA 9F 83 00                                   .....
 (627454 nsec)
 <= 9F 83 06 95 00 03 04 04 00 90 00                 ...........
Status: No Error
Individual score: 40
Transaction #38 Overall duration: 0d:0h:0m:2s.968
 => 00 A4 04 00 07 A0 00 00 00 04 10 10 00           .............
 (2894 usec)
 <= 6F 17 84 07 A0 00 00 00 04 10 10 A5 0C 50 0A 4D  o...........P.M
    61 73 74 65 72 43 61 72 64 90 00                 asterCard..
Status: No Error
 => 80 A8 00 00 02 83 00 00                          ........
 (3141 usec)
 <= 77 16 82 02 79 00 94 10 10 02 02 01 18 01 01 00  w...y...........
    20 01 01 00 28 01 02 00 90 00                     ...(.....
Status: No Error
 => 00 20 00 80 08 24 12 34 FF FF FF FF FF           . ...$.4.....
 (1863 usec)
 <= 90 00                                            ..
Status: No Error
 => 80 AE 80 00 2B 57 04 23 85 78 36 86 85 83 97 30  ....+W.#.x6....0
    80 32 17 E6 ED AD AD 29 53 41 43 09 17 8D 5A 9F  .2.....)SAC...Z.
    3C 12 EA DE C6 04 0D F1 FC 80 0E EA B3 AB DE 90  <...............
    00                                               .
 (4458 usec)
 <= 77 81 29 9F 27 01 80 9F 36 02 00 27 9F 10 12 01  w.).'...6..'....
    10 A5 00 03 04 04 00 DE C6 00 00 00 00 00 00 0C  ................
    FF 9F 26 08 59 54 84 ED C8 04 8E 35 90 00        ..&.YT.....5..
Status: No Error
 => 80 AE 40 00 1D FC 22 5B 29 51 A1 22 12 4F 1D 7C  ..@..."[)Q.".O.|
    83 0A D8 08 80 8C 28 68 43 C0 4A B0 80 79 16 EA  ......(hC.J..y..
    6C 9F 00                                         l..
 (5444 usec)
 <= 77 81 29 9F 27 01 00 9F 36 02 00 27 9F 10 12 01  w.).'...6..'....
    10 25 10 03 04 04 00 DE C6 00 00 00 00 00 00 0C  .%..............
    FF 9F 26 08 48 C8 E7 00 B2 F8 01 F0 90 00        ..&.H.........
Status: No Error
 => 80 CA 9F 81 00                                   .....
 (509842 nsec)
 <= 9F 81 01 0C 90 00                                ......
Status: No Error
 => 80 CA 9F 82 00                                   .....
 (537499 nsec)
 <= 9F 82 06 00 00 00 00 00 00 90 00                 ...........
Status: No Error
Individual score: 8
Transaction #39 Overall duration: 0d:0h:0m:3s.0
 => 00 A4 04 00 07 A0 00 00 00 04 10 10 00           .............
 (3103 usec)
```

Fig. 6. Use of the JCOP simulator to perform fuzzing on smart card applications

[10] Julien Lancia. Un framework de fuzzing pour cartes puce: application aux protocoles emv. In *Symposium sur la scurit des technologies de l'information et des communications*, 2011.

[11] Pedram Amini. Sulley fuzzing framework. http://code.google.com/p/sulley/.

[12] Sylvain Vernois. Wsct framework.

[13] Matthew Bartschi Wall. *A Genetic Algorithm for Resource-Constrained Scheduling*. PhD thesis, Department of Mechanical Engineering – Massachusetts Institute of Technology, 1996.

[14] C. Houck, J. Joines, and M. G. Kay. The genetic algorithm optimization toolbox (gaot) for matlab 5. Technical report, North Carolina State University, 2005.

[15] Aforge.net :: Computer vision, artificial intelligence, robotics. http://www.aforgenet.com/.

[16] MasterCard. *M/Chip 4 Version 1.1 - Card Application Specifications for Debit and Credit*. MasterCard, October 2006.