Janusz CHORKO, Tomasz KOBYLARZ, Piotr NAZIMEK

Warsaw University of Technology

Testing fault susceptibility of Java Cards

Abstract. *The paper presents a tool for fault injection in Java Card environment based on cref simulator. Using it we examine the problems related to faults sensitivity and the impact of these on security functions and proper Java Card applet execution. During experiments we have checked applets reliability and possible information leaks or security faults in the reference to the injected faults. In this article we describe our experiments and their results.*

Streszczenie. *W artykule przedstawiono narz¸edzie do wstrzykiwania bł¸edów w ´srodowisku Java Card przygotowane na podstawie symulatora cref. Z jego pomoc ¸a sprawdzona została wra˙zliẃosc apletów na zakłócenia tj. ich wpływ na zabezpieczenia oraz poprawne działanie aplikacji dla Java Card. Wykonane eksperymenty miały na celu zbadanie niezáwodnosci działania apletów oraz mo˙zliwych wycieków informacji cźy naruszeń bezpieczenstwa na skutek wprowadzonych bł¸edów. W artykule zaprezentowano przeprowadzone eksperymenty i omówiono ich rezultaty.* ***(Testowańie wra˙zliwosci na bł¸edy kart Java Card)***

Keywords: smart cards, security, dependability, fault injection
Słowa kluczowe: karty inteligentne, bezpieczeństwo, niezawodność, wstrzykiwanie błędów

doi:10.12915/pe.2014.02.21

Introduction

Nowadays smart cards are very important elements of safe and convenient access to commercial and public information services. They are used with success for many years by many systems, such as Global System for Mobile Communications (GSM), where each telephone number is assigned to the Subscriber Identity Module card (usually called a SIM card), acting the key role for secure access to the mobile network. Smart cards are also used in systems providing secure transactions, cashless payments and cryptographic services [1].

One of the key stages of implementing payment cards in banking systems was the EMV standard published in 1995 by Europay, Visa and MasterCard organizations [2]. The main goal of it was to increase the security of transactions and reduce the scale of frauds which was the result of easy stealing data from the magnetic cards that doesn't have strong security mechanisms. Smart cards are used also in other fields where secure storage of information is needed, i.e., electronic signature where smart card stores and manages user private key, electronic tickets used in transportation systems or health cards with personal information related to owner health.

Information security is now one of the most important requirements [7, 12]. For this reason we wanted to check how data security is ensured for Java Card exposed to the different kinds of faults. Using Java Cards cref simulator we have prepared fault injection environment. Based on this we have examined the behavior of typical card applets in case of faults.

Smart card architecture

General hardware architecture of the smart card is shown in Fig. 1. It is usually a plastic card with embedded microprocessor with ROM, RAM and EEPROM memory. The device is controlled by the CPU with dedicated operating system. To accelerate cryptographic operations and provide better security often cryptoprocessor is used. From software point of view Java Card is a smart card that allows running Java programs also called smart card applets. Java Card includes an integrated circuit with native OS for which Java Card Virtual Machine (JCVM) is created. On the top of JCVM there are Java Card APIs [5]. Java Card Virtual Machine implements a subset of Java Virtual Machine capabilities. For this reason there is also Java Card Assembly (JCA) language created.

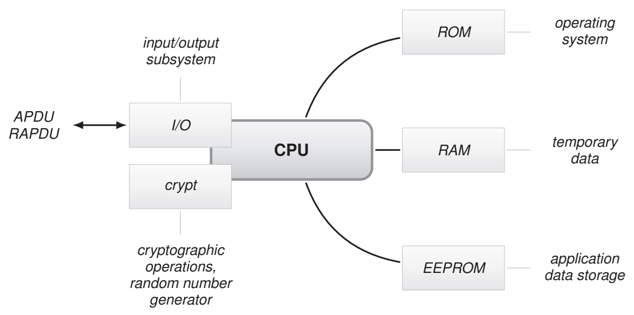

Fig. 1. Smart card architecture overview

Smart card applets behaves differently than desktop Java applications. When an instance of card application (applet) is created it works until it is intentionally removed. Power interruption will freeze the application execution. For this reason, the data stored by the applet is contained in EEPROM memory. RAM memory is used only for temporary computations and its content is lost with the power off.

Communication between smart card and external system is based on a challenge-response protocol. Smart card always waits for an Application Protocol Data Unit (APDU) message from off-card application. APDU structure (Fig. 2) consist of instruction class byte (CLA), instruction code (INS), instruction parameters (P1 and P2) and, optionally, data field with a given length (Lc) as well as the expected data length in response message (Le). Smart card executes the requested action and replies with a Response Application Protocol Data Unit (RAPDU). RAPDU structure contains data from card application and status word bytes (SW1 and SW2) which describe status of the operation. The format of the APDU, RAPDU and general SW values are defined in ISO/IEC specification 7816-4 [3].

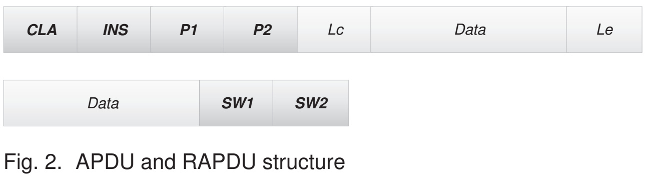

Fig. 2. APDU and RAPDU structure

During an implementation of Java Card applet a programmer can specify actions for given APDUs and required flow of a challenge-response protocol [4]. After that the applet should be compiled into CAP file and transformed into APDU script which is loaded into the card. APDU and RAPDU. Using APDU and RAPDU structure is the only way to exchange insensitive as well as sensitive (cryptographic keys, PIN code) data with the applet.

In the process of applet development most of complex calculations and memory consuming actions were raised beyond the card (called off-card) to the PC environment, where the problem of limited resources usually doesn't occur. These actions includes: applet verification [6], compilation, optimization and consolidation or conversion to CAP files. On-card verification and inspection are usually not performed due to performance limitations.

Fault injection environment

Fault injection is a technique to improve software testing and software reliability study. It allows to examine the behavior of the software in case of various disturbances in its hardware environment. It is hard and expensive to simulate specified faults and their effects in real hardware [8, 9]. For this reason we have used C reference implementation of Java Card run-time environment called cref tool. Cref is a simulator closest to real implementation of Java Card and it supports loading and dumping EEPROM image of Java Card. The cref tool is only available in Sun's Java Card Development Kit as an executable file (source codes are not included).

We have extended cref tool to implement software based fault injector. Its structure is shown in Fig. 3. Into the cref simulator code we have inserted references to callback functions which allow us to impact on applet execution on the fly. Our cref plugin (set of callback functions) traces applet execution loaded from EEPROM image in cref tool and it is able to modify current instruction, its parameters and memory contents according to the selected method like random bit inversion in memory data or instruction opcode as well as in its parameter given by cref driver [10]. Communication between cref driver and cref plugin is based on TCP/IP protocol to avoid the impact of critical errors in cref tool on our driver. We have also created framework for automatic execution of many tests with different types of test scenarios and injected faults (test configuration) with the possibility of tracing their effects (test results).

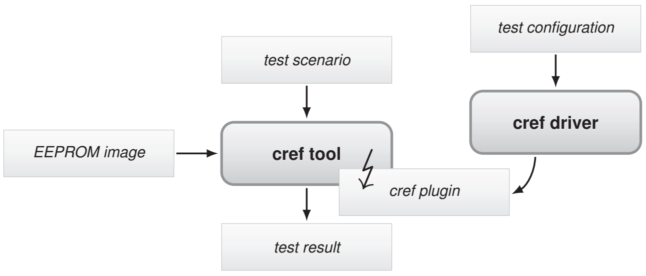

Fig. 3. Fault injection environment

The fault environment allows us to inject various types of faults. The most important are:

- replacement of specific instructions by another one — it can simulate errors in the EEPROM area, in the interpreter module, as well as errors inside implementation of JCVM,
- random modification of specific bits in the instruction opcode -— it simulates errors during EEPROM read or in instructions interpreter module; we have implemented some specific types of disturbances:
  - SET -— set to 1 a specific bit of instruction opcode,
  - RESET -— set to 0 a specific bit of instruction opcode,
  - TOGGLE – change a specific bit of instruction opcode (bit flip),
  - XOR – performs operations on bits of instruction opcode with a given mask,
- set a fixed value of the instruction parameter,
- disturbances like SET, RESET, TOGGLE, XOR within parameter of given instruction -— depending on the type of instruction parameter it can simulate errors during EEPROM applet data read, RAM memory access or in instruction interpreter module.

Using these operations, we can simulate transient as well as permanent faults that correspond to the virtual machine errors during the card operation and faults occurred when loading the applet into the card. As transient error we mean the fault that is introduced only once for the test execution. Other executions aren't disturbed for the test. Permanent error occurs continuously during the test. Our fault generation algorithm doesn't use any prediction to cause a particular fault effect [11]. We create all possible faults according to the guidelines of fault injection. For this reason, the number of generated tests depends on the applet code size.

Created fault injection environment determines a single test result, which can be completed as:

- correct result – inspected applet behaved as expected (correct response, correct SW),
- incorrect result – inspected applet behaved in a wrong way (incorrect response or incorrect SW); some of faults effects, which are considered as incorrect, are presented in the following section,
- fatal error – virtual machine runtime environment in the card discontinued an operation because of the unrecoverable error detected internally, the effect of this error is the lack of response from the card until next reset, card does not respond to any APDU requests,
- crash – cref simulator crashed, software malfunctions caused by introduced disturbance prevented the test completion.

We have also prepared scripts for card response analyze in case of incorrect result. In data obtained from the applet scripts are looking for sensitive information such as a PIN code or cryptographic key used by card.

In a single experiment performed on the card applet we use specific scenario of communication and specific configuration of generated faults. Each of our experiments consists of a set of tests where the same initial conditions are used. Each single test is executed on a correct instance of the investigated card applet and have the same communication scenario (APDU commands). All faults introduced in the previous tests have no impact on the current test. During the test we inject into applet code or memory area specific fault, we observe applet behavior and collect data obtained from the applet. After test execution our environment determines test result. In case of incorrect result we are looking for sensitive data in collected card responses.

During an experiment, to ensure full card applet code coverage in relation to experiment fault generator configuration, we use predictable rules for fault injection place in the code. It means that our algorithm generates all possible tests for a given code. Therefore, the number of generated test depends on the complexity of the tested applet. Disturbance parameters are generated randomly.

Fault injection environment has been developed in such way that the experiment is performed without user interaction. After the experiment user have an access to the statistics of the experiment and detailed information (result, injected faults, received data) about each of the tests.

Fault injection scenarios

Fault injection into an applet could have many different effects. Introduction of an error at the stage when the applet is outside off the card (modified source/binary application), usually has the permanent effect. Faults during the execution of the applet could be transient errors. In the case of disturbance type (fault type) action occurred might be a whole range of effects, such as:

- incorrect data during card reading – an invalid value is returned, card memory isn't read in a proper way (wrong memory address, memory corruption),
- error during data writing – invalid data is stored in the card memory,
- memory dump – reading data, which should not be accessed (memory access violation),
- fault in applet execution scheme – a part of the code fails to execute (i.e. change in goto instruction parameter to a non-existent address or in logical condition of a conditional statement),
- errors in cryptographic algorithms operations – can lead to reduce the key space, leakage of the secret key, its parts or other sensitive information.

These effects are classified as incorrect result of applet execution.

In this study we have investigated with our fault injection environment three typical Java Card applets:

- PIN verification – applet verifies PIN code given by the user, properly verified PIN code allows the user to perform the protected applet services like payment or electronic signature,
- Wallet – applet manages electronic wallet operations like credit and debit, card owner can use the applet as an electronic purse to keep the money and to make payments,
- 3DES encryption – this applet encrypts one block data with 3DES algorithm, applet provides secure storage of cryptographic key and can be used for operations like data encryption and decryption without access to a key in the plain form.

We have compiled and prepared EEPROM images with cref simulator for each card application. Then we performed experiments which were targeted at checking the behavior of investigated applet in the presence of transient and permanent faults. In each test we used base EEPROM image of the investigated applet.

PIN verification applet is a part of standard wallet sample from Java Card package. The used test stimuli is showed in Fig. 4. We send VERIFY PIN command to the applet with wrong PIN code and we expect 0x6300 SW (no authorization) as card response.

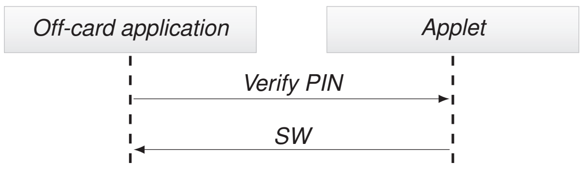

Fig. 4. PIN verification scenario

For the PIN verification scenario we performed about 11,500 tests. Duration of the experiment on a typical desktop computer was about 6 hours. For each of tests, we introduced into the applet one of following faults:

- permanent: change in the lowest bit of goto instruction parameter (jump offset), change in the parameter of logic condition instructions like ifeq, ifne, iflt, ifge, ifgt, ifle, ifnull, ifnonull and variable index change for instructions load, store, getfield, putfield and inc,
- transient: change in the lowest bit of goto instruction parameter (jump offset), random change in instruction opcode.

Wallet applet is also a part of popular example application from Java Card package. Our test scenario is showed in Fig. 5. We send CREDIT command to the applet with 50 units. Then we check balance with GET BALANCE command. Next wallet debit operation is performed using DEBIT command with 30 units. Then we check balance once again. We expect proper SW values (0x9000) for each executed command and proper wallet balance after each performed operation like credit or debit.

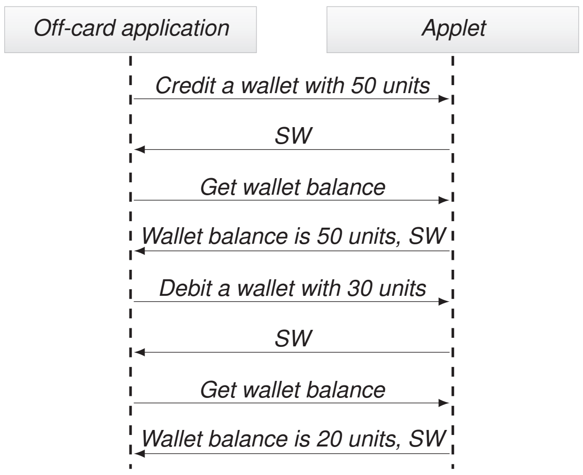
Off-card application
Applet
Credit a wallet with 50 units
SW
Get wallet balance
Wallet balance is 50 units, SW
Debit a wallet with 30 units
SW
Get wallet balance
Wallet balance is 20 units, SW

Fig. 5. Wallet transaction scenario

We performed about 58,500 tests in wallet applet experiment. Duration of the experiment was about 32 hours. For each test we introduced to the examined applet one of following faults:

- permanent: change in the parameter of logic condition instructions like ifeq, ifne, iflt, ifge, ifgt, ifle, ifnull, ifnonull and variable index change for instructions load, store, getfield, putfield and inc,
- transient: random change in instruction opcode.

3DES encryption applet is used to encrypt one block of given data with 3DES algorithm. This applet should return valid encrypted block with 0x9000 SW. The test stimuli is showed in Fig. 6.

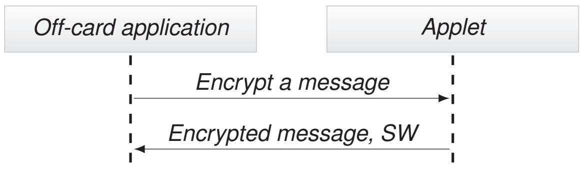

Fig. 6. Triple DES encryption scenario

We performed about 50,000 tests using 3DES applet with the same block of data to encrypt. Duration of the experiment was about 27 hours. For each test, we introduced to the applet one of following transient faults:

- change in one arithmetic instruction (add changed into sub or xor),
- random change in instruction opcode.

Experimental results

Our experiments results for each applet are presented in Fig. 7. We haven't observed PIN code or encryption key leakage for the investigated applets.

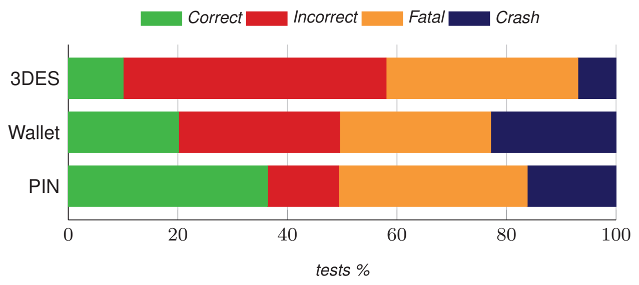

Fig. 7. Experimental results

For PIN code applet approximately 25% of tests with incorrect result terminated with status code 0x9000 suggesting the ability to bypass the PIN code verification. In that case injected fault typically lead to skip checking the PIN code method by modifying the goto or conditional instruction. It means that it would be possible to perform a protected operation like payment or electronic signature without a knowledge of valid PIN code. We haven't observed any situation where applet accepted remembered incorrect PIN for future operations. Properties of the PIN code object remained unchanged despite of the injected faults. Our additional investigation have shown that it is possible to make such change. However, level of success is very low and high precision is needed to inject that type of fault. It is necessary to change the status of the object that holds PIN code or jump address during PIN code verification.

During wallet applet investigation we observe lower level of correct results than in PIN verification experiment. This is an effect of more complex test scenario. Applet is more sensitive to the introduced errors. Within the incorrect results improper balance value was returned in about 4% of responses with SW signifying no error (no error detected by the card), of which about 70% of the cases balance was reduced to zero after the credit transaction, 18% of cases were reduced to zero after debit, and in only 12% of tests remained unchanged. We didn't observe any cases with a higher balance than 50 units.

3DES applet is most sensitive to errors. This is the result of poor fault tolerance for cryptographic algorithms [12]. Even a small disturbance leads to errors in the returned cryptogram. No part of the input buffer was exposed during the experiment. The returned RAPDU, which was different than expected, in about 75% cases consists cryptograms generated from another APDU buffer fragments i.e. including CLA and INS bytes.

In our opinion the best chances of obtaining confidential data in this case is possible with attack targeted the input APDU buffer into the card. During experiments we observed situations where as a result an encrypted piece of virtual machine memory was presented. In the absence of sufficient inspection firewall, this may allow to read data belonging to another applet or, in particular, sensitive data like cryptographic keys.

Conclusion

One of the purposes of analysis the reliability of Java Card is a description of the situation critical or unique, in which errors effects could compromise technology in terms of its safety. In the context of card applications we can analyze and verify the correctness of the code due to errors susceptibility. Simulation is one way to help us find out what are possible effects of errors.

The objective of our work was to prepare the environment to simulate hardware errors for Java Card. We used it to study applet behavior under pressure of transient and permanent faults.

During the experiments we haven't observed critical data leakage (PIN, cryptographic key). This is the result of good data protection by the virtual machine's security mechanisms like error handling and reliable applet implementation. Additionally, in case of real Java Cards, this type of data could be in protected memory areas, where access is strictly controlled and during unauthorized attempts to write or read the whole memory section could be cleaned.

Our idea or designed environment for faults simulation could be used as a tool for testing the susceptibility of the applet in case of errors that should not necessarily be the result of different types of attacks, but may be caused by malfunctioning equipment or other random situations. The experiments results can be used to determine the critical points of the applet, where additional protection should be used. Using our environment we could verify the security and reliability of the applets or test the effectiveness of the safeguards against attacks.

REFERENCES

[1] Rankl W., Effing W.: Smart Card Handbook – Third Edition, WILEY, England, 2003.
[2] EMVCo, EMV 2.0, [web page] https://www.emvco.com/, 1995. [Accessed on 15 Aug. 2013.].
[3] PN-EN ISO/IEC 7816, Identification cards - Integrated circuit cards.
[4] Chen Z.: Technology for Smart Cards: Architecture and Programmer's Guide, Sun Microsystems, 2000.
[5] Sun Microsystems, Java Card Platform Specification, 2002.
[6] Sun Microsystems, Java Card 2.2 Off-Card Verifier White Paper, 2002.
[7] Sun Microsystems, Java Card Security White Paper, 2001.
[8] Gawkowski P., Sosnowski J.: Experiences with software implemented fault injection, Proc. of the 20th International Conference on Architecture of Computing Systems, pp. 73–80, Mar. 2007.
[9] Gawkowski P., Sosnowski J.: Using software implemented fault inserter in dependability analysis, Dependable Computing, pp. 81–88, 2002.
[10] Govindavajhala S., Appel A. W.: Using Memory Errors to Attack a Virtual Machine, Proceedings of the 2003 IEEE Symposium on Security and Privacy, IEEE Computer Society, Washington, DC, USA, 2003.
[11] Lancia J.: Java Card Combined Attacks with Localization-Agnostic Fault Injection, Smart Card Research and Advanced Applications, 11th International Conference, CARDIS 2012, pp. 31–45, Graz, Austria, 2012.
[12] Nazimek P., Sosnowski J., Gawkowski P.: Checking of fault susceptibility of cryptographic algorithms, PAK 2009 nr 10, pp. 827–830.

Authors: *M. Sc. Janusz Chorko, M. Sc. Tomasz Kobylarz, Ph. D. Piotr Nazimek, Institute of Computer Science, Faculty of Electronics and Information Technology, Warsaw University of Technology, ul. Nowowiejska 15/19, 00-665 Warszawa, Poland, email: pnazimek@ii.pw.edu.pl*