USING FUZZING APPROACH TO STRESS THE SECURITY OF ISO7816 COMMUNICATION PROTOCOL DRIVERS IMPLEMENTED IN SMARTCARDS

Internship feedback for graduation
16th June 2020

Boris SIMUNOVIC

GRENOBLE INSTITUTE OF TECHNOLOGY - ESISAR
NATIONAL CYBERSECURITY AGENCY OF FRANCE

Company supervisors : Guillaume BOUFFARD and Eliane JAULMES
Academic supervisor : David HELY

INTERNSHIP FEEDBACK FOR GRADUATION

GRENOBLE-INP ESISAR 2017-2020

Keywords: cybersecurity, fuzzing, smartcards, ISO7816, embedded software development, software testing

Abstract: Smartcards are small electronic devices capable of processing and storing informations in a secure and tamper-resistant way. They are equipped with cutting-edge technologies to provide strong authentication and act as root-of-trust in bigger systems. These devices are ruling important and diverse interactions in our daily modern life by being for example involved in banking systems, telecommunications, public transportation and access control. In order to make sure those devices security mechanisms and countermeasures stay at the state-of-the-art, many governmental, industrial and academic research teams are working on this subject. In this internship a new method is proposed to asses the security and to benchmark the quality of the ISO7816 communication interfaces implemented as software executed on the card. This report describes a way to apply existing fuzz testing tools and methods to the smartcard specific case.

Mots clés: Cybersécurité, fuzzing, cartes à puce, ISO7816, développement sur cible embarquée, test logiciel

Résumé: Les cartes à puce sont des composants électroniques miniaturisés capables de traiter de l'information de maniere sécurisée. Ses caractéristiques en font une racine de confiance pour construire la sécurité à l'échelle d'un système complet, les rendant ainsi incontournables dans notre monde moderne et connecté. Nous pouvons mentionner, pour exemple, son utilisation dans les transactions bancaires, les télécommunications, les transports en commun et le contrôle d'accès.

Pour améliorer la sécurité de celles-ci, les cartes à puce alimentent de nombreux travaux de recherche. C'est dans ce cadre que ce projet propose une nouvelle méthodologie, sur la base des techniques de fuzzing, pour évaluer la sécurité des implémentations logicielles des interfaces de communication ISO7816 de ces cartes. Dans ce rapport, nous présentons une manière d'appliquer des outils de fuzzing existants et éprouvés au cas particulier des cartes à puce.

Contents

Glossary

ACK	Acknowledgment
AHB	AMBA High-performance Bus
ANSI	American National Standards Institute
ANSSI	Agence Nationale de la Sécurité des Systèmes d'Information - National Cybersecurity Agency of France
APDU	Application Protocol Data Unit
API	Application Programming Interface
ARM	Advanced RISC Machine
ATR	Answer To Reset
CMSIS	Cortex Microcontroller Software Interface Standard
CPU	Central Processing Unit
EEPROM	Electrically Erasable Programmable Read-Only Memory
ESN	Electronic Serial Number
ETU	Elementary Time Unit
FIFO	First In First Out
FTP	File Transfer Protocol
GDB	GNU Debugger
GNU	GNU's Not Unix!
GPL	GNU General Public License
GT	Guard Time
HAL	Hardware Abstraction Layer
IDE	Integrated Development Environment
IP	Internet Protocol
JTAG	Joint Test Action Group
JVM	Java Virtual Machine
LRC	Longitudinal Redundancy Check
LSC	Laboratoire de la Sécurité des Composants - Hardware Security Lab
MSISDN	Mobile Station Integrated Services Digital Network
NFC	Near Field Communication
OIV	Opérateur d'Importance Vitale
OSI	Open Systems Interconnection model
RAM	Random-Access Memory
ROM	Read-Only Memory

SDE	Sous-Direction Expertise - Expertise Department
SGDSN	Secrétariat Général de la Défense et de la Sécurité Nationale
SIM	Subscriber Identity Module
TACS	Total Access Communication System
TCG	Trusted Computing Group
TCP	Transmission Control Protocol
TPDU	Transmission Protocol Data Unit
TRNG	True Random Number Generator
UART	Universal Asynchronous Receiver-Transmitter
UML	Unified Modeling Language
USB	Universal Serial Bus
WT	Wait Time

Remerciements

Je souhaite profiter de ce rapport de stage qui clôture mes études universitaires pour remercier les personnes qui m'ont entouré durant ma formation d'ingénieur, à l'école comme en entreprise.

Je pense tout particulièrement au laboratoire de la sécurité des composants de l'ANSSI, à Eliane Jaulmes et Guillaume Bouffard, mes encadrants, pour m'avoir fait confiance et avoir choisi de m'intégrer dans l'équipe comme leur premier apprenti.

J'y ai côtoyé durant trois ans des collègues formidables qui ont montré énormément de patience et de bienveillance, qui se sont montrés tout particulièrement disponibles pour échanger aussi bien à propos de mon projet qu'à propos de leurs propres activités de recherche. Des personnes passionnées qui m'ont donné le goût pour des disciplines variées telles que l'analyse side-channel, les attaques en fautes, la cryptographie, les statistiques, l'intelligence artificielle, l'architecture des processeurs, les cartes à puce et la sécurité des logiciels embarqués. Je pense à mes encadrants, mais également à Adrian Thillard, Louiza Khati, Emmanuel Prouff, Ryad Benadjila, David El Baze, Thomas Trouchkine et Karim Khalfallah.

Je veux remercier mes enseignants de l'Esisar ainsi que mon tuteur académique David Hely. J'ai fréquenté cette école durant cinq années mémorables, j'y ai rencontré de bons amis, et aussi des professeurs qui m'ont marqués, qui m'ont transmis leurs connaissances et qui m'ont servi d'exemple. Je pense aussi au personnel de la scolarité et aux enseignants qui m'ont soutenu dans des moments difficiles, notamment Nathalie Fulget, Nicolas Charroud, Romain Siragusa, David Hely et Mylène Chavarot, mais aussi la Fondation Grenoble-INP, en particulier Murielle Brachotte.

Je souhaite également remercier mes parents pour m'avoir transmis le goût de l'effort et cette avidité de comprendre les choses qui m'entourent. Pour finir, je remercie les personnes m'ayant aidé dans la relecture de ce rapport, notamment Jean Bonnevie, Christopher Lievin, Emeline Leveaux, Mehdi Hidri, Guillaume Bruchon, Benoit Etcheberry ainsi que mes encadrants.

Chapter 1

Project definition and goals

1.1 Introduction

This report is closing a three years period of education at the Grenoble national institute of technology and of apprenticeship at the national cybersecurity agency of France to obtain the engineering degree. This document goes back on three years of skills upgrading about the smartcards subject and presents the final internship project consisting in fuzzing smartcards communication interfaces.

First chapters of this report aim to explain project stakes and context and to settle all the necessary background information to understand the problematic and proposed solutions. Especially, it will bring a particular attention to the definition of a smartcard and its communication protocols. In a similar way it will go through the concept of fuzzing, explain why this methodology is relevant to the given situation. Subsequent chapters demonstrate the technical solutions developed during the internship to apply fuzzing discipline to smartcards and finally, results are presented and next steps for further work are identified.

1.2 The French national cybersecurity agency and its research facilities

The two latest defence white papers ordered in 2008[3] and 2013[4] by the French presidents Nicolas Sarkozy and Francois Hollande are both mentioning the strategic aspects around cybersecurity. It is clearly stated as a priority in the next years for the sovereignty and security of the nation. The politics and decision makers are advised to invest the resources to develop the trust, reliability and the security in the systems of informations. As a result of these politics the National Cybersecurity Agency of France (*ANSSI*) was created in 2009 and was given a great power to coordinate all the efforts in cybersecurity on a country-wide scale.

ANSSI is part of the General Secretariat for Defence and National Security (*SGDSN*), an interministerial body placed under the authority of the French Prime Minister which "assists the head of government in designing and implementing security and defence policies"[2]. The agency was given various missions

including anticipation of the threats against systems of information, defence of governmental infrastructures and networks, protection of the vital operators (*OIV*), pro-active response to cybersecurity issues, coordination between state entities, enhancing security awareness and education[6].

The internship takes place in the expertise department (*SDE*) of the agency. It is mainly structured as an aggregation of research laboratories. The goal is to have teams of researchers who are permanently at the state-of-the-art on critical designated subjects to be able to provide expertise to other departments or entities. They are also given the mission to stimulate and orient the academic and industrial research in their respective areas at the scale of France and international community. The research efforts led by the agency are often published in academic conferences and journals which enables peer reviews by the international scientific community, thus reinforcing the scientific grounding and credibility of the work accomplished by the agency. This also provides scientific references to the industrial partners[5] and helps to improve the overall security by making the informations widespread and publicly available.

More precisely, the internship takes place in the hardware security lab (*LSC*) whose work aims to improve the security of hardware components like cryptographic accelerators, secure elements and processors against a wide range of physical attacks like side-channel and fault attacks and keep protected critical assets like encryption keys.

1.3 Interest for smartcards and goals

Smartcards are small physical components with the ability to communicate, store and process data. They usually aim to provide secure and tamper-resistant storage to protect cryptographic assets and strong security primitives like cryptographic accelerators. As they are designed solely for this purpose, they are often part of a bigger system and act as a root of trust. Smartcard characteristics are going to be discussed in depth in Section 2. Given its wide spread, its critical position inside systems and its hardware nature, it naturally falls in the field of interest of the hardware security research lab (*LSC*).

This project is focused on the communication ability of the device. It is the natural way to interact with the card, to execute commands on the card and to access informations. Thus, it could be one of the biggest and most straightforward entry-point for a potential attacker, its implementation has to be very secure, robust and the protocol specifications have to be strictly followed. This research project aims to find new ways to asses the security level of the software implementations of the communications drivers in current card implementations.

1.4 The functional needs

The smartcards have several standardized ways to communicate with the outside world. The two most popular ones are the wireless protocol defined by the ISO14443[9] using a close to *NFC* technology and the second one, the most deployed one is the wired protocol defined by the ISO7816[12] standard. In this project, we focus on the ISO7816 protocol detailed in Section 2.

The goal is to develop a methodology and tools to be able to assert the quality

of the software implementation of the ISO7816 protocol of any smartcard. The approach has to be automatable and reproducible. The developed tool has to stress the implementation of the protocol to asses how well it is compliant with the specifications and especially how well it defends itself against atypical and malformed inputs. The tool has to produce reliable, interpretable and comparable metrics in order to be able to take decisions about the quality of the driver. It is also expected to detect when an input produces unexpected behaviour and gather all the necessary diagnostic informations to reproduce it and fix it.

1.5 The attacker model

When dealing with security, it is often important to make a model of the opponent, especially how powerful he is, how much resources he can invest, how much informations about the target he can access. The goal is to identify and categorize its abilities. This model is then used to design the system and to introduce the proper countermeasures. A very common way to categorize an attacker is focusing on how well he is informed about the target. How much information he can have about the design of the target and how much informations he can access during the runtime of the device. People mainly agree on three categories.

The first one is the white-box model with the highest assumptions on the opponent. In this scenario, the attacker has all the informations he wants about the target, he has access to all the datasheets and he can instrument the target, gather a lot of informations about its internals during the runtime of the target device. The attacker has a clear view of the design and of what is happening inside the target. As the assumptions are the highest, the attacks considered in this model are potentially the most sophisticated and powerful and they give the best appreciation of the security level.

The second one is the black-box model with the lowest assumptions about the knowledge of the attacker. It is assumed that the attacker has no informations about the design and about what is going on inside the target. He is only aware of the input applied to the "black-box" and its output.

There is an intermediate grey-box model. In this case, the attacker is supposed to have partial informations about the design and he is able to obtain some intermediate computation results or measure some side-effects.

Nowadays from a designer point of view, in extension to the Kerckhoffs rules[13], the approach is usually to make the higher assumptions about the attacker and to design the security as if the attacker knew everything about the target. The system is then called "secured by design", the developer does as if the design was publicly available, the security is brought by a smart design itself rather than by opacity. This is a relatively new approach in hardware design and it is opposed (and has proven to be better) to the old scheme of security by opacity where security is broken from the moment the attacker access critical informations (through social engineering for example), thus breaking the designers assumptions about the attacker.

From the attacker point of view, the tendency is rather to make less assumptions about the information it can access, thus he can make his attack scenario more general and less dependent on sometimes hard to get or expensive inform-

ation. Given the very limited access to smartcards applications and operating system and the very restricted access to documentation, the black-box model was naturally chosen in order to stay coherent with the open-sourcing goal. This is a way to check the results an attacker could obtain even if he knows nothing about the target.

1.6 Discussing the open-source nature of the project

The project is meant to be open-sourced. The long-term goal is to make the code, the documentation, the design and the tools freely and widely available to the academic community, the industrials, the citizens and the certification and governmental bodies. Everyone has to be able to build the tool, use it, modify it and contribute. Thus, this project will contribute to improve the academic state-of-the-art in this discipline, it provides tools to the designers and certifications bodies to stress the security of their implementations. It enables the clients to tests on their own the quality of the products they are buying and it compels the industrials to produce devices which are resilient to state-of-the-art attacks with publicly available tools. From the overall point of view and from the citizen point of view, it brings more transparency and a greater trust into the targeted systems.

From a project management point of view, open-sourcing is a way to leverage more resources by interesting other people in order to contribute and make the project go further. It ensures more sustainability by having more people working on it from several entities and companies. It is not dependent on a single team or a single person who can leave the company. It also introduces a way of working proper to collaborative open-sources projects like rigorous methods of versioning and of documentation writing.

Chapter 2

Introduction to smartcards and their communication protocols

2.1 The need for a secure elements and a root of trust

Before giving the functional and physical definition of a smartcard in Section 2.2, they are several examples to explain how came the need for such components. This helps to understand the criticality of those devices and the reason why the LSC is focusing on this important field of research. The purpose of this section is to go through the examples of the mobile telephony and of the banking systems in order to make the reader to understand the role the smartcards are playing in overall systems.

In the early 1980s, one of the first mobile phone systems was in use in the United Kingdom. It was called *TACS* standing for "Total Access Communication System"[8] and it was a pretty good achievement with the technologies of its time so the focus was not really on the security and the confidentiality. The authentication of the user on the network was quite simple. The mobile handset was storing two numbers in its internal memory. The first one, the Mobile Station ISDN number (*MSISDN*) is more or less equivalent to nowadays phone number, it is used to dial with a specific user. The second one, the Electronic Serial Number (*ESN*), is a unique identifier for the handset. During the authentication process, the phone was sending over the network the two numbers, the network was then comparing the received pair with the ones stored in its central database. If the *ESN* serial number of the phone was matching the *MSISDN* number it is supposed to be attached to, then the access was granted and the handset authenticated over the network. The security stopped so far. There was no encryption process to ensure the confidentiality of the *MSISDN* and *ESN* identifiers being sent on the network and no software or hardware mechanism to prevent somebody (not being the service provider) from re-writing those numbers in the internal memory. It was then possible for a third-party to eavesdrop the authentication process, to steal the two numbers and write them inside their

own phone and then usurp the identity of the victim and do phone calls for free. In the previously described case, there are several noticeable issues : the user of the handset can not be trusted, there is a need to have a secure piece of memory with read and write operations protected from an unauthorized person. It has to be protected against physical means and software attacks. There is also a need for cryptographic primitives to provide encryption. Those observations conducted to the usage of *SIM* cards which are smartcards. In the banking environment, similar situation are encountered with the credit cards. There is a need to securely store the customer identifiers and prevent a third-party to steal and copy signature keys. Globally, there is a need for a secure component providing security primitive in order to build an overall trust in the system.

2.2 What is a smartcard ? Definition and description

As the form-factor of a smartcard can vary, its functional definition is more relevant in the case of this project. In [8], it is defined as an object which :

1. can participate in an automated electronic transaction
2. is used primarly to add security
3. it is not easily forged or copied
4. can store data securely
5. can host/run a range of security algorithms and functions

Every computer system is built with multiple levels of abstraction and generally, higher layers must trust lower layers. The initial source of trust is called the root of trust[15]. The Trusted Computing Group (*TCG*) encourage building secure systems with a "physically secure trusted component that can be used as a foundation upon which trust in the rest of the system can be built"[8, p. 157]. All the entities in the overall system interact with this secure module and can thus be given a good level of insurance that it is behaving as expected. The smartcards, with the functional description aforementioned, are designed to provide this root of trust.

A smartcard is typically composed of a few square milimeter large piece of silicon embodied in a piece of plastic shaped as a card as depicted in Figure 2.1. It contains a microcontroller able to process information, cryptographic accelerators, *TRNG* and memories like *ROM*, *NVM* and *RAM*. It is also provided with metallic connectors to be able to communicate with an external card reader via the ISO7816 protocol described further in Sections 2.6 and 2.7. It eventually embodies an antenna for *NFC* communication using the ISO14443 protocol. The shape and physical characteristics of the card are standardized in ISO7816-2[11]. When dealing with the wired communication, the electrical characteristics and low level communication protocols are defined in the ISO7816-3[12]. Then, the ISO7816-4 standard gives a functional description of the application level communication protocol used both in wired and *NFC* communications. It is based on the *APDU*s described in Section 2.5.

In both contact and contact-less modes, the card interacts with a reader device. Its role is to provide power and clock to the card and initiate the

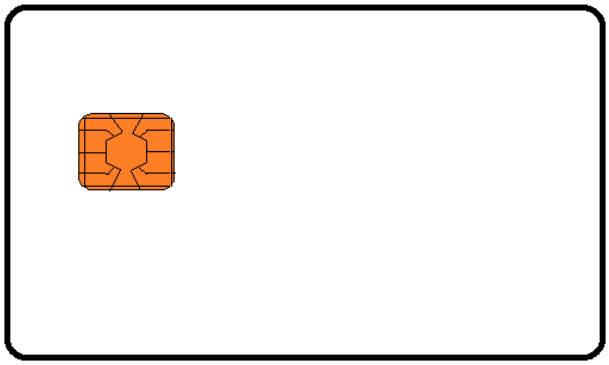


Figure 2.1: Typical smartcard shape with its contacts for providing ISO7816 communication. Klipe / CC BY-SA (https://creativecommons.org/licenses/by-sa/4.0)

transactions. It typically request the card to read or write informations and execute actions like a cryptographic computation. The card can be compared to a web server[1, p. 21], it reacts on requests, stores and serves the information requested by the reader while making sure that the information are served to a strongly authenticated user.

2.3 Electrical interface

When using the wired communication protocol, the card is connected to the reader by the metallic contacts depicted in yellow on Figure 2.1. This connector is composed of several pins detailed on Figure 2.2 and their purpose is defined in the ISO7816-3 standard[12].

The pin C5 (GND) is used to share a common ground, a reference electric potential, across the reader and the card. It is used for both the power supply and the communication. As long as the card is powered by the reader, the pin C1 (VCC) is used to provide power supply to the card. Depending on the use-case it can be 1.8V, 3V or 5V. The card does not have its own clock source, the reader is in charge of providing the common time source through the C3 (CLK) input pin. It is specified to vary and take any value between 1MHz and 5MHz. The C2 (RST) is an input pin used by the reader to send a reset signal to the card. The C7 (I/O) pin is the data transmission line used to exchange informations with the protocols described in Sections 2.6 and 2.7. It is both an input and an output because the communication protocol is half-duplex. All the other pins are reserved either for future use or for proprietary use.

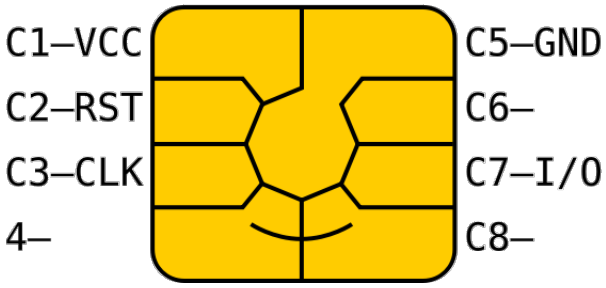


Figure 2.2: Smartcard ISO7816-3 connector pinout. Dacs, WhiteTimberwolf / CC BY-SA (https://creativecommons.org/licenses/by-sa/3.0)

2.4 Operating system structure

To provide all the functionalities described in Section 2.2, the microcontroller inside the card is running various pieces of software. Figure 2.3 gives a top-level view of the different software layers of abstraction on a JavaCard technology smartcard. First of all, the chip manufacturer provides a hardware abstraction layer (*HAL*) with the microcontroller. This piece of software eases the access to the hardware resources (like for eg. the *UART*/ISO7816 interface). It theoretically also makes the following layers independent from the underlying hardware. Then, on the top of the *HAL*, there is an operating system in charge of managing the physical resources like memory and *CPU* time. It is also in charge of dealing with the ISO7816-3 communication interface via a dedicated driver. It then exposes this service to the upon applications via a system *API*. Depending on the implementation, this driver is usually executed in the kernel space. It could be partially or not isolated. This makes it critical from a security point of view, so this is the main part of code whose security has to be stressed with the fuzzing approach. Then, on the top of this operating system, usually runs a Java virtual machine (*JVM*) which emulates the java instruction set to run multiple java applets and ensuring java security policies. On the top of the operating system and the *JVM* are running one or more Java applets, computer programs defining the behaviour of the smartcard. Depending on the use-case, those can be loaded in by the manufacturer, the card issuer or the user.

According to the attacker model defined in Section 1.5, in the frame of this project, there is no access to the operating system to get informations about how the driver under test behaves, no access to any logs or internal system administration tools. Nor is it possible to have an accomplice applet on the target.

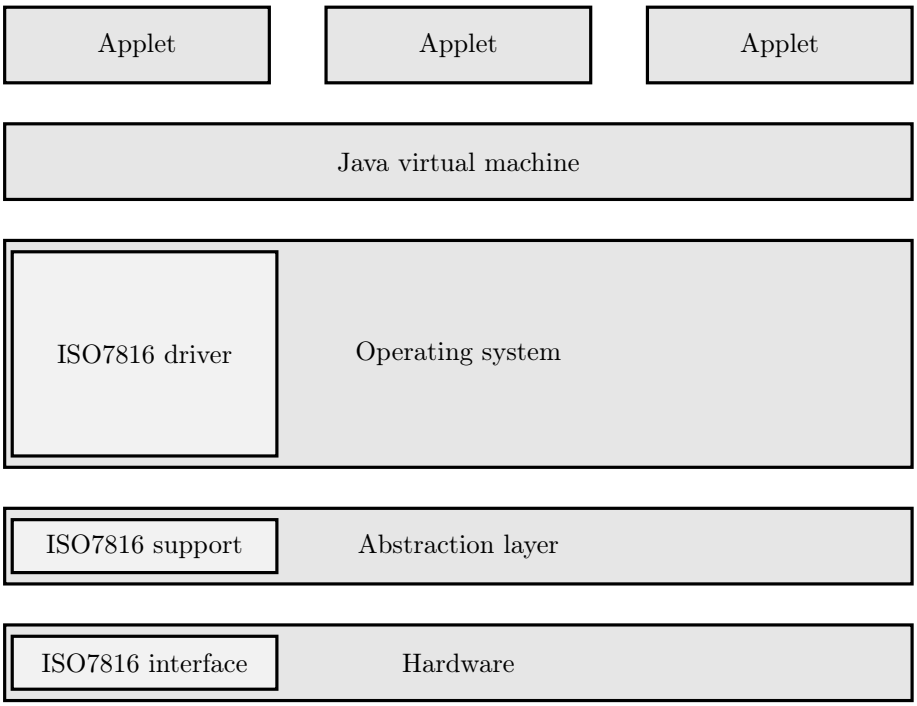


Figure 2.3: Top-level software architecture of the smartcard using te JavaCard technology.

2.5 The Application protocol data unit (*APDU*)

The communication between the reader and the card is organized upon several layers. Each layer brings a new service and a new abstraction level. The top-level most abstracted unit described by the ISO7816 is the Application Data Protocol Unit (*APDU*). It can be compared to the "application layer" described in the well known *OSI* model[10]. It directly carries application data and operations to be executed by the card. These can be intended to the operating systems, for example for file system management and system administration or can be intended as well to specific applets running on the card. It is organized as a client-server model, where the card reader sends requests and commands to the card whose role is to respond, serve information and execute the requested actions.

An *APDU* is a well defined data structure organized as a sequence of bytes. It can be represented as a frame like described in Figures 2.5 and 2.6. There are two kinds of *APDU*s : the *APDU* commands and the *APDU* responses. As shown in Figure 2.4, the reader is always initiating the communication asking the card to do something by sending an *APDU* command. Then, the card is supposed to give an answer in the form of an *APDU* response. The roles can not be inverted.

The *APDU* command (illustrated in Figure 2.5) is composed of a mandatory header and a optional body. Structure and size of the header can not vary, it is always composed of four fields of information designated by the names : CLA,

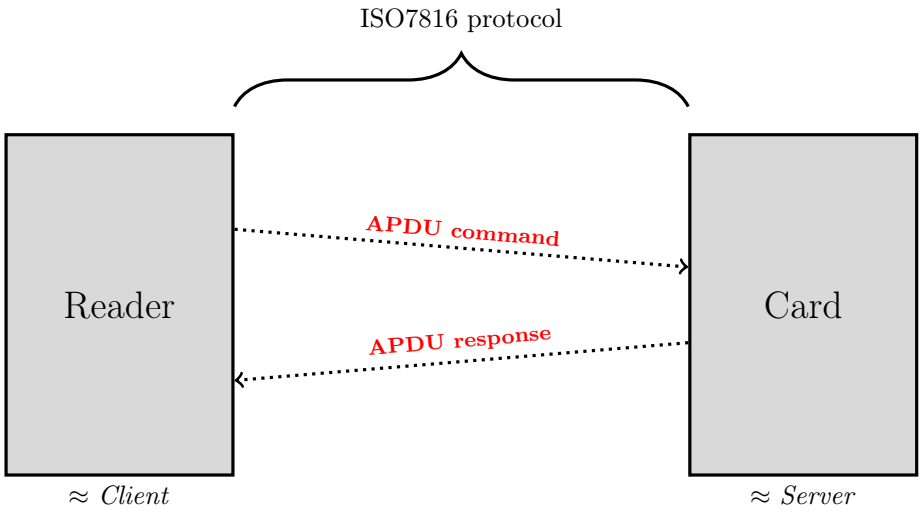


Figure 2.4: Illustration of a typical *APDU* transaction. The reader sends an *APDU* command to the card. The card answers back with an *APDU* response.

INS, P1 and P2. The CLA byte carries information about the class of the instruction. INS is a byte coding for the instruction itself, it corresponds to the action to be executed by the operating system. For example running an applet or getting the content of a file. P1 and P2 bytes contain the operands and options associated to the previous INS byte. Then, the body is composed of the *Lc*, *Le* and data bytes from Figure 2.5, they are all optional. *Lc* can be one or two bytes and it encodes the number of data bytes carried in the following data field of the *APDU* command. *Le* can also be one or two bytes, it encodes the number of data bytes expected as a response to this command.

The *APDU* response illustrated in Figure 2.6 is sent by the card when it has finished to process the previously sent command. It is first composed of an optional body eventually containing data returned by the card. It is then followed by a mandatory two bytes trailer. These bytes called SW1 and SW2 encode an execution code indicating the status of the card, it usually says if the command was processed correctly or if an error has occurred.

The bytes of information composing the *APDU* command are not directly pushed in the transmission line. This is only a conceptual abstract object from the application layer, it is not understandable by the physical components in charge of emission and reception of bits. It has to go through several intermediate steps in order to serialize the data and deal with the reality of the physical transmission line. This is partially the purpose of the underlying link layer protocol which encapsulates the bytes of information from the *APDU*. Smartcards have the particularity that the ISO7816-3 standard defines two different protocols to do the same task as pictured on Figure 2.7. They are respectively called the T=0 and T=1 protocols, they are both discussed in Section 2.6 and Section 2.7.

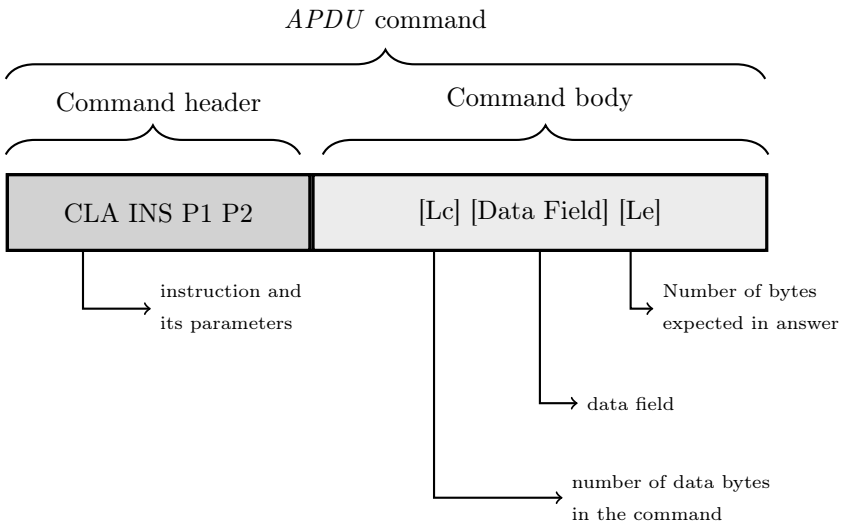


Figure 2.5: Illustration of the *APDU* command structure

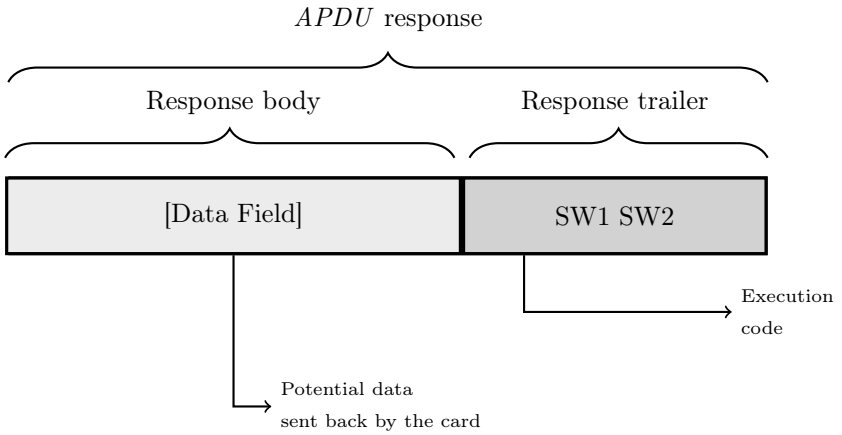


Figure 2.6: Illustration of the *APDU* response structure

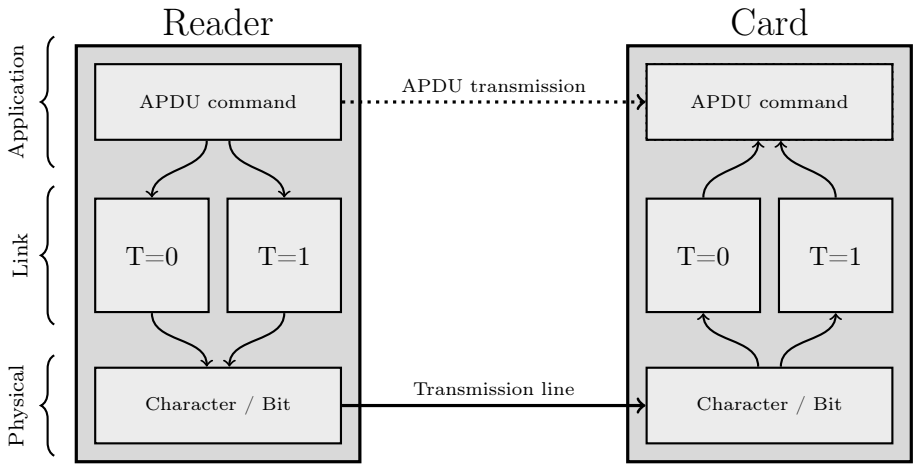


Figure 2.7: ISO7816 protocol is built upon several abstraction layers of abstraction. The *APDU* objects can be carried by either the T=0 and T=1 underlying protocols.

2.6 The ISO7816-3 T=0 communication protocol

The aim of this section is to briefly discuss T=0 protocol. This protocol can be roughly compared to the "link layer" defined in the well known *OSI* model[10] even if in practice the isolation of this layer from the other ones is not very strong. Its purpose is to enable the transmission of the *APDU* abstract object over the transmission line.

This protocol operates by decomposing the *APDU* object into intermediate structures called *TPDU* commands. As illustrated in Figure 2.8, the *TPDU* can be respresented as a sequence of bytes. Its header is a sequence of five bytes CLA, INS, P1, P2 and P3. The four first ones are matched on CLA, INS, P1 and P2 from the related *APDU* structure defined in Section 2.5. P3 byte is used to encode the number of data to be transferred during the command.

There are different scenarios depending on the *Lc* and *Le* values defined in Section 2.5 of the *APDU* structure. The exchange of *TPDU*s between the reader and the card is not going to be the same depending whether data have to

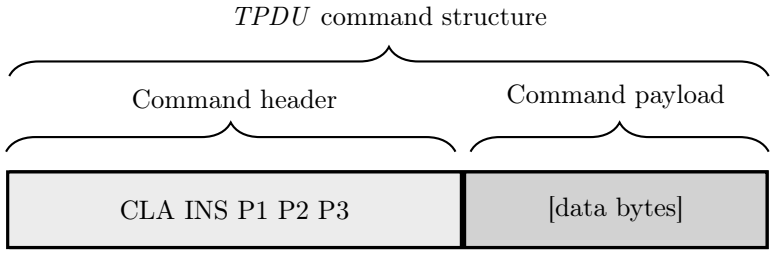


Figure 2.8: Illustration of the *TPDU* command structure. CLA, INS P1 and P2 are same as the *APDU* defined in Section 2.5. P3 encodes the number of data bytes.

Case 1	No command data field	No response data field
Case 2	No command data field	Response data field
Case 3	Command data field	No response data field
Case 4	Command data field	Response data field
Case 2E	No command data field	Extended response data field
Case 3E	Extended command data field	No response data field
Case 4E	Extended command data field	Extended response data field

Figure 2.9: The four structures of command *APDU*s.

be sent to the card or not and depending whether data are going to be expected back from the card, how many data are expected, if both entities agree on the amount of data etc... The possible scenarios are detailed in Figure 2.9. Usually several *TPDU* commands and responses are exchanged between the reader and the card to process a single *APDU*. Typically in the case 4 from Figure 2.9 (where the processing of the *APDU* command requires to send data bytes in the direction of the card and then to get back data bytes from the card as a response), a first *TPDU* is sent with the original command described in the related *APDU* (CLA, INS...), with the data bytes which have to be sent, then, when the card is ready, the reader sends another *TPDU* command to request the answer from the card. The sequence of bytes fixed by the T=0 protocol goes through the physical layer detailled in Section 2.8.

The reader first initiates the transaction by sending the first five bytes of the header in a row. Then it waits for the card to answer back a so called "procedure byte". It can be one of the four subsequent possibilities :

- 0x60, is a null byte, it tells the reader device the card needs more time to process the instruction. It resets the timeout counter.
- 0x6x or 0x9x is an *APDU* response SW1 byte as defined in Section 2.5. The reader is then supposed to continue to listen to receive the subsequent SW2 byte.
- The same INS as the previous *TPDU* header (as defined in Figure 2.8). It is an *ACK* byte, the reader is expected to send all the *TPDU* data bytes in a row.
- The same INS as the previous *TPDU* header xored with 0xFF is an *ACK* byte. The reader is expected to send only the next byte of the data field and then to wait for another procedure byte.

2.7 The ISO7816-3 T=1 communication protocol

The T=1 protocol was designed to replace T=0 by bringing some improvements, however T=0 is still largely in use nowadays. T=1 protocol is designed to fulfil the same (link layer) functional needs than T=0 (detailed in Section 2.6), however it is designed to improve some lacks in the previous protocol, especially the isolation between the abstraction levels of the protocol stack. It makes a clear distinction between the physical layer, the link layer and the application layer. It is block oriented and improves error correction capacity. For the reasons

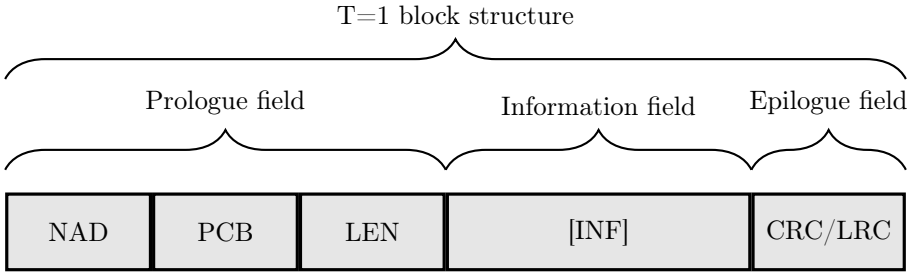


Figure 2.10: Illustration of a T=1 protocol block as defined in ISO7816-3[12]. The block is composed of two madatory fields : epilogue and prologue and one optionnal field : data. NAD, PCB and LEN are one byte long each. CRC/LRC is on or two bytes depending on the checksum algorithm. Data field is up to 254 bytes.

exposed later in Section 4.1, the security testing efforts of this project are going to be focused on that part of the ISO7816 protocol.

T=1 protocol is block oriented, that is, the *APDU* structure described in Section 2.5 is cut in few bytes pieces which are then encapsulated in T=1 block structures illustrated in Figure 2.10. This block structure is composed of three byte fields : the prologue field, the information field and the epilogue field. The prologue is a three bytes long mandatory field, it carries control informations and metadata about the content of the block. The Node Address byte (NAD) is used in the case of several devices on the same shared bus and is an advanced functionality of the protocol. The Protocol Control Byte (PCB) encodes informations required to control the transmission, especially the "type" of the current block among the three following possibilities :

- I-Block, meant to carry a payload composed of data bytes corresponding to pieces of *APDU*s.
- R-Block, used as acknowledgement or error signaling.
- S-Block designed to negotiate communication parameters between the reader and the card.

The LEN byte purpose is to encode the length of the subsequent information field between 0 and 254. The information field is designed to carry the payload encapsulated inside the current block (upper layer protocol data). The block finally ends with a correction code, it can be one or two bytes long depending on the kind of error detection/correction currently in use. It can be either Longitudinal Redundancy Code (1 byte LRC) or Cyclic Redundancy Code (2 bytes long CRC).

I-Blocks are used to carry informations and data. If the maximum size of the block is not enough to fit all the data bytes, then several blocks have to be sent in a so called "chaining process". For this purpose, the I-Block PCB contains a "sequence number bit" counting (modulo two) the sequence number of the current block. It contains as well a "more data bit" used to indicate to the other device that there is more data to be transmitted, and so, this is not the last I-Block. After the reception of a chained I-Block, the device uses an

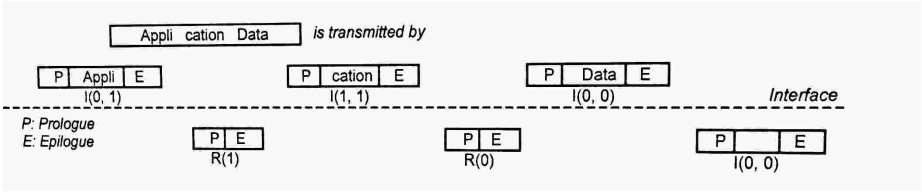


Figure 2.11: Illustration of the chaining process taken from the ISO7816-3[12] standard. It represents the "application data" frame being split in three distinct blocks. I(M, S) is for an I-Block with more data bit M and sequence number S. R(S) is for an R-Block requesting the I-Block with sequence number S.

R-Block to acknowledge it and to request the next one by explicitly indicating the requested sequence number in the R-Block structure.

R-Blocks don't carry any information field, they are sent as a response to a previous block. They are used to acknowledge it or to ask the next one. If something went wrong, for example a CRC error or wrong sequence number received, their aim is to request again the block with the expected sequence number. In their PCB, they carry a sequence number bit for this purpose.

S-Blocks are used to re-negotiate T=1 communication parameters. There is the four following kinds of S-Blocks and they can be sent either by the reader and the card. Each of them exists in its request form and its response form. One device initiates the negotiation by sending an S-Block request, then the other device can either acknowledge it or refuse by sending back an S-Block response. S-Blocks eventually contain a 1 byte long information field encoding the requested value for the parameter :

- IFS request and IFS response : is used in order to negotiate the maximum size (in number of bytes) of the I-Blocks information fields.
- WTX request and WTX response : is used in order to negotiate timeout value in between blocks.
- RESYNCH request and RESYNCH response : after several protocol errors (like non-acknowledgement, wrong sequence numbers etc...) a device can request a resynchronization, the protocol state machine goes back to a known state.
- ABORT request and ABORT response : if after several errors and resynchronisation attempts there is no improvements, then a device can request to completely abort the transaction with this block.

2.8 Transmission of characters

In the *OSI* model[10] the "physical layer" corresponds here to the transmission of characters/bytes and bits on the transmission line (I/O pin C7 from Section 2.3). The ISO7816-3 standard describes how to transmit a character in the I/O line, it is quite similar to the standard *UART* protocol with some differences which are going to be discussed in this section.

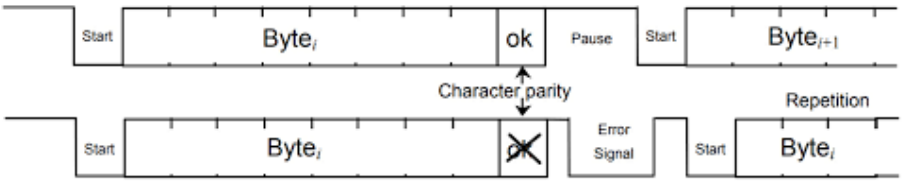


Figure 2.12: Illustration of a character frame taken from the ISO7816-3 standard[12]. It is possible to see the ten moments composing the character frame, the guard time and the error signal.

First of all, it is not exactly asynchronous because there is a shared clock across the two devices via the pin C3-CLK (see 2.3), the two devices agree on an elementary time unit (ETU) which is a multiple of the shared clock period. This number of ETU per second is equivalent to the baudrate in $UART$ protocol. A character is composed of 10 moments followed by a pause time as depicted in Figure 2.12. A moment is defined as a 1 ETU period of time where the I/O line is either at the state high or low. The pause time is defined as a several ETU time period where the reader remains in reception mode. The first moment is a start bit, the I/O line should be pulled to low state by the reader. Then, the next 8 moments are coding for the byte of data to be sent. The 10th moment is defined to be a parity bit to enable some basic error detection. After the last moment the reader should remain in the pause state during at least two ETU. This pause can be used by the other device to pull the I/O line to low state to signal an error, the character is then sent again. The standard also defines a guard time (GT) and a wait time (WT) and constrains on the time precision of the ETUs. The GT is a minimum delay to be respected between the leading edges of two consecutive characters and WT is the maximum delay between the leading edges of two consecutive characters. It allows to detect an unresponsive card.

Another big difference is that the communication is half-duplex, both the card and the reader are using the same transmission line to emit characters. It is assumed that (because of the higher level protocols described in Section 2.6 and Section 2.7) both devices know when it is their time to emit a character, so no collision should happen. Thus, there is no need for an medium access management protocol.

Chapter 3

Introduction to fuzzing

Fuzzing is a software testing discipline, meaning it is a set of methods aimed to help increase the level of insurance that a particular piece of software is behaving as it was specified to. These tests are not specifically security oriented, it could be used to test performances like execution time and memory usage to identify situations where computer resources are used in an abnormal way while other kinds of tests aim to make sure the outputs provided by the tested software are those specified by the functional needs. Finally, it is also used to check if the target is resilient to malicious inputs aimed to trigger non intended behaviours. This project is mainly focused around the two last points. Fuzzing mainly differs from conventional testing by the fact that the test routines are not hand written accordingly to a specification, they are rather automatically generated and checked by an algorithm. The purpose of this section is to introduce the reader to fuzzing methodology and to settle down the necessary vocabulary and background information.

3.1 Terms and definitions

In the fuzzing literature and in the very numerous available tools, the involved terms are quite diverse and heterogeneous. This Section defines the most important vocabulary to describe the fuzzing process mainly accordingly to [14].

The Program Under Test (P.U.T) is the software implementation currently being tested, it is represented on the right side in Figure 3.1. Fuzzer is another entity external to the P.U.T represented on the left side, it performs the fuzz testing (or fuzzing) by interacting with the P.U.T by applying inputs. Fuzzer has access to the program output and depending on the attacker model (discussed in Section 3.4) it can eventually have an embedded knowledge about target implementation or access information about program execution. Then, "*Fuzzing is the execution of the P.U.T using input(s) sampled from an input space (the "fuzz input space") that protrudes the expected input space of the P.U.T*"[14]. This means that the "fuzz input space" is not equal to the initially specified or expected input space. The generated inputs can be a part of the expected input space but could also be out-of-specification, malformed and unexpected inputs which may be processed incorrectly and may trigger unintended behaviours. A test-case is one element taken from the "fuzz input space" which is going to be

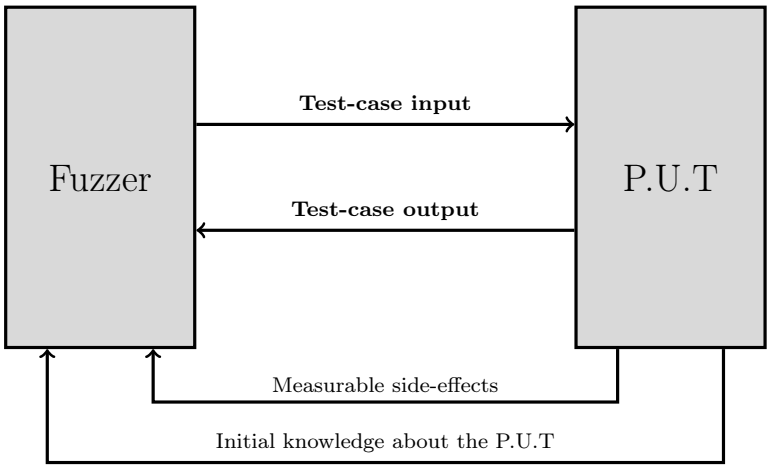


Figure 3.1: Illustration of the interactions between the fuzzer and the P.U.T (Program Under Test).

applied to the P.U.T. The fuzzing algorithm detailed in Section 3.3 sequentially chooses test-cases in the "fuzz input space", applies them to the P.U.T in a process called a fuzz iteration. The bug oracle is the part of the fuzzer in charge of discriminating on the basis of the feedback informations if such an unintended behaviour happened. The fuzz configuration is a set of parameters controlling fuzzer execution, it is mainly composed of informations and description about the input space. Different ways of describing it are presented later in Section 3.4.

3.2 The goal of the fuzzing algorithm

Typically the fuzz input space depicted in Section 3.1 is extremely large or even infinite. Naive fuzzer implementations randomly sample this space. It is not really efficient because the number of fuzz iterations achievable in a reasonable finite time is not significant compared to the space size. A smarter fuzzing process would rather implement a strategy to bring out the most promising test-cases and try them first. Most of state-of-the-art implementations can be compared to an optimisation problem. First, the operator chooses a metric correlated with the goal he wants to achieve, it could be for example a code coverage ratio, the number of conditional branchments looked over or the number of unit bugs found. Then, the fuzz algorithm goal is to try to browse the fuzz input space in such a way that it maximises or minimizes this metric. Choice of the metric is decisive, each of them has advantages and counterparts. For example, a lot of available fuzzing tools are using code coverage which is easy to compute but not always relevant because it gives the same weight to benign portions of code and to critical ones. It also easily sticks into a local maximum. Most sophisticated fuzzers implements evolutionary algorithms[14], machine learning and deep learning methods[7] to learn in real time how the P.U.T reacts and which test-case should be chosen next. Quality of the metric and of the optimisation is directly correlated to the amount of available informations about

P.U.T execution and to the attacker model.

3.3 A general fuzzing algorithm

```
1  Input: ℂ, t_limit
2  Output: 𝔹 // a finite set of bugs
3
4  𝔹 ← ∅
5  ℂ ← Preprocess(ℂ)
6
7  while t_elapsed < t_limit ∧ Continue(ℂ) do
8          conf ← Schedule(ℂ, t_elapsed, t_limit)
9          tcs ← InputGen(conf)
10         // O_bug is embedded in a fuzzer
11         𝔹′, execinfos ← InputEval(conf, tcs, O_bug)
12         ℂ ← ConfUpdate(ℂ, conf, execinfos)
13         𝔹 ← 𝔹 ∪ 𝔹′
14
15 return 𝔹
```

Figure 3.2: A generalization of a fuzz testing algorithm taken from [14].

In [14] is presented a fuzzing algorithm which generalizes well most of the existing fuzzing processes and helps to understand the method implemented in this project. It is presented in Figure 3.2 and takes as arguments a fuzzing configuration $\mathbb{C}$, a finite time bound t_{limit} and outputs a set of bugs $\mathbb{B}$.

The Preprocess() instruction at line 5 is in charge of preprocessing the fuzz configuration $\mathbb{C}$. Typically it analyses the description of the input space and tries to minimize it to save execution time. Depending on the scenario and the attacker model this instruction also instruments the code to be able in the next steps to gather feedback information about P.U.T execution.

Then, at line 7 the fuzzing process itself starts and runs until the time limit is reached or until the Continue() function decides on the basis of the configuration data to stop the process. Each execution of the content of this loop is called a fuzz iteration.

At line 8, the Schedule() function takes as parameters information about the current fuzz iteration and the global configuration $\mathbb{C}$. It outputs the configuration $conf$ to be applied for the current fuzz iteration.

The InputGen() function at line 9 takes as a parameters the current fuzz iteration configuration $conf$ and builds tcs, the concrete test-case to be applied as inputs to the P.U.T.

At line 11, the InputEval() function takes as parameters $conf$, tcs and a bug oracle O_{bug} and applies the test-case to the P.U.T. On the basis of O_{bug}, the function decides if the execution went well or not and eventually returns found bug $\mathbb{B}'$. It is also in charge of gathering information about the current execution : $execinfos$, to be able to optimize in real-time to fuzzing process.

Finally, ConfUpdate() function at line 12 uses the gathered feedback information about the current fuzz iteration to optimize the fuzzing process, especially the input space browsing, by updating the global configuration $\mathbb{C}$.

3.4 Taxonomy

There is plenty of available fuzzing tools often designed for specific situations and, in order to ease the choice of a particular tool, they can be put into different categories. This Section presents several common ways to categorize fuzzing tools.

The attacker model : Many available fuzzing tools are designed for a specific attacker model, could be white-box, black-box or grey-box, conditioning the amount of information the attacker has access to.

In the black-box model, the fuzzer is only aware of the current input to the P.U.T and the current output and no more. To take a decision, the bug oracle has only access to the target answer, same for the input space optimisation algorithms.

The white-box attacker model is the opposite one. The fuzzer has access to information about the P.U.T internals and design to elaborate a better space search strategy. It could for example make use of static analysis techniques and dynamic symbolic execution to identify critical and highly promising regions before P.U.T execution. In this model the P.U.T is instrumented, for example with a customized compilation process adding extra instructions and the fuzzer is able to gather a lot of informations about the target execution, especially with the help of the operating system and process monitoring tools.

The grey-box attacker model falls in-between the two previously discussed ones. The fuzzer does not have access to P.U.T internals, however the code could be instrumented and some sparse side-effects could be measured and used as metrics to be optimized thanks to the operating system and process monitoring.

The input space description : There is two main concurrent ways to describe the fuzz input space to the fuzzer. First approach is the model-based one where the space is defined by a formal model, often in the form of a language grammar. All the test-cases are generated according to this model. The other way is the mutation-based approach where the space is rather described by a single example. The fuzzer is then in charge of deriving this example sample by applying mutations to it, for example bitwise operations, permutations etc...

The state awareness : Many fuzzer implementations do not take in consideration the current state of the P.U.T internal state-machine. A state-aware fuzzer would rather try to exploit sequences of test-cases to bring the targeted state-machine in a particular state. This enables exploring in depth P.U.T state-machine to reach bugs which are not in the shallow states. It improves as well the reproducibility of the found bugs because it keeps trace of the sequence of successive states which led to it.

Chapter 4

Top-level engineering choices and strategy

4.1 Fuzzing at the T=1 abstraction level

The ISO7816-3 protocol stack illustrated in Figure 2.7 is a practical top-level view for discussing the strategic decisions which led to the fuzzer conception work. This schematic puts in evidence the three different abstraction layers in use by the protocol : the physical layer, the link layer and the application layer. The driver software aimed to be tested in this project is in charge of running the link layer, it receives application data from applets running on the card in the form of an *APDU* data structure, then it slices it up and passes the correct sequences of bytes to be sent to the physical layer. For this reason, in order to make sure to target specifically the driver implementation and not the application running on top of it, the decision is taken to fuzz at the link layer abstraction layer. This means that the test-cases forged and handled by the fuzzer are T=0 or T=1 data structures and are not *APDU* data structures. First, it brings the problem discussed in Section 4.2 because the smartcard readers available on the market are providing only the ability to handle *APDU*s with no control over the underlying layers. Secondly, because there is two distinct link layer protocols (T=0 and T=1, see Sections 2.6 and 2.7) the developed fuzzing tool has to be specific to each of those protocols. Previously realized projects, especially the implementation of the reader version of the T=1 protocol showed several interesting lacks in the protocol specification discussed in section 4.3. This led to prioritize first the development of the T=1 version.

4.2 Interfacing the targeted card with the fuzzer

A fuzzing software is very demanding in computing resources, it needs to perform a lot of computations and data manipulations to derive the models, find optimal test-cases, check the answers and log the results. This is why it has to be run on a regular computer rather than on an embedded target. Moreover, the very large majority of existing tools are designed to run on a computer. The problem comes with the observation that computers do not natively have

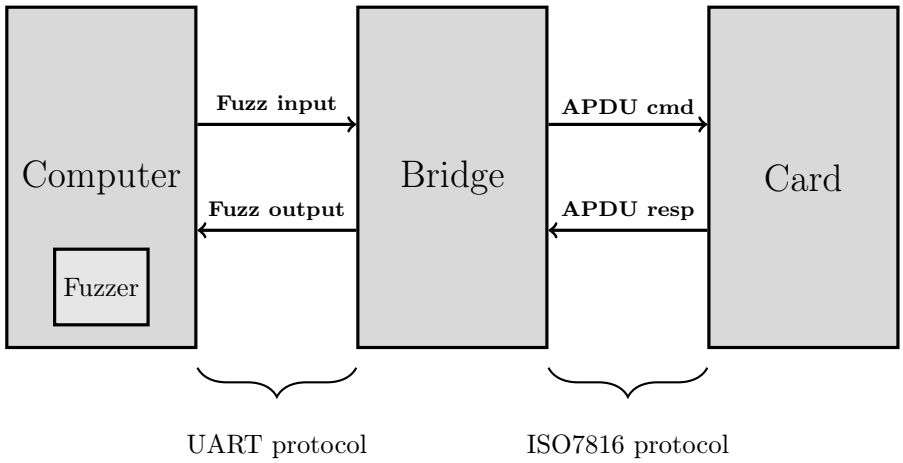


Figure 4.1: Schematic illustration of the overall fuzzing setup. The fuzzer on the left side running on a computer is interfaced with the smartcard by the bridge device depicted in the middle.

an ISO7816 interface to deal with a smartcard, an intermediate interface like depicted in Figure 4.1 has to be added between the computer and the card. It could have been a regular *USB* reader from the market however those are not satisfying because they expose solely an interface at the *APDU* abstraction level and this is incompatible with the needs expressed in Section 4.1. There is a need for a reader capable of letting full control over T=0 and T=1 protocol flow. This could help to enforce some particular cases in the T=1 protocol and to push its state-machine into some previously identified and interesting states discussed in the next Section 4.3. As a consequence, it was decided to conceive a custom reader interface explained in Chapter 5 to match this need. As suggested in Figure 4.1, its functional role is to receive test-cases from the fuzzer running on the computer and then to retransmit them back to the card following the ISO7816 specification.

4.3 Identifying interesting areas to be fuzzed

Work previously done during the two first years of apprenticeship has led to a very precise knowledge and understanding of the T=1 protocol specification. This experience is useful to identify parts of the specified state-machine which are prone to software development errors and also to identify un-precise areas of the specifications which can lead to different interpretations depending on the developer. Several of these points are discussed in this section and are used to prioritize the test-cases with the most promising results.

The primarily targeted dysfunction in the card driver is the buffer overflow situation which generates big security concerns. It is a famous and frequently exploited bug where the attacker tries to mislead the program to write more data than it is expecting at a given place in memory. This enables the opponent to (re)write program memory in locations where it is not supposed to have access. The attacker can then change return values of functions or change informations

influencing the program execution flow. Typically this is achieved by inputting more bytes than it is currently expected, when the program writes these bytes into its own memory locations, it exceeds the initially allocated space and the bytes are being written in memory locations which were not initially allocated for it.

Applied to the smartcard and T=1 situation, given a naively implemented driver without any countermeasure, this kind of attacks could be, for example, performed by sending an information block (I-Block, see Section 2.7) with more data bytes than indicated in the LEN field. The driver would prepare a space in memory of the size of LEN bytes and store the LEN first received bytes at the expected location, then the received data bytes having indexes LEN+1, LEN+2 etc ... would be written outside the initially planned location, thus overwriting other, potentially critical, memory locations.

For these reasons it is planned to focus fuzzing efforts on the control of data field sizes. The T=1 I-Block LEN field are one way to influence this parameter, but it is also possible to mention the IFS S-Blocks used to negotiate maximum data field size (see Section 2.7).

Another point of interest is about the S-Blocks behaviour. These special blocks can be sent either by the reader or the card and are used to negotiate communication parameters or to resynchronize or abort a transaction (see Section 2.7). The resynchronization request is particularly interesting because previous development work on the T=1 protocol has put in evidence that its effect on the protocol state-machine is not perfectly well defined and specified. It is not very clear which settings have to be reset which should not be. The state where the state-machine has to come back after such a block is free to interpretation by the developer. This might lead to potential exploitable bugs for this project.

Chapter 5

The bridge connector between the fuzzer and the smartcard

For the technical and strategic reasons given in Section 4 there is a need for an interface device connecting the card to the fuzzer running on a computer. In the context of this project, this device is called "bridge" and is conceptually placed in between the computer and the card as depicted in Figure 4.1. It has on its left an interface capable of dealing with a computer (discussed in Section 5.2) and on its right another interface able to communicate with the smartcard (discussed in Section 5.1). The purpose of this section is to go through the characteristics of the bridge device, explain the design choices and the implementation details.

5.1 The smartcard communication interface

The smartcard communication interface is a conceptual component of the bridge device in charge of enabling information exchange with the card. It is currently a full stack implementation of the ISO7816-3[12] reader protocol (including *APDU*s, T=0, T=1 and physical layer, respectively detailed in Sections 2.5, 2.6, 2.7, 2.8). On Figure 4.1, it is located in the entity labelled "Bridge" and deals with the right side of the schematic. It is also represented on Figure 5.7, labelled as "Reader". The implementation of this piece of software was part of this project and was preliminary coded and tested during the two first years of apprenticeship. Those kind of devices and implementations were already existing on the market at the time, however, for the reasons exposed in Section 4.2 it was chosen to re-implement it all from scratch. Especially it enables, for example, to deal directly with the link layer protocols when fuzzing (where the readers from the market are only exposing an *APDU* interface). Functionally, it is a piece of software which takes as an input a sequence of characters to be sent to the card, sends them, gets back the answer for the cards and finally serves back to answer as an output.

Using this custom reader implementation gives the possibility to have more sophisticated interactions with the card and also to enforce some specific behaviours at the low layers of the protocol stack which there is no control on from the applicative layer. This makes the fuzzing process more complete. For example, it is possible to directly interact with the character level (explained in

```
rv = READER_HAL_SendChar(pSettings, READER_HAL_PROTOCOL_T1, byte, TIMEOUT);
if(rv != READER_OK) return BRIDGE2_ERR;

rv = READER_HAL_RcvChar(pSettings, READER_HAL_PROTOCOL_T1, &byte, TIMEOUT);
if((rv != READER_OK) && (rv != READER_TIMEOUT)) return BRIDGE2_ERR;
```

Figure 5.1: Usage of the custom reader library *API* to deal with the physical layer, transmitting and receiving characters.

Section 2.8) by using the *API* pictured in Figure 5.1.

5.2 The computer communication interface

The computer communication interface is a bridge component enabling communication between the fuzzer running on the computer and the bridge. Functionally it is a piece of software inside the bridge which takes as an input informations from the fuzzer, typically a test case to be applied to the card or configuration informations, then forwards them to the bridge state machine. It is also in charge of sending back to the fuzzer all the necessary feedbacks.

The choice of the communication technology across computer and bridge was tough because various technologies can perform the same task. Mainly the possibilities across *USB*, Ethernet, *TCP/IP* and serial were discussed. The *USB* technology provides great throughput and reliability however such high performances are not needed in this case and the software development overhead and the cost in *CPU* time would have been too important. Moreover, the chosen fuzzing tool discussed in Section 6.1 does not provide natively a *USB* interface. For those reasons *USB* was rapidly put apart. Ethernet could have been an interesting choice but it is also slightly too complex for our purpose.. The microcontroller being used in this project (see Section 5.3) has a native hardware support for Ethernet however, the fuzzer chosen in Section 6.1 supports natively only *TCP/IP* connections. So it would be necessary to implement the *TCP/IP* protocol stack in software, which is an important overhead in performances, development time and an important source of errors and bugs. Thus, *TCP/IP* and Ethernet were put apart too. The serial connection was found to be very promising by being very lightweight with full hardware support for the microcontroller side and a native implementation in the selected fuzzing tool (see Section 6.1).

5.3 The hardware running the bridge software

As specified in Sections 5.1 and 5.2, the bridge is an hardware device physically doing the link between the computer and the card and is expected to be able to run the software implementation of the ISO7816-3 protocol from one side and provide a serial interface from the other side. It is also expected to have the computing capacity to run some middleware logic in the form of a state machine in order to make the translation across protocols and eventually to execute actions depending on the situation. These expectations are quite low so

Figure 5.2: Picture of the STM32F407 Discovery board provided by STMicroelectronics taken from its user manual[17]. The main chip in the middle is the STM32F407 microcontroller, which is surrounded by all the necessary equipment to make it run while easing programming and debugging.

a basic single core microcontroller with *UART* hardware peripherals is sufficient. As long as the reader library mentioned in Section 5.1 has been developed for an STM32F407 microcontroller target, the decision was taken to keep the same target to host the whole bridge device. Given the fact that this device is not aimed to be industrialized, sold or distributed, the pricing difference across the different models and references are not criteria of choice. For the same reasons it was also decided to design the whole project on the associated prototyping board widely provided by the STMicroelectronics company called the "discovery board" under the STM32F407G-DISC1 reference.

This prototyping board, visible on Figure 5.2, comes with a very complete STM32F407[16] microcontroller and all the surrounding equipment necessary to execute programs and to ease the development. For example it provides a regulated power source, a crystal oscillator, *USB* to *UART* conversion chips to ease interfacing with other devices, great connectors to ease pluging and programming and debugging hardware.

The STM32F407 is a 32 bits architecture microcontroller chip manufactured by the STMicroelectronics company. It integrates a powerful *ARM* Cortex-M4 core and very numerous peripherals all interconnected with an *AHB* internal bus. Especially, it comes with very configurable *UART* peripherals that are going to be discussed later in this report. Also, this internal *AHB* bus is synchronized at 128MHz in this project setup, it enables to achieve a great processing speed compared to the protocols involved in computer and card communication

interfaces. Indeed, the magnitude order for the transmission of a character on both sides is one millisecond. This gives the comfort of coding the middleware logic without caring too much about the real time constraints.

Another important criteria of choice for this device were the characteristics of its integrated *UART* peripherals. The aim of this logic circuitry is to handle a whole *UART* character transmission without involving the *CPU* in order to save *CPU* time. As explained in Section 2.8, the physical layer of the ISO7816-3 protocol is similar to *UART* process to transmit and receive characters. The *UART* peripherals embedded in this microcontroller are sophisticated enough and flexible enough to get a behaviour matching the ISO7816-3 standard. Especially, it supports half-duplex communication, precise tuning of the number of stop-bits, introduction of guard-time in-between characters and error detection. A fractional baudrate generator enables easy and precise clock source generation at destination of the card and the configuration registers enable to produce frames with the right structure (start bit, 8 bits payload and parity bit). This way, it saves time on the development of the physical layer and unloads the *CPU* to save performances for dealing with the higher layers.

5.4 The first, naive, strategy

Two different approaches have been used to conceive bridges state machine. This Section describes the approach which was used as a first shot, it advantages and the drawbacks that led to the second approach detailed in Section 5.5.

The first approach was focused on keeping it simple and getting some exploitable results as fast as possible. It consisted in a simple byte repeater between the two interfaces previously described in Sections 5.1 and 5.2. When the bridge was receiving a character from the computer on the serial interface, it repeated it immediately back on the ISO7816 interface and vice versa.

This has several advantages, first the implementation of the state machine is extremely simple, thus avoiding bugs and saving development time. The second point is that, as long as the chosen fuzzer natively supports serial interface (see Section 6.1) there is no need to work on a plugin interface in the fuzzer side, it is completely transparent.

This method was working well in a matter of few days, however it suffered from several serious lacks. Some very specific cases were problematic, for example if the card starts to answer before the fuzzer finished to send its test case to the card, there will be an access conflict over the ISO7816 half-duplex transmission line. This could easily happen in a fuzzing situation were the length field (see Section 2.7) is typically going to be altered to a smaller value to trigger a buffer overflow. Moreover, it would be very practical if the fuzzer could request the reader to perform some control actions like resetting the card or changing communication parameters and this is not possible with this design. The reset operation is particularly important because the fuzzer chosen in Section 6.1 is state-aware and studying which sequence of actions is leading to a bug, so it is important to be able to replace the card state machine in a very well known state after each test case. Being able to change physical layer communication parameters also offers great further work perspectives by hardening the fuzzing process by playing with the physical layer. These observations led to the method presented in Section 5.5.

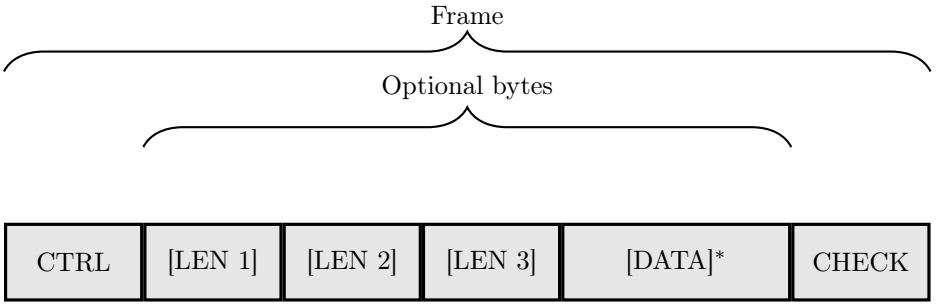


Figure 5.3: Illustration of the block structure of the communication protocol implemented between the computer and the bridge.

5.5 Second approach, the implemented protocol

The solution considered to solve the issues mentioned in Section 5.4 is to implement a real application layer communication protocol over the serial link to handle properly the data interchange and control flow information. The data to send and actions to execute are encapsulated in block structures as depicted in Figure 5.3. These block structures can be represented as frames composed of the following bytes sequences :

First comes the mandatory CTRL byte, it is used to encode the type of the block being currently sent. It can be one of the following values :

- DATA_BLOCK, its purpose is to carry a data payload. Typically it carries the data to be sent by the reader or the response from the card.
- ACK_BLOCK is used to acknowledge a previously received block.
- NACK_BLOCK is used to indicate an error at block reception.
- COLD_RST_BLOCK is a control block used to ask the reader device to perform a cold reset operation on the card.

Then, in case of a data block, the three next bytes, LEN1, LEN2 and LEN3 are big-endian encoding the length (in number of bytes) of the subsequent data field. They are then followed right after by the optional data bytes. Finally, the frame ends with a mandatory CHECK byte carrying an *LRC* error detection code.

The protocol operation is then quite simple, each block sent by a device has to be acknowledged by the other device with the dedicated block. The acknowledgement block does not need to be acknowledged. Figure 5.4 describes how the fuzzer and the bridge are interacting over time in order to process a fuzz iteration.

First, the computer sends to the bridge a cold reset block to tell the reader to execute the cold reset procedure on the card. After the reception of this information, the bridge processes it and acknowledges the correct reception of this block by sending back to the computer an *ACK* block. After *ACK* reception, the computer transmits a data block containing fuzzers payload to be sent by the reader. If it is correctly received, the bridge sends back an *ACK* block. The bridge now internally retrieves the data from the block and pass

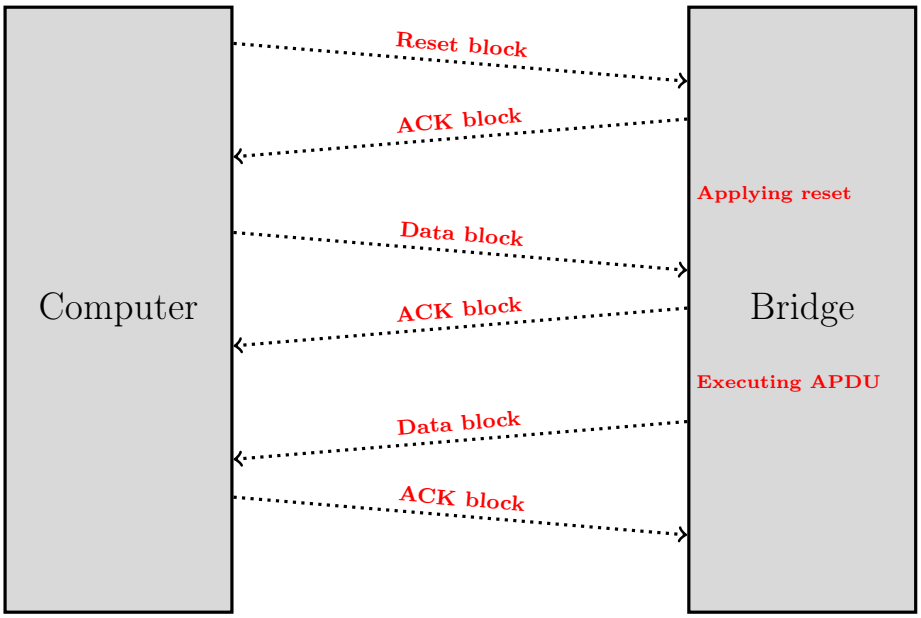


Figure 5.4: Illustration of the block structure of the communication protocol implemented between the computer and the bridge.

them to the reader interface for them to be transmitted to card. The computer is waiting until the reader gets back the answer from the card and initiates a new block transmission to send the response back to the computer. When done, the computer acknowledges it back.

5.6 Protocol state machine and its specificities

The piece of software running the computer-bridge communication protocol detailed in Section 5.5 was meant to be full-asynchronous, full-duplex and independent form the hardware. The asynchronous property gives the ability to think about keeping exchanging informations and commands with the computer while the bridge is processing a request to the card at the same time. For example, in a further improvement, it could enable the fuzzer to request a clock glitch (or other) while the bridge is interacting with the card. The full-duplex property enables the bridge to send data to the computer at the same time it is receiving. In a further improvement of the bridge it could help to push up the performances by saving communication time. This driver is also designed to be completely independent from the hardware, it eases the testing and portability. The aim of this Section is to go through the details of the state machine driving this protocol.

Because of the full-duplex property, there are two separate state-machines running at the same time, one is for block transmission to the computer, the other one is for block reception, respectively depicted as "Transmission SM" and "Reception SM" in Figure 5.7. There is a double-sided arrow across these two because even if they are independent and concurrent processes they need

to interact together, indeed, each block transmission is followed by an *ACK* reception and vice-versa. In this section we are going to focus on these two state-machines, the interprocess communication and the critical resource sharing.

Both the transmission and reception state-machines have a very similar structure which can be represented by Figure 5.8. In this section, unless specified otherwise, the considered state machine is the reception one, all the explanations are transposable to the transmission state-machine.

To achieve the concurrency and the asynchronous property the state-machines are interruption driven. Their state is re-evaluated at each TX empty (when *UART* peripheral is ready to accept a new byte to be sent) and RX not empty (when us *UART* peripheral has received a byte) interruptions. The state-machine library exposes a set of public functions, the most important ones are :

```
1 SM_Status SM_EvolveStateOnByteReception(SM_Handle *pHandle, uint8_t rcvdByte);
2 SM_Status SM_ReceiveBlock(SM_Handle *pHandle, BUFF_Buffer *pBuffer);
3 SM_Status SM_BlockRecievedCallback(SM_Handle *pHandle);
```

Figure 5.5: A selection of the most important public functions of the reception state-machine *API*.

The first one, `SM_EvolveStateOnByteReception()`, is probably the most important one. It is in charge of triggering the re-evaluation of the state-machine current state. This function is typically designed to be put by the developer into the RX not empty (RXNE) interrupt routine. It takes as arguments a pointer on the current communication context structure and the value of the byte that has just been received. The source code of this function is listed in Figure 5.6, it helps to understand how the code is structured. Most of its code is aimed to serialize the access to the communication context protected by mutexes. The three most important instructions are at lines 16, 19 and 22 :

- `SM_ComputeNextRcvState()` computes the next state to go using the current context and the received byte.
- `SM_MoveToNextRcvState()` moves the state-machine current state to the one previously computed.
- `SM_ApplyRcvState()` applies the actions corresponding to the current state.

The user of the library is free to use the `SM_EvolveStateOnByteReception()` function where he wants in its code, it is his own responsibility to integrate it to the interrupt routine. The state machine library is not directly provided with the interrupt routines to keep independence from the hardware.

The second one, `SM_ReceiveBlock()` function is used to start the reception process of a new block. It basically puts the state-machine to its initial state and checks if sometimes a reception process is not already ongoing.

The last listed function, `SM_BlockRecievedCallback()`, is a callback function defined with a weak attribute which is aimed to be redefined by the user of

```
1  SM_Status SM_EvolveStateOnByteReception(SM_Handle *pHandle, uint8_t rcvdByte){
2    SM_Status rv;
3    SEM_Status mutexRv;
4    SM_RcvState nextState;
5
6    if((pHandle->rcvHandle.flagRcptOngoing) == 0){
7      return SM_ERR;
8    }
9
10   /* Check if SM context is already accessed by another interrupt routine */
11   mutexRv = SEM_TryLock(&(pHandle->rcvHandle.contextAccessMutex));
12   if((mutexRv != SEM_LOCKED) && (mutexRv != SEM_UNLOCKED)) return SM_ERR;
13
14   if(mutexRv == SEM_UNLOCKED){
15     /* If everything is okay we process normally the state ... */
16     rv = SM_ComputeNextRcvState(pHandle, rcvdByte, &nextState);
17     if(rv != SM_OK) return SM_ERR;
18
19     rv = SM_MoveToNextRcvState(pHandle, nextState);
20     if(rv != SM_OK) return SM_ERR;
21
22     rv = SM_ApplyRcvState(pHandle, rcvdByte);
23     if(rv != SM_OK) return SM_ERR;
24
25     mutexRv = SEM_Release(&(pHandle->rcvHandle.contextAccessMutex));
26     if(mutexRv != SEM_OK) return SM_ERR;
27   }
28   else{
29     /* If reception context is already locked by another interrupt routine */
30     return SM_BUSY;
31   }
32   return SM_OK;
33 }
```

Figure 5.6: C source code of the public function on charge of re-evaluating the state-machine state at each interruption. The important operations are located at lines 16, 19 and 22, the rest of the code is in charge of context access serialization.

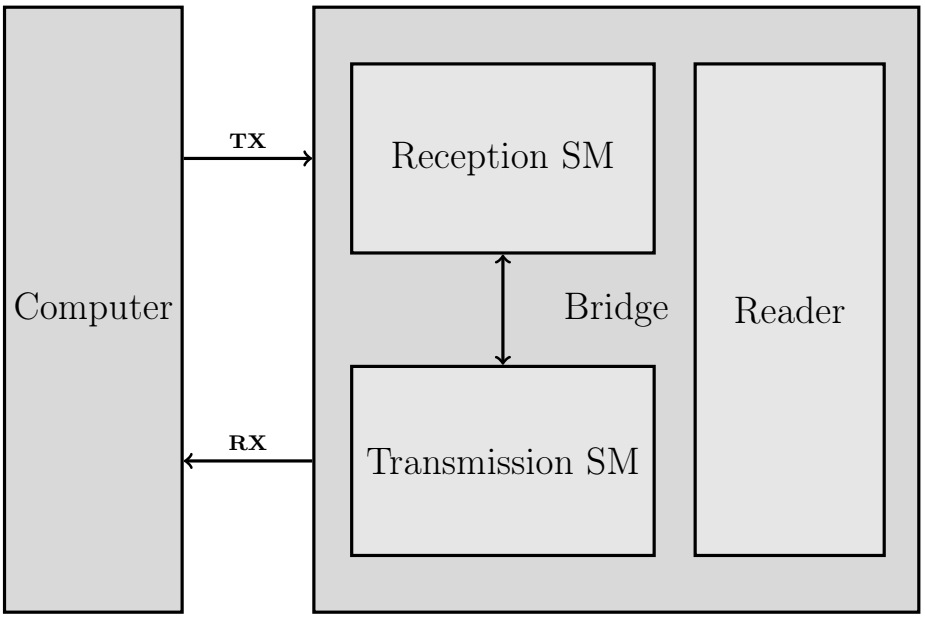


Figure 5.7: Illustration of the block structure of the communication protocol implemented between the computer and the bridge.

the state-machine library. This callback is then a custom function automatically called when the block reception process is over.

The first steps of the reception state machine depicted in Figure 5.8 are quite straigth-forward. When the reception process is triggered by a call to the `SM_ReceiveBlock()` function, the state-machine is placed into its initial state, CTRL, and the RX interruptions are enabled. In this state, the process is waiting for the reception of the first byte of the frame of the custom protocol described in Section 5.5, it is the control byte. If it encodes a data block, then the state-machine moves to the branch on the right side, it proceeds to the reception of bytes LEN1, LEN2, LEN3 and all the data bytes. If it is not, the state-machine directly moves to the reception of the checksum byte (CHECK).

The following part is then more subtle because it handles the interaction with the transmission state-machine. When the CHECK byte is received and when the block content has been verified, the reception state-machine has to ask the transmission state machine to send an acknowledgement block to the computer. It does it by setting a flag in the transmission state-machine communication context. This reception state-machine is also capable of providing an analogue service to the transmission state-machine (receiving ACK after block transmission). The main challenge of this state-machine implementation rises with the full-duplex property because both a block transmission and a block reception can occur at the same time, thus both an ACK transmission and reception can be asked at the same time. However, during a block reception (or transmission), the respective communication context are locked by a mutex structure, thus making the start a new process impossible until the previous one has finished, inducing a very problematic deadlock.

The reader has to keep in mind that the following part of the state-machine

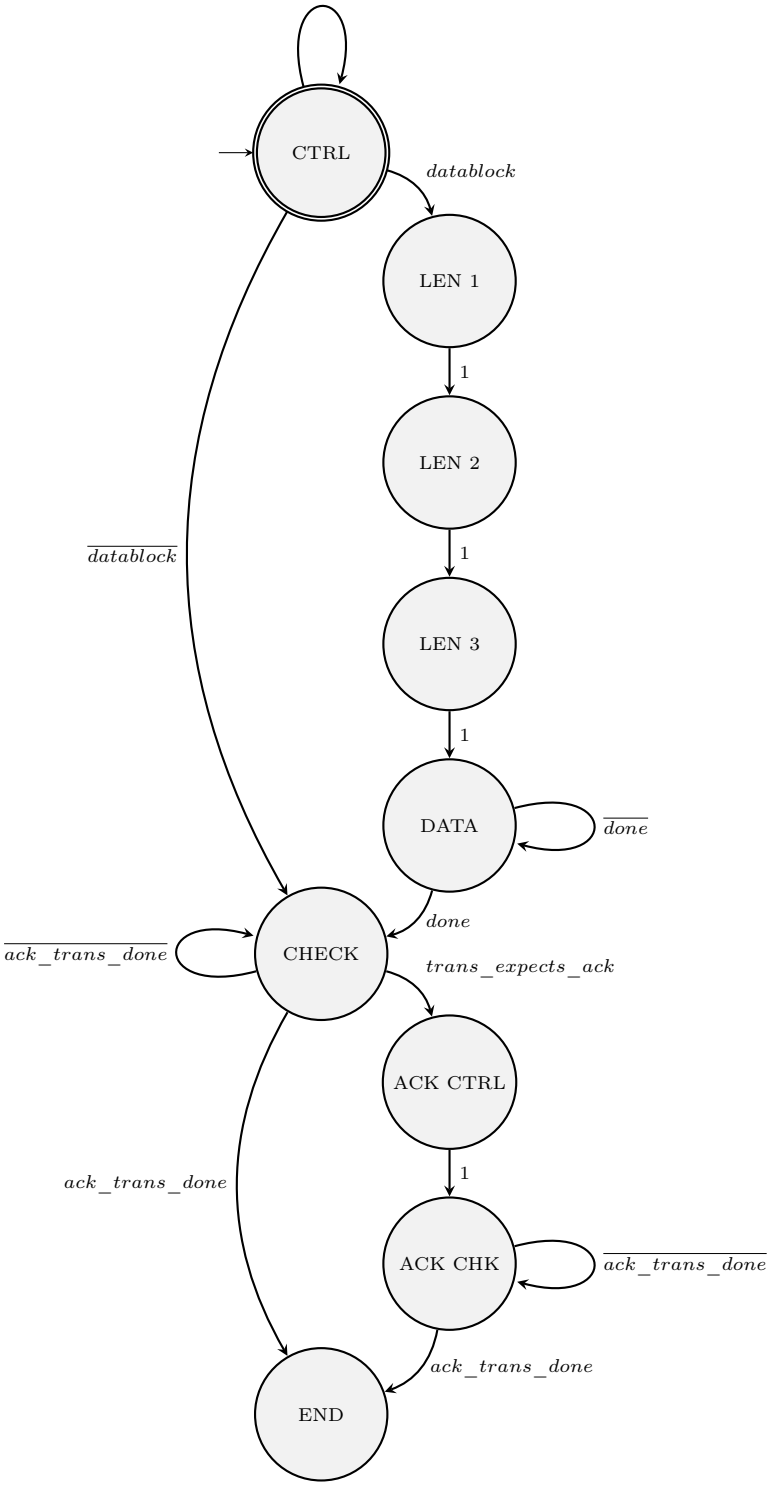


Figure 5.8: Illustration of the bridge reception state-machine.

has been designed to bring a solution to avoid this deadlock situation. The main observation is that, when being in the CHECK state waiting for the transmission of its own *ACK* block, the reception state machine is not using the reception hardware any more (which the transmission state machine is currently waiting for). The idea is to integrate the *ACK* reception (for the transmission state-machine) in the reception state-machine when the hardware resource becomes free. Thus avoiding the transmission state-machine to start the new block reception process in order to receive its own *ACK*.

When being in the CHECK state, if there is no *ACK* reception service to provide to the other state machine (normal situation, left branch of the graph), the state-machine tries to initiate a new block transmission process to send its own *ACK* block. If it is busy (transmission process already ongoing), then it notifies it to the transmission state-machine with a flag that an *ACK* transmission is expected. The state-machine stays in the CHECK state while this *ACK* has not been transmitted. When the transmission state-machine finally processes the *ACK*, it triggers a re-evaluation of the reception state-machine state to move it to the END state.

When being in the CHECK state, if there is an *ACK* reception service to provide to the other state machine (right branch of the graph), the reception state-machine initiates its own *ACK* transmission process as aforementioned, then it takes advantage of this waiting time to handle *ACK* reception for the transmission state machine. It receives the control block of the expected *ACK* block and its CHECK block. Then, it continues waiting for the transmission of its own *ACK* block before moving the the END state.

5.7 Development framework and toolchain

For the reasons exposed in Section 5.3, the STM32F407 microcontroller based on a *ARM* Cortex-m4 core was chosen for this project. The devices from *ARM* and STMicroelectronics come with a very wide range of tools, compilators, *IDE* and libraries issued by the integrators, the founder, specialized companies and the open-source community. For example, it is possible to cite the Eclipse[1] *IDE* support for STM32 and the Keil[2] development environment provided by *ARM* but as well the STM32Cube[3] device pre-configuration tool provided by STMicroelectronics. In order to keep maximum control and understandability over the compilation, flashing and debugging processes, the choice was made not to use any *IDE*, but rather to make each step separately by hand.

Several languages were discussed to carry the project like C, Rust, Ada and C++. Finally the C language using the *ANSI*/ISO C89 standard and no dynamic memory allocation was chosen for several reasons. First, it ensures maximum compatibility with those embedded targets and with the available libraries discussed further in this Section. Then, it was saving the time of learning a new language and removing the dynamic memory allocation helps partially bearing memory safety problems specific to C language. It also aims to keep code simple, with a clear vision of what is going on during the compilation

[1] https://www.st.com/en/development-tools/sw4stm32.html
[2] http://www.keil.com/
[3] https://www.st.com/en/ecosystems/stm32cube.html

process. The compilation is done using the open-source `arm-none-eabi-gcc`[4] compilator initiated by the *ARM* company.

The microcontroller also comes with a whole ecosystem of libraries, from various origins, in order to avoid dealing directly with hardware, improving portability and saving development time. In this project, the choice was made to use the *CMSIS*[5] hardware abstraction library for the cortex-m core provided and open-sourced by *ARM* coupled with the STMicroelectronics STM32 *HAL* library abstracting accesses to microcontrollers hardware peripherals. Some other libraries with higher abstraction levels are also available like the Mbed[6] framework also provided by STMicroelectronics, it was judged not necessary and obfuscating the precise operations happening on the device. These two aforementioned libraries are not provided stand-alone but are rather included in the heavyweight *IDE* software mentioned above not being used in the project. For this reason, time was invested to understand and document the compilation and linking process of those libraries in order to write equivalent build instructions with a makefile.

Flashing targets memory was achieved using the STLink[7] tool provided by STMicroelectronics. The on-target tools setup is presented in Section 5.8

5.8 Code verification and testing strategy

To be able to asses the quality of the bridge implementation, the respect of the functional needs and to save on-target debugging time, the software development was framed by rigorous testing methods in an approach close to "test driven". The ultimate goal was to be able to check and validate the behaviour of the state-machine described in Section 5.5 on the development machine before flashing it on the board.

First, the Unity[8] library was incorporated to the project. It provides a powerful set of test assertions macros illustrated in Figure 5.9. This framework was chosen mainly because it is written in pure C code following the *ANSI* C standard, thus enabling compiling the tests with the same compilator options as for the real target bringing us closer to the reality when executing tests. Also it is conceived to be eventually run on the embedded target, providing for example configuration options to manage memory used by the framework. It is also a well maintained, documented and updated open-source project.

Dedicated compilation procedures have been written to build the tests to be able to run on the development machine. It brings the capacity to run the tests on the computer when developing, saving time consuming hardware setup. It is also a good way to stress the code portability by running it on another architecture.

The difficulty of embedded code testing comes when dealing with hardware dependent functions. As long as the tests are run on the development computer, some portions of code can not be compiled for this target, they need to be emulated. For example, it does not make sense to run on the computer the

[4] https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain
[5] https://developer.arm.com/tools-and-software/embedded/cmsis
[6] https://os.mbed.com/
[7] https://www.st.com/en/development-tools/st-link-v2.html
[8] http://www.throwtheswitch.org/unity

```
1  void test_BUFF_Init_shouldSetInitValues(void){
2    BUFF_Buffer buffer;
3    BUFF_Status retVal;
4
5    retVal = BUFF_Init(&buffer);
6    TEST_ASSERT_TRUE(retVal == BUFF_OK);
7
8    TEST_ASSERT_EQUAL_UINT32(0x00000000, buffer.writeIndex);
9    TEST_ASSERT_EQUAL_UINT32(0x00000000, buffer.readIndex);
10   TEST_ASSERT_EQUAL_UINT32(0x00000000, buffer.currentSize);
11 }
```

Figure 5.9: Example of a very basic unit test using the Unity library on the circular static circular buffer implemented in the project. In lines 6, 8, 9 and 10 Unity test assertions macros are checking that the initial values of the circular buffer structure are correctly set after the call to `BUFF_Init()`.

microcontroller hardware initialization code, neither the ISO7816 characters emission and reception functions.

To overcome this problem, the CMock[9] framework was added to the project. It provides a Ruby script automatically generating function stubs from header files. For each mocked function, it generates a set of methods to be called during the test routine in order to configure the behaviour of the mocked function. It is a very sophisticated and complete tool providing many ways to explicit the expected arguments for the mocked function and the values in has to return. An example of CMock usage is shown in Figure 5.10, it prepares the emulation of a sequence of calls to the `READER_HAL_SendChar()` function. It sets the expected arguments for each successive call to this function and the value to be returned (`READER_OK`). When compiling the tests, the stubs generated by CMock are linked instead of the real functions which are not compiled at all.

This framework was kept for the project because very few tools are doing this job and it is very complete. The project is open-sourced, very complete, very well documented and maintained. The stubs are generated in *ANSI* C code.

In order to gather some metrics about the quality of the testing, the process was coupled with code coverage measurement tools. The goal is to be able to measure which proportion of the written code has been run during the execution of the tests and also to identify other portions of code which have not been involved at all, suggesting incomplete testing routines to be improved. This was achieved using `gcov` and `lcov` tools provided with the *GNU* toolchain. The test binaries are compiled with the `-fprofile-arcs -ftest-coverage` options enabling code instrumentation at compilation time. The instrumented test code is then run and the code coverage information are gathered and analysed next with the `lcov` tool and finally processed by `genhtml` tool to get a visually nice and colourized coverage feedback as depicted in Figure 5.12. This testing method has the benefit of being completely automatic and very fast. After each modification in the source code it is possible in a matter of seconds to check

[9] http://www.throwtheswitch.org/cmock

```
void setExpectedCharFrame(uint8_t *expectedBytes, uint32_t size){
  uint32_t timeout = 1000;
  uint32_t i;

  for(i=0; i<size; i++){
    READER_HAL_SendChar_ExpectAndReturn(expectedBytes[i], timeout, READER_OK);
    READER_HAL_SendChar_IgnoreArg_pSettings();
    READER_HAL_SendChar_IgnoreArg_protocol();
    READER_HAL_SendChar_IgnoreArg_timeout();
  }
}
```

Figure 5.10: Example of CMock framework usage configuring the way it is going to emulate the behaviour of the function in charge of sending a character to the smartcard.

```
state_machine.c:38:test_SM_ReceiveDataBlockShouldWork:PASS
state_machine.c:39:test_SM_ReceiveEmptyDataBlockShouldWork:PASS
state_machine.c:40:test_SM_ReceiveControlBlockShouldWork:PASS
state_machine.c:41:test_SM_TwoReceiveInARow:PASS
state_machine.c:42:test_SM_SendDataBlockShouldWork:PASS
state_machine.c:43:test_SM_TwoSendInARow:PASS
state_machine.c:44:test_SM_SendEmptyDataBlockShouldWork:PASS
state_machine.c:45:test_SM_SendDataBlockWhenRcvOngoing_case01:PASS
state_machine.c:46:test_SM_SendDataBlockWhenRcvOngoing_case02:PASS
state_machine.c:47:test_SM_SendControlBlockShouldWork:PASS
state_machine.c:48:test_SM_SendDataBlock_anotherBlockInsteadAck:PASS
state_machine.c:49:test_SM_SerializationOfDataBlockSendCalls:PASS
state_machine.c:50:test_SM_SerializationOfNonAckedBlockSendCalls:PASS
state_machine.c:51:test_SM_shouldNotWaitForAckAfterBusyBlock:PASS
state_machine.c:52:test_SM_rcvMutexShouldWork:PASS
state_machine.c:53:test_SM_SendMutexShouldWork:PASS
state_machine.c:54:test_SM_ExtraParasitByteReceived:PASS
state_machine.c:55:test_SM_checkAckIsSentAfterDataRcpt:PASS
state_machine.c:56:test_SM_checkAckIsSentAfterCtrlRcpt:PASS
state_machine.c:57:test_SM_checkAckIsCorrectlyQueued:PASS
state_machine.c:58:test_SM_EvolveStateOnByteTransmission_shouldResetTxeWhenDone:PASS
state_machine.c:59:test_SM_ACK_BLOCK_ReceivedCallback_Case01:PASS
state_machine.c:60:test_SM_ACK_BLOCK_ReceivedCallback_Case02:PASS
state_machine.c:61:test_SM_ACK_BLOCK_ReceivedCallback_Case03:PASS

-----------------------
24 Tests 0 Failures 0 Ignored
OK
```

Figure 5.11: Terminal output after running state-machine functional tests on the development computer. It gives a good confidence level that it is behaving as expected.

```
149 : SM_Status SM_EvolveStateOnByteReception(SM_Handle *pHandle, uint8_t rcvdByte){
    :         SM_Status rv;
    :         SEM_Status mutexRv;
    :         SM_RcvState nextState;
    :
    :
149 :         if((pHandle->rcvHandle.flagRcptOngoing) == 0){
  0 :                 return SM_ERR;
    :         }
    :
    :
    :         /* We check if the state machine context is already accessed by another interrupt routine ...  */
149 :         mutexRv = SEM_TryLock(&(pHandle->rcvHandle.contextAccessMutex));
149 :         if((mutexRv != SEM_LOCKED) && (mutexRv != SEM_UNLOCKED)) return SM_ERR;
    :
149 :         if(mutexRv == SEM_UNLOCKED){
    :                 /* If everything is okay we process normally the state ... */
148 :                 rv = SM_ComputeNextRcvState(pHandle, rcvdByte, &nextState);
148 :                 if(rv != SM_OK) return SM_ERR;
    :
148 :                 rv = SM_MoveToNextRcvState(pHandle, nextState);
148 :                 if(rv != SM_OK) return SM_ERR;
    :
148 :                 rv = SM_ApplyRcvState(pHandle, rcvdByte);
148 :                 if(rv != SM_OK) return SM_ERR;
    :
148 :                 mutexRv = SEM_Release(&(pHandle->rcvHandle.contextAccessMutex));
148 :                 if(mutexRv != SEM_OK) return SM_ERR;
```

Figure 5.12: Screenshot of a portion of the colourized code coverage report. It clearly puts in evidence that the test procedure never went through the line 8 of the source code.

whether there is a regression for example.

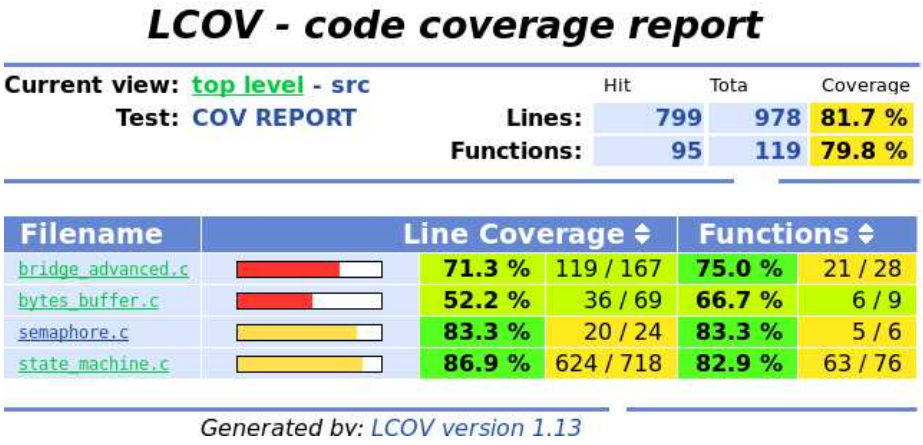
When the unit and functional tests are passed and when there is a great level of confidence that the state machine is behaving as it was conceived to, the program is cross-compiled for the real target and flashed in the microcontroller memory. The program is then run and begins the on-target testing phase. Those tests are not automatic and way more time consuming. They are performed using a logic analyser with probes connected on the ISO7816 electrical interface (see Section 2.3) and on the computer-bridge serial link TX and RX pins. It is then possible to observe the characters exchanged through the wires and check that it is compliant with the protocol detailed in Section 5.5. A tiny Python script is running on the computer to simulate the fuzzer sending a test case to the bridge through the serial link.

Figure 5.14 is a screenshot from the logic analyser, it demonstrates the correctness of the protocol execution flow. The first line corresponds to the probe on the ISO7816 clock line, it is a good indicator to see if the card reboots. The second line is the probe placed on the ISO7816 I/O line, it gives visibility on the characters exchanged between the bridge and the card. The third line is probed on the serial link between the computer and the bridge, it catches the characters being sent from computer to the bridge. The last line shows the characters being sent in the opposite direction. The following is happening :

1. The computer send to the bridge a cold reset request. The bridge acknowledges it back to the computer.

2. The reader part of the bridge performs the cold reset of the card. It is followed by the *ATR* from the card, indicating that the reset happened.

3. The computer sends to the bridge a data block containing a test case to be applied to the card. After reception the bridge sends back an *ACK* block to acknowledge the correct reception.



LCOV - code coverage report

Current view: top level - src		Hit	Tota	Coverage
Test: COV REPORT	Lines:	799	978	81.7 %
	Functions:	95	119	79.8 %

Filename	Line Coverage		Functions	
bridge_advanced.c	71.3 %	119 / 167	75.0 %	21 / 28
bytes_buffer.c	52.2 %	36 / 69	66.7 %	6 / 9
semaphore.c	83.3 %	20 / 24	83.3 %	5 / 6
state_machine.c	86.9 %	624 / 718	82.9 %	63 / 76

Generated by: LCOV version 1.13

Figure 5.13: Global code coverage report over the whole project.

Figure 5.14: Screenshot of the logic analyser output. The computer is sending a cold reset request, the card reboots, then the computer sends a test case to be applied to the card. The card answers and the response is sent back to the computer through the bridge.

4. The content of the previously received data block is sent to the card through the ISO7816 IO line. The card answers back to the bridge right after on the same half-duplex line.

5. The bridge sends back to the computer the previously received response from the card. It encapsulates it into a data block. The computer acknowledges the reception of the card response.

Testing methods aforementioned are well suited for putting in evidence the presence of bugs or non-conform behaviour. The challenge is then to precisely identify the portion of code inducing the bug to be able to fix it. This brings the need for powerful debugging tools. When the tests are run on the computer it is a relatively easy process with well known techniques and tools. In the project, this is achieved using the very famous command line *GDB* debugger. However, once flashed on the microcontroller, new issues can rise due to the hardware interaction and dependency. It becomes then quite more challenging to debug in real-time on target. The set up in this project (shown on Figure 5.15) is to use the *JTAG* core embedded in the Cortex-m *CPU*. *JTAG* is a

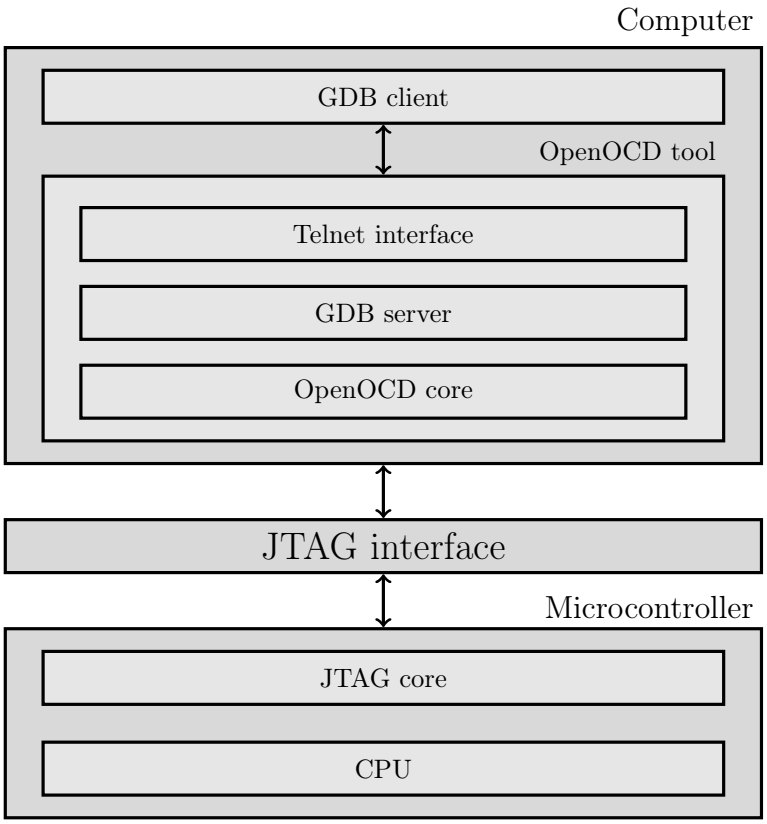


Figure 5.15: On-target debugging process. The angular stone is the OpenOCD software running on the computer. From one side it connects to the target using *JTAG* protocol, from the other side it exposes a *GDB* server interface.

debugging system and communication protocol with a tester, giving the ability to access in real time all the memory and registers and to control the execution flow. The computer used for debugging (on the top of the schematic) runs the OpenOCD[10] software which can monitor the *CPU* through the *JTAG* interface. On the other hand, OpenOCD also exposes a *GDB* server through a telnet interface. The *GDB* client used for debugging can then connect to this server and perform all the necessary operations.

[10] http://openocd.org/

Chapter 6

The fuzzing engine

6.1 Choosing the appropriate fuzzing tool

Fuzzing methods explained in Chapter 3 have quickly become a mainstream and popular technique for testing software and assessing its security. As a consequence, extremely numerous fuzzing software projects have been initiated. A good example is the GitHub platform where more than a thousand of repositories are linked to the "fuzzing" keyword. It has not always been so famous and historically, the first experimentations which could be designated as "fuzzing" nowadays were not initiated by science work, but rather by many independent people empirically and anarchically trying to improve their conventional way of testing on their own side without coordination. This situation has led to a very large and heterogeneous ecosystem of fuzzing software. Academic and research work has came later, generalizing the concepts, providing a common ground, common vocabulary and the common "fuzzing" word to group all these techniques. This situation has made the process of choosing a fuzzing tool a tremendously time-consuming task.

Because some of these tools exists for a long time and because the research in this area is currently very dynamic, the fuzzing algorithms are the result of many years of research and empirical improvements, thus making some of them very efficient. This maturity is the reason to use an existing core and not to develop a new one from scratch. Finally, when taking all the requirements together very few projects were matching them and the choice was made to use a tool named Boofuzz[1].

Boofuzz is an open-source fuzzing framework under the *GPL* v.2 license which makes it fit with the open-sourcing goal of this project. At the same time, the project GitHub repository shows a great and recent activity from its numerous maintainers and the documentation is decent in comparison to other projects. It is also written in Python language which makes it very easy to modify and contribute to. For example it opens the perspective to change mutations algorithms or to adapt it to the particularities of the target being fuzzed in this project.

Technically speaking this fuzzer is designed to be run in black-box attacker model and is specialized in protocol testing, thus making it equipped with all the

[1] https://github.com/jtpereyda/boofuzz

necessary gear to be state-aware and to explore the state-machine of the targeted device. It is also mainly model-based for the test-cases generation and brings a powerful and very concise syntax to define them directly in Python language. It is capable as well to do test-cases generation from models completed with a mutation capability. From a practical point of view it comes with a native serial interface which is saving development time when integrating with the bridge interface.

6.2 Boofuzz usage

Boofuzz first needs to be initialized before being used, it is the process of creating a Boofuzz `Session` object. It mainly involves the preliminary creation and initialisation of a `Connection` object and a `Monitor` object as shown in Figure 6.1 taken as parameters by Session.

The connection object is mandatory and encapsulates the communication methods necessary for the fuzzer to send requests to the target and to receive responses. Boofuzz framework in its original version comes with two different connection classes : `TCPSocketConnection` and `SerialConnection`, respectively enabling the fuzzer to connect via a *TCP* connection or via a *UART* link. the connection object is then wrapped into a `target` object, mainly adding logging capability.

The monitor object is not mandatory and provides monitoring and administration of the target, especially : starting a process, rebooting, getting crash synopsis, gathering logs, detecting crash etc...

```
1  procmon = ProcessMonitor(target_ip, 26002)
2  connection=TCPSocketConnection(target_ip, 21)
3
4  session = Session(
5    target=Target(
6          connection=connection,
7          monitors=[procmon],
8    ),
9    sleep_time=1,
10 )
```

Figure 6.1: Example of Boofuzz Session initialisation code taken from Boofuzz documentation examples. Session initialisation requires as parameters a Connection Object and a Monitor object.

6.3 Plugging the bridge to the fuzzer

In its current version Boofuzz is supporting only *TCP* and serial connections to the targets to be fuzzed. The main counterpart of implementing the bridge to computer communication protocol described in Section 5.5 is the necessity to implement this protocol state-machine in the Boofuzz tool as well. It came out

to be very easy because Boofuzz is an open-source, completely Python scriptable and well structured software.

The first step is to develop the Connection object (see Section 6.2) specific to the bridge target and its custom protocol. After a Booffuzz source code analysis it turns out that all the Connection classes are implementing a common interface named `ITargetConnection` specifying its functional contract as depicted in Figure 6.3. The operation contract specified by this interface is very understandable, the `SmartcardConnection` class added to the framework has to implement the following methods :

- `open()` method opens the communication interface and initializes it for the subsequent `send()` and `recv()` operations. In the case of the SmartcardConnection class to be implemented it is opening the serial *FIFO* file provided by the operating system and setting up communication parameters like baudrate.
- `close()` method closes the current connection. In the case of the SmartcardConnection class is closes the serial *FIFO* file provided by the operating system.
- `send()` method sends a request (as defined in Section 6.4) to the target. In the case of the SmartcardConnection class to be implemented it runs the transmission state-machine of the custom protocol described in Section 5.5 and sends each individual character over serial link.
- `recv()` method receives the response from the target. In the case of the SmartcardConnection class to be implemented, it runs the reception state-machine of the custom protocol described in Section 5.5.
- `info()` method displays informations about the current connection.

Boofuzz source code comes with a class named SerialLowLevel (represented on Figure 6.3) grouping and isolating all the necessary primitives to interact with the serial link, thus helping the development of the SmartcardConnection class.

The second step to create the smartcard plug-in is to develop a Monitor object (see Section 6.2) specific to the bridge target and its custom protocol. To do so, it is necessary to create a new SmartcardMonitor class completing the ProcessMonitor and NetworkMonitor classes already existing by implementing the BaseMonitor interface as shown in Figure 6.4.

In the BaseMonitor operation contract, the most important operation for this project is the `restart_target()` method. The implementation of this operation sends a cold reset block to the bridge in order to reset the card. This helps to get relevant and reproducible results by bringing back the targeted card protocol state-machine to a well known state after each fuzz iteration.

Finally Boofuzz framework initialization looks like in Figure 6.2.

```
con = SmartcardConnection(
        port='/dev/ttyUSB0',
        baudrate=9600,
        timeout=10,
        content_checker=None)

sm_monitor = SmartcardMonitor(connection=con)
target = Target(connection=con, monitors=[sm_monitor])

session = Session(
    target=target,
    restart_interval=1,
    restart_sleep_time=0.1,
    sleep_time=0.1,
    check_data_received_each_request=True,
    receive_data_after_fuzz=True)
```

Figure 6.2: Initialization code to establish the connection and the monitoring with the smartcard.

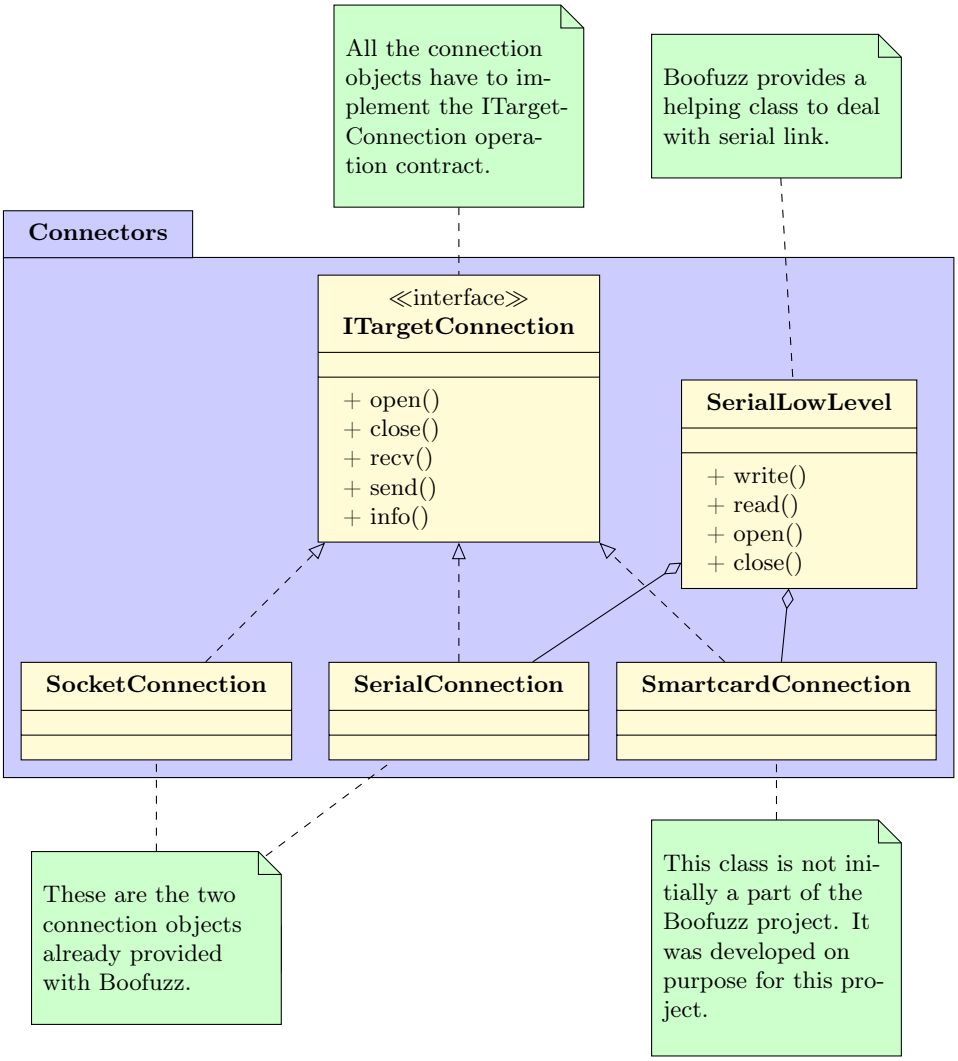


Figure 6.3: *UML* class diagram showing the associations and dependencies across the Boofuzz framework connection objects.

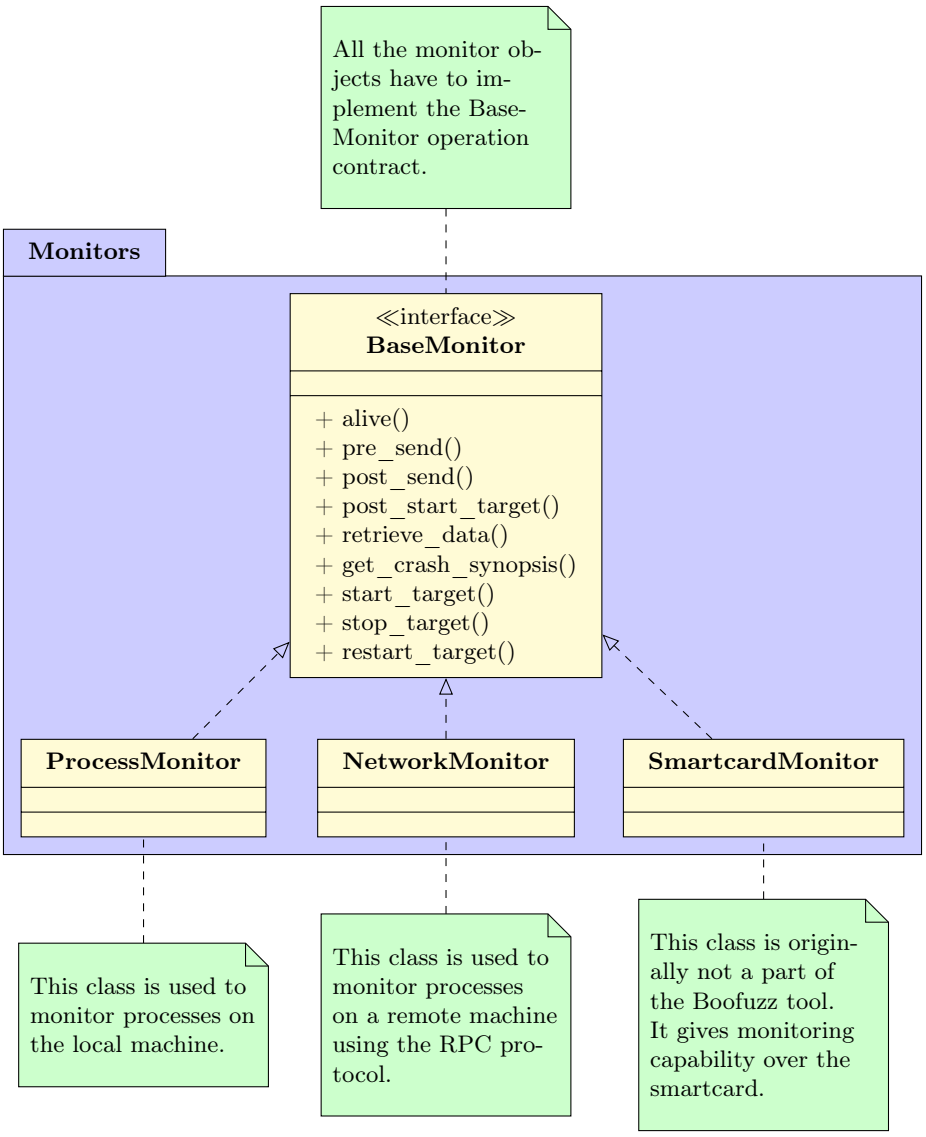


Figure 6.4: *UML* class design representing the dependencies across the monitor objects.

6.4 Communication protocol modelling

Boofuzz explores targets state-machine implementation by sending sequences of requests (also called messages) rather than single messages devoid of context. To achieve that, it maintains its own graph (initially parametrised by the user) of requests combinations to be sent shown in Figure 6.5. In this graph, each node corresponds to a request to be sent. Each request is defined by a formal model of how it is supposed to look like and how to derive from it the fuzzed messages. Each connection across nodes is a path to be tested configuring the possible request sequences to be explored. A complete sequence of requests going from the initial node to one of the graph leafs is called a test-case.

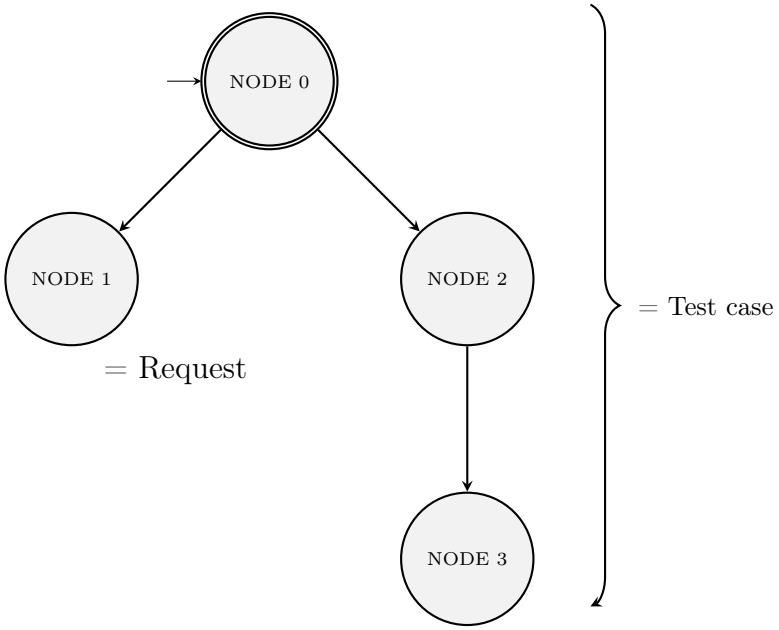


Figure 6.5: Illustration of the state-awareness concept involved in the Boofuzz tool and the associated terms.

Boofuzz is mainly a model-based fuzzer, the model definition of the requests is directly done in Python syntax and is simple and concise in comparison to other tools. A message is subdivided in chunks called blocks and each block is made of primitive elements. Various primitive types are provided with Boofuzz and new types can be created if needed. As an example it is possible to cite :

- `string`
- `delimiter`
- `static`
- `bit field`
- `binary`
- `random`
- `byte`
- `word`

Figure 6.6 presents a basic example usage of the above mentioned primitives in the case of *FTP* protocol. It models the initial request sent by the client

to the *FTP* server to authenticate itself with a given username. The first line sets the unique name to identify this request model in the Boofuzz framework, it is set to be "user". Then, the model is defined to be two strings "USER" and "anonymous" in a row separated by a delimiter and ended by a carriage return. The type of the primitive has a influence over the kind of mutations to be applied.

```
1 s_initialize("user")
2 s_string("USER")
3 s_delim(" ")
4 s_string("anonymous")
5 s_static("\r\n")
```

Figure 6.6: Example of a basic model definition of message taken from Boofuzz documentation. It defines a message named "user" composed of four primitive elements : a string, a delimiter, another string and a static (non-fuzzable) element.

These sequences of primitive elements can then be grouped as blocks to ease performing operations on them. These operations are as various as computing a checksum of a block for adding it to the fuzzed frame, computing a size block and repeating a block. This concept is illustrated in Figure 6.7 giving an example model of a T=1 protocol (see Section 2.7) I-Block frame carrying an *APDU* command (see Section 2.5). There are three main blocks named "prologue-field", "information-field" and "epilogue-field". There is also a "body" block encapsulating the prologue field and the information field which are involved in the frame checksum computation. This enables, on line 27 to add a byte to the fuzzed frame containing the *LRC* checksum as described in ISO7816. The "bytes" block from line 18 contains all the *APDU* data field bytes, it gives the possibility to compute their length to put it into the T=1 LEN field (see Section 2.7).

```
s_initialize(name="req0")

s_block_start("body")

s_block_start("prologue-field")
s_bit_field(name="NAD", value=0b00000000, endian='<', width=8, output_format='binary',
                                                                    fuzzable=False)
s_bit_field(name="PCB", value=0b00000000, endian='<', width=8, output_format='binary',
                                                                    fuzzable=False)
s_size(name='LEN', block_name='information-field', offset=0, length=1,
       output_format='binary', signed=False, inclusive=False, fuzzable=False)
s_block_end("prologue-field")

s_block_start("information-field")
s_byte(name="CLA", value=0x00, output_format='binary', signed=False, fuzzable=False)
s_byte(name="INS", value=0xA4, output_format='binary', signed=False, fuzzable=False)
s_byte(name="P1", value=0x04, output_format='binary', signed=False, fuzzable=False)
s_byte(name="P2", value=0x00, output_format='binary', signed=False, fuzzable=False)

s_block_start("bytes")
s_byte(name="byte", value=0xCA, output_format='binary', signed=False, fuzzable=True)
s_block_end("bytes")
s_repeat(name='data', block_name="bytes", min_reps=0, max_reps=10, step=1, fuzzable=True)
s_block_end("information-field")

s_block_end("body")

s_block_start("epilogue-field")
s_checksum(name='check', block_name='body', algorithm=compute_lrc_check, length=1,
                                                                    fuzzable=False)
s_block_end("epilogue-field")
```

Figure 6.7: An example model definition of the ISO7816 T=1 information block carrying an *APDU* using the Boofuzz tool.

Finally, all the requests models are connected together to create a graph as shown in Figure 6.5. It is done like in Figure 6.8 with the `session.connect()` method.

```
session.connect(s_get("user", "ROOT"))
session.connect(s_get("user"), s_get("pass"))
session.connect(s_get("pass"), s_get("stor"))
session.connect(s_get("pass"), s_get("retr"))
```

Figure 6.8: An example of requests graph building taken from the Boofuzz documentation (applied to the *FTP* protocol). "user", "pass", "stor" and "retr" are requests identifiers.

6.5 Oracle definition

Oracle is a piece of software being part of the fuzzer. It is in charge of determining whether the reaction of the card to a request or a test-case is nominal or

symptomatic of a malfunction or an error. As long as the chosen attacker model is the black-box one, the oracle is only aware of the input provided to the target and the answer received back from it and can not access more informations. For example, like it is very common with grey-box fuzzing techniques there is no access to operating system to see if the tested driver has crashed, if some pointers have abnormally moved etc...

This module is still not developed yet, however several solutions are currently being considered. The first idea is to emulate a fully controlled card version of the state-machine on the computer side. At each fuzz iteration, the generated input would be injected to the target card and at the same time it would be provided to the concurrent emulated state-machine. The oracle would then compare the two outputs to decide if there is a deviation from a reference implementation. Two solutions are under discussion to get this card state-machine implementation. A solution could be to reuse the reader implementation of the ISO7816-3 protocol and doing some changes to make it behave like a card. Another solution would be to obtain the card driver implementation from an industrial, virtualise it and make it run on computer. The later is politically and legally more complicated and could has consequences on the project open-source nature.

The other way is to apply a technique called "differential fuzzing". It consists in fuzzing two cards at the same time and comparing the answers. It would be performed with cards from different constructors having a different driver implementation. This solution is way less time consuming than the previously presented ones, however it could produce less reliable results, depending on the trust in the reference card.

Chapter 7

Results

As a result of all the previously discussed work, this project ends with a framework of tools giving the ability to generate relevant test-cases, apply them to a card and get back the response. Figure 7.1 demonstrates it by showing the output of the terminal when running Boofuzz tool with the previously described T=1 models.

At lines 1 and 24 it is visible that the fuzzer resets the card with the smartcard monitor detailed in Section 6.3. At line 4 it opens the connection with the card and sends the fuzzed request at line 11. Finally at line 14, it receives back the answer.

```
1  [19:27:56]     Info: Restarting target process using SmartcardMonitor
2  [19:27:56] Test Case: 2: req0.byte.2
3  [19:27:56]     Info: Type: Byte. Default value: b'\xca'. Case 2 of 123 overall.
4  [19:27:56]     Info: Opening target connection (port: /dev/ttyUSB0, baudrate: 9600)...
5  [19:27:56]     Info: Connection opened.
6  [19:27:56]   Test Step: Monitor ProcessMonitor#140340867959888.pre_send()
7  [19:27:56]   Test Step: Monitor CallbackMonitor#140340842364320[pre=[],post=[],restart=[],
8                                                       post_start_target=[]].pre_send()
9  [19:27:56]   Test Step: Fuzzing Node 'req0'
10 [19:27:56]     Info: Sending 9 bytes...
11 [19:27:56]     Transmitted 9 bytes: 00 00 05 00 a4 04 00 01 a4 b'\x00\x00\x05\x00\xa4\x04
12                                                                          \x00\x01\xa4'
13 [19:27:56]     Info: Receiving...
14 [19:27:56]     Received: 00 00 02 67 00 65 b'\x00\x00\x02g\x00e'
15 [19:27:56]   Test Step: Contact target monitors
16 [19:27:56]   Test Step: Cleaning up connections from callbacks
17 [19:27:56]       Check OK: No crash detected.
18 [19:27:56]     Info: Closing target connection...
19 [19:27:56]     Info: Connection closed.
20 [19:27:56]   Test Step: Sleep between tests.
21 [19:27:56]     Info: sleeping for 0.100000 seconds
22 [19:27:57]   Test Step: restart interval of 1 reached
23 [19:27:57]   Test Step: Restarting target
24 [19:27:57]     Info: Restarting target process using SmartcardMonitor
```

Figure 7.1: Output of the terminal when running Boofuzz on a smartcard. It shows the successful execution of one test-case.

Chapter 8

Conclusion and further work

All the previous chapters have demonstrated how it is technically feasible to apply state-of-the-art fuzzing methods to the software implementation of ISO7816 drivers in smartcards. It first puts in evidence the smartcard world specificities which renders the usage of available fuzzing tools not straightforward. It then proposed solutions to make it work, to create the link between the fuzzing tool and the card and it then demonstrated that these solutions are suitable for finding vulnerabilities.

The value brought by this project is first a proof-of-concept showing the feasibility of such a fuzzing approach and identifying the tools and solutions to achieve it. Then, it brought a powerful and ready to use framework and setup opening a great perspective over the second part of the project aiming to find and exploit vulnerabilities.

Nevertheless, the project initial goals are not all fulfilled, there is still a large amount of work to be done before being able to first identify irregularities in drivers implementations and even more efforts to be done to prove their exploitability to break security. The next steps are to settle an oracle strategy, to improve the T=1 protocol models and to establish a testing plan. Over the longer term it is also planned to model T=0 protocol, to parallelize the fuzzing process over several cards at the same time to improve performances. If the results are satisfying it is taken under consideration to publish method and results in an academic conference.

Bibliography

[1] Samia Bouzefrane and Pierre Paradinas. *Les cartes à puce*. Lavoisier, 2013.

[2] Secrétariat Général de la Défense et de la Sécurité Nationale. SGDSN in english, 2020. `http://www.sgdsn.gouv.fr/accueil/sgdsn-in-english/` [Online; accessed 10-May-2020].

[3] Présidence de la République and Ministère de la Défense. *Défense et Sécurité nationale : le Livre blanc*. La Documentation française, 2008.

[4] Présidence de la République and Commission du Livre blanc sur la Défense et la Sécurité nationale. *Livre blanc sur la défense et la sécurité nationale 2013*. La Documentation française, 2013.

[5] Agence Nationale de la Sécurité des Systèmes d'Information. La Sous-direction Expertise (SDE), 2020. `https://www.ssi.gouv.fr/agence/organisation/les-sous-directions/la-sous-direction-expertise/` [Online; accessed 1-April-2020].

[6] Agence Nationale de la Sécurité des Systèmes d'Information. L'édito du Directeur général, 2020. `https://www.ssi.gouv.fr/agence/missions/ledito-du-dg/` [Online; accessed 1-April-2020].

[7] Joeri De Ruiter and Erik Poll. Protocol state fuzzing of tls implementations. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, page 193–206, USA, 2015. USENIX Association.

[8] Keith E. Mayes and Konstantinos Markantonakis. *Smart Cards, Tokens, Security and Applications*. Springer, 2007.

[9] Cartes de proximité – Interface radio frequence et signaux de communication. Standard, International Organization for Standardization, Geneva, CH, April 2016.

[10] Information technology — Open Systems Interconnection — Basic Reference Model: The Basic Model. Standard, International Organization for Standardization, Geneva, CH, 1994.

[11] Cards with contacts – Dimensions and location of the contacts. Standard, International Organization for Standardization, Geneva, CH, October 2007.

[12] Cards with contatcs – Electrical interface and transmission protocols. Standard, International Organization for Standardization, Geneva, CH, November 2006.

[13] Auguste Kerckhoffs. La Cryptographie Millitaire. *Journal des sciences militaires*, 9:5–38, Jan 1883.

[14] Valentin Manes, HyungSeok Han, Choongwoo Han, sang cha, Manuel Egele, Edward Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, PP:1–1, 10 2019.

[15] National Institute of Standards and Technology, Computer Security Division and Cryptographic Technology Group. *Roots of Trust in Mobile Devices*, feb 2012. Regenscheid, Andrew.

[16] STM32F405/415, STM32F407/417, STM32F427/437 and STM32F429/439 advanced Arm ® -based 32-bit MCUs. Reference manual, STMicroelectronics, 2019.

[17] Discovery kit with STM32F407VG MCU. User manual, STMicroelectronics, 2017.