Evaluation of the robustness of payment terminals with the use of fuzzing

Master Thesis

Evgenios Karampatzakis

Supervisors:
dr. Emmanuele Zambon(UT)
prof dr. Sandro Etalle(UT)
Leo Kool MSc.(Brightsight)

Twente, July 2015
Abstract

Payment terminals are the devices that store and process our sensitive data and financial transaction details. The security of these devices is crucial since a possible breach would mean considerable losses and disclosure of confidential data. Researchers have reported successful attempts to take over specific devices and use them for malicious purposes. One way to estimate the robustness of a computer system to vulnerabilities like buffer overflow and integer overflow is fuzzing. Fuzzing feeds the system under test with semi valid data possible to cause unexpected behavior. In this research we carry out a study of the EMV protocol as the main protocol used for the communication between the payment terminal and the smart card. We then develop an evaluation method of the robustness of the payment terminal with the use of fuzzing and we implement a fuzzing tool as the proof of concept of this method. Finally we test the fuzzing tool in payment terminals of different vendors and we present the results. We believe that the results can be used further in order to secure the payment terminals from possible future attacks.
Acknowledgements

First and foremost I would like to thank my parents and sister, without the help of whom nothing of these would have been possible.

My gratitude also goes to my supervisors Emmanuele and Leo for their valuable help and advice.

Finally I would like to thank everyone in Brightsight for their help and for the great time I had during my graduation.
Contents

1 Introduction 1
  1.1 Related Work .............................................. 3
  1.2 Research Question ........................................ 3
  1.3 Contributions .......................................... 3
  1.4 Research Methodology .................................. 4
  1.5 Outline .................................................. 4

2 Fuzzing Methods Classification 5
  2.1 Fuzzing Definition ....................................... 5
  2.2 Fuzzing methods classification .......................... 6
    2.2.1 Intelligence of Fuzzer ............................... 6
    2.2.2 Target Awareness .................................... 7
  2.3 Classification .......................................... 8
  2.4 Discussion ............................................. 18

3 Payment Terminals 20
  3.1 Payment Terminals ....................................... 20
    3.1.1 Related definitions .................................. 20
    3.1.2 Assets of the payment terminal ...................... 20
  3.1.3 Architecture, interfaces, protocols and standards ........ 21
  3.1.4 Operating Systems (OS’s) and programming languages .... 24

4 The EMV protocol 25
  4.1 ISO/IEC 7816 ............................................. 25
  4.2 The EMV specifications ................................... 27
  4.3 APDU ....................................................... 27
    4.3.1 Command Application Protocol Data Unit (C-APDU) .... 27
    4.3.2 Response Application Protocol Data Unit (R-APDU) .... 29
  4.4 TLV ........................................................ 32
  4.5 Data Elements ............................................ 33

Evaluation of the robustness of payment terminals with the use of fuzzing
## List of Figures

1.1 Percentage of Card-Present Transactions that are EMV (reproduced from [26]) .......................... 2  
1.2 Worldwide EMV chip card deployment and adoption (reproduced from [26]) .... 2  
2.1 CFG inference illustrated (reproduced from [50]) ................................................................. 9  
2.2 Regular and mutated data generation runs using the CFG inferred in Figure 2.1 (reproduced from [50]) ................................................................. 10  
2.3 Incoming message with length field value ................................................................. 10  
2.4 Incoming message with separators ................................................................. 11  
2.5 Overview of the mini-simulation concept ................................................................. 13  
2.6 Search space for the code in listing 2. Each node contains one of the path constraints in $p = (i_0 \neq b, i_1 \neq a, i_2 \neq d, i_3 \neq !)$. The left branches contain the inputs that comply with that constraint while the right branches contain the inputs that do not. Along with input there is its corresponding cnt value. The inputs with $cnt \geq 3$ exercise line 7 in listing 2. ................................................................. 15  
2.7 Architecture of SAGE. In parentheses there are the tools used in each phase [28]. ................................................................. 16  
2.8 Classification of fuzzing techniques. ................................................................. 19  
3.1 Diagram of a payment terminal’s architecture ................................................................. 22  
4.1 Contact designations and functions according to ISO/IEC 7816-2 ............................. 25  
4.2 EMV transaction flow (reproduced from [40]) ................................................................. 28  
4.3 C-APDU format ................................................................. 29  
4.4 FCI of an application (reproduced from [40]) ................................................................. 30  
4.5 R-APDU format ................................................................. 30  
4.6 Abstract BER-TLV example (reproduced from [40]) ................................................................. 33  
5.1 The terminal first asks for the size of the requested record or file. After the smart card informs it about the correct size it asks and receives the record or file. ................................................................. 37  
5.2 The classification of the various fuzzing methods and the placement of our EMV fuzzing methodology among them. ................................................................. 38  
5.3 Message sequence which shows the interaction between the various components of the fuzzing tool ................................................................. 40  
6.1 Instrumentation of the SUT and the testing equipment ................................................................. 44
List of Tables

4.1 Summary of transmission protocols according to ISO/IEC 7816-3. T=0 and T=1 are of special interest in the EMV protocol (reproduced from [41]) .......................... 26
4.2 SELECT FILE C-APDU ................................................................. 29
4.3 READ RECORD C-APDU ................................................................. 30
4.4 VERIFY C-APDU ................................................................. 30
4.5 GET PROCESSING OPTIONS C-APDU ................................................ 31
4.6 GENERATE APPLICATION CRYPTOGRAM C-APDU ....................... 31
4.7 GET DATA C-APDU ................................................................. 31
4.8 Encoding of the Length field in the BER-TLV structure ......................... 32
4.9 Data Elements [18] ................................................................. 35
7.1 Number of normal transactions per terminal ........................................ 46
Listings

2.1 sample code for the whitebox fuzzing approach .................. 14
2.2 sample code for the dynamic code generation demonstration .......... 15
2.3 sample code for the grammar-based whitebox fuzzing demonstration ....... 17
Chapter 1

Introduction

Chip and PIN [52] or Chip Card is a payment technology which succeeded the Magnetic Stripe Card in order to protect financial transactions. This was due to the fact that the latter was vulnerable to fraud and more specifically to cloning attacks [39].

During everyday financial transactions people interact with devices which process their financial details such as the account number and Personal Identification Number (PIN). These devices, which support the Chip and PIN payment technology, are known as payment terminals. It is self-evident that the security of this end of the communication, assuming that the other end is the smart card, is crucial.

At the Black Hat Conference in 2012 researchers from the MWR Labs\(^1\) presented vulnerabilities in payment terminals as well as ways to exploit them in order to run arbitrary code [24]. More specifically, the researchers demonstrated a way to run malicious code in the payment terminal in order to record the cardholder details (account number and PIN) of future customers by inserting a malicious smart card in the Integrated Circuit Card (ICC) Interface. After they performed a seemingly normal and successful transaction, they were able to harvest the recorded data just by inserting another malicious smart card in the same interface.

Europay MasterCard Visa (EMV), the protocol behind the Chip and PIN technology, was written in 1993 and 1994 and one of its goals was to secure the financial transactions from fraudsters. Other reasons for the advent of this technology is to enforce better decision support at the point of service concerning the risk management of the transaction as well as more robust cardholder verification methods [40]. Today, 32% of financial transactions use the EMV protocol [26]. As seen in Figure 1.1, EMV is widely used in Europe, Canada, Latin America, Africa and the Middle East while Asia and the United States have not fully adopted it yet. According to Figure 1.2 in 2014 the number of EMV enabled smart cards was estimated around 3.4 billions.

These statistics which show the emerging prevalence of EMV render not only the smart cards but also the payment terminals a valuable target to possible fraudsters. The fact that payment terminals are publicly available and in some cases unattended (e.g. at gas stations) makes these concerns more justified. An attacker might exploit a vulnerability in the system in order to compromise it and make it act maliciously.

In order to identify possible vulnerabilities in a system, testers often make use of the fuzzing testing technique. Fuzzing is a method which feeds the system with malformed and unexpected data in order to make it respond in a way not defined in the specifications. Examples of such behavior is the System Under Test (SUT) to crash, hang (become unresponsive) or display data that was not supposed to be displayed. In this way, fuzzing can uncover possible vulnerabilities in the SUT which would otherwise go unnoticed.

The focus of this research is the implementation of a testing methodology in order to perform fuzzing of the ICC Interface of payment terminals and the development of a proof of concept tool. For this we make use of the EMV protocol.

\(^1\)https://labs.mwrinfosecurity.com
CHAPTER 1. INTRODUCTION

![Figure 1.1: Percentage of Card-Present Transactions that are EMV (reproduced from [26])](image)

<table>
<thead>
<tr>
<th>Region</th>
<th>2013</th>
<th>2014</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>EMV Cards</td>
<td>Adoption Rate</td>
</tr>
<tr>
<td>Canada, Latin America, and the Caribbean</td>
<td>471M</td>
<td>54.2%</td>
</tr>
<tr>
<td>Asia Pacific</td>
<td>942M</td>
<td>17.4%</td>
</tr>
<tr>
<td>Africa &amp; the Middle East</td>
<td>77M</td>
<td>38.9%</td>
</tr>
<tr>
<td>Europe Zone 1</td>
<td>794M</td>
<td>81.5%</td>
</tr>
<tr>
<td>Europe Zone 2</td>
<td>84M</td>
<td>24.4%</td>
</tr>
<tr>
<td>United States</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

![Figure 1.2: Worldwide EMV chip card deployment and adoption (reproduced from [26])](image)

Evaluation of the robustness of payment terminals with the use of fuzzing
1.1 Related Work

To the best of our knowledge academic research focused mainly on fuzzing the smart card end of the EMV protocol. In [10] the authors propose a fuzzing method to dig up vulnerabilities in smart card applets used for financial transactions and telecommunications. In [22] the authors propose a method in which fuzzing is used to reverse engineer hand held smart card readers for Internet banking. The Security Group of the Faculty of Computer Science and Technology in University of Cambridge has published several researches concerning the security of EMV from a logical point of view as well as from a hardware perspective [25][39].

The Black Hat Conference 2012 and 2014 proceedings presented by MWR researchers [24][23] as well as the research carried out by the German security research firm SRLabs [34] are examples of work which focuses on the software of the payment terminals.

1.2 Research Question

The research mentioned above, as well as the public demos of the exploitation of the vulnerabilities in payment terminals, make us formulate the following research question which is the basis of this very research.

How is it possible to use fuzzing testing techniques in an efficient way so as to evaluate the robustness of payment terminals.

This question breaks down to:

- Analysis of the protocol to be fuzzed.
- Finding the most suitable fuzzing testing methodology to be used given the time and the resources available for this interface and protocol.
- Devising an instrumentation of the SUT and the testing equipment that could lead to reliable results.

In the rest of this research we will try to answer this questions and propose a solution.

1.3 Contributions

Our contributions are summarized here:

- Literature study of the state of the art in the fuzzing field and suggestions on when it would be suitable for each one of them to be used.
- Analysis of the EMV protocol and the Tag Length Value (TLV) structure that it uses from the fuzzing point of view.
- Proposition about an efficient fuzzing methodology given the time and resource constraints.
- Development of a proof of concept tool which is able to deliver around 200 test cases to the ICC interface of any payment terminal.
- Test of the tool on various payment terminals from different vendors.

\[^2\text{http://www.cl.cam.ac.uk/research/security/}\]
\[^3\text{https://srlabs.de/}\]
1.4 Research Methodology

- Literature Study. We made a research over various fuzzing methods and the EMV protocol. We ended up in a classification of the fuzzing methods and a proposal about which is the most suitable solution for the EMV protocol.

- Evaluation method based on fuzzing. We developed an evaluation method of the robustness of payment terminals with the use of fuzzing. We implemented a proof of concept tool which generates and delivers the fuzzing test cases to the SUT.

1.5 Outline

This research is divided in two parts. The first part is the literature study concerning the fuzzing as a testing method. The second part is the actual implementation of the tool and the results from the tests. The chapters are organized as follows.

Chapter 2 describes a two axis classification of the various fuzzing methodologies. The first axis is the knowledge which the fuzzer needs in order to generate the test cases and the second axis is the intelligence of the fuzzer. Chapter 3 gives a high level overview of the payment terminals. Chapter 4 is an introduction to the EMV protocol. It provides all the necessary information in order for the reader to understand the implementation details. Chapter 5 describes the implementation of the fuzzing tool. Chapter 6 describes the testing methodology. Chapter 7 presents the result of the fuzzing tests with five payment terminals the details of which (vendor, model etc.) we cannot reveal. The thesis is concluded in Chapter 8 with a summary, conclusions and propositions for future work.
Chapter 2

Fuzzing Methods Classification

In this chapter we make a classification of the various fuzzing methods. It is by no means exhaustive but rather representative of the methods that already exist or have been proposed. Firstly we define fuzzing as a software testing technique. Secondly we describe the two dimensions of our classification. Thirdly we give an overview of the fuzzing methods according to the classification. We summarize this chapter in terms of pros, cons and cases in which each one of the fuzzing methods can be used.

2.1 Fuzzing Definition

Software Security is a wide term used to specify the engineering of software so that it continues to function properly under malicious attacks [35]. Hundreds of software security best practices have been published during the years in order to mitigate or at least suppress the impact of known vulnerabilities. The principles of Least Privilege, Fail Securely, and Separation of Data and Code are some of them [2]. In order to challenge the level of security in the software, during the Integration and Testing phase of the Software Development Life Cycle (SDLC) the implemented software is tested and inspected with the use of several testing techniques. One of these is the Blackbox testing technique.

**Definition** Black box testing (also called functional testing) is testing that ignores the internal mechanism of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions [13]. The qualities examined by the Blackbox testing include:

- conformance of the software against the specifications
- performance and efficiency of the software in real-use scenarios
- robustness of the software.

From the security point of view the last category is the most crucial as it incorporates the negative testing of the software.

Every system should be specified in terms of positive and negative requirements. The positive requirements describe what the system should do while the negative requirements describe what the system should not do [48]. The negative testing examines that the negative requirements have been successfully incorporated into the software.

If for example one wants to test an authentication function, the positive testing would be to use the legitimate credentials while the negative testing would be to call the function with all the other illegal combinations of username and password. One of the various types of Black-box negative testing is Fuzzing or Fuzz Testing.
Definition  Fuzzing or fuzz testing is a negative software testing method that feeds malformed and unexpected input data to a program, device or system[49].

"Differently from other Quality Assurance (QA) tests, such as conformance testing, the space of illegal inputs is infinite and therefore a system can never be tested with all possible illegal inputs" [11]. This is the reason why fuzzing, at its early version delivered purely random input data to the software or device in order to dredge up system behavior such as crashes, information leaks etc. "Its main focus it is not to validate any correct behavior of the software but to explore the challenging area of negative requirements" [48].

The term fuzzing was first coined in 1988 by Professor Barton P. Miller for the needs of a project assignment in the Advanced Operating Systems course. The motivation behind the project was the crash of common UNIX utilities while remotely operated, under noise due to heavy rain. Since then, and against the disbeliefs and criticism due to its random nature, fuzzing has evolved in a valuable tool used in the SDLC. Also, many manufacturers have adapted fuzzing tools (proprietary, commercial or open source) into their QA process [48].

2.2 Fuzzing methods classification

The two dimensions we are going to discuss here are:

• the intelligence of the fuzzer during the fuzzed data creation and

• the level of target awareness (i.e. what the fuzzer knows about the target).

2.2.1 Intelligence of Fuzzer

We distinguish four different categories of fuzzers according to the way they produce the data. These are the Random, the Mutative, the Generative and the Robustness Testing category.

Random

This class of fuzzing methods does not include any kind of intelligence during the data generation. The choices made by the fuzzer are totally random. On the one hand this is the simplest type of fuzzing to implement and it requires no knowledge of the target. On the other hand it suffers from low code coverage and subsequently from poor bug finding capability [49]. In bibliography it is not uncommon for this class of fuzzers to fall under the generative class which will be explained later. For the purposes of this research we separate the two since, in terms of intelligence, there are differences between them.

Mutative

The fuzzers belonging in this class basically distort a known valid input sample in order to create numerous semi-valid ones. This can be done in two ways, either completely randomly or by merging some knowledge about the field boundaries within the message structure (e.g. which field is data and which is length). For the former variant no knowledge of the input format is required. For the latter the tester must have some basic understanding of the message structure. What is required though for mutative fuzzers is the initial data set captures which provide a sort of partial interface definition[49].

This fuzzing category is much simpler to implement over its generative counterpart since it does not incorporate the same amount of protocol knowledge during fuzzing. This advantage can be proven useful especially when the target interface runs complex protocols. Another benefit is that they are mostly general purpose\(^1\). But its effectiveness is directly dependent to the initial data

---

\(^1\) General purpose fuzzers are built to test more than one protocol. The opposite of general purpose is one-offs which are built to test only one protocol or even only one application.
set \[38\]. If a particular feature is missing from the initial data sample, then this feature will not be exercised by the fuzzer as well.

**Generative**

Generative fuzzers create totally new input data by themselves. In order to be able to create the new input, generative fuzzers require knowledge of the data structure. As explained before, we do not consider the random fuzzers to fall under this class even though they too create new input but without using any kind of rationality. The detail as well as the implementation level of the embedded knowledge determines the efficiency of the fuzzer. Generative fuzzers are mostly one-offs (built to test one protocol). Their implementation requires comprehension of the tested protocol by the tester-developer \[47\]. This feature makes their design complex. This class of fuzzers can succeed high code coverage especially with complicated protocols where mutative fuzzers may fail \[38\].

**Robustness Testing**

"Robustness can be defined as an ability to tolerate exceptional inputs and stressful environmental conditions" \[49\]. The property of this class is the lack of randomness during the test case generation. A library of known to trigger vulnerabilities input vectors (anomaly library) is used instead to stress the target to its limits. Even though fuzzing contains the notion of randomness we include this class because some of the methods we are about to describe make use of the anomaly library along with the random input generation.

The advantage of this class is the fact that it does not consume resources (e.g. effort, time) in order to create vectors that would cause an unexpected behavior. The drawback is that is totally based on previously known input vectors and it is unlikely to discover new types of vulnerabilities.

**2.2.2 Target Awareness**

The next classification axis is the level of knowledge a fuzzer uses in order to create the fuzzed data. This level varies across Black-box, White-box and Grey-box.

**Black-box**

Black-box fuzzers assume no knowledge about the target. For the purpose of this research, fuzzers that require an initial valid data set fall under this class (i.e. we do not consider the initial data set as knowledge about the target since it can be obtained with a sniffer). Also in this class we include fuzzers that assume access to the binary code of the application. Black box fuzzers differ from black box functional testing in that the latter examines what the program does or in other words how it behaves. It is also mentioned as black-box testing because the tester considers the program as a system which accepts inputs and deliver outputs.

**White-box**

Fuzzers that fall under this class know everything about the target. By everything we mean the protocol specification, the input data structure, the state transition of the target\(^2\), even the source code of the application and the OS on which it runs. White-box fuzzers differ from structural testing in that the latter examines how the program does what it does. It is known as white-box testing since it examines the inner structures of the target.

\(^2\)The target is considered a Finite State Machine (FSM) with a finite number of states. The state transition is the change from one state to another with regard to outer stimuli (e.g. incoming messages)
CHAPTER 2. FUZZING METHODS CLASSIFICATION

Grey-box

Grey-box fuzzers fall between the other two classes. They require partial knowledge of the target. This knowledge could be the message structure or (partial) protocol details.

2.3 Classification

In this section we classify the various fuzzing methods with regard to the axes described in Section 2.2. As already stated the list of methods described in this section is not exhaustive but rather representative. Our goal is to provide a summary of the methods’ suitability, features, advantages and disadvantages.

Oblivious

Oblivious fuzzing incorporates no knowledge concerning the message structure or the input in general that the target expects to see. The fuzz tool [37] is an example of this method. According to the fuzz tool documentation: “fuzz is a program designed to find bugs in other programs. It does this by doing a form of black-box testing. It does this by running the target program multiple times and passing it random input.” The mutative version of the oblivious method chooses from an initial input bytes at random and replaces them with random ones [36].

Heuristic-Based

A fuzzer that incorporates heuristics, produces the test cases by making use of libraries of common error sources [36]. These may include strings of certain lengths or numeric data values at the boundaries of the various numeric data types (byte, int, short etc.) [38]. Such a fuzzer requires a partial knowledge of the input structure since it has to be able to change each field value with a corresponding one from the anomaly library. In [47] Sutton names this method as Pregenerated Test Cases.

An advantage of this technique is that it can be used to test multiple implementations of the same protocol. The drawback is that these fuzzers are limited in trying to uncover known vulnerability types. Since there is no random component once the list of test cases is exhausted fuzzing is finished [47].

Evolutionary-Based

In the field of artificial intelligence, a Genetic Algorithm (GA) (also known as Evolutionary Algorithm (EA)) is a search heuristic that mimics the process of natural selection. This heuristic is routinely used to generate useful solutions to optimization and search problems [54]. GA’s are comprised of three components:

1. the initial population of solutions
2. the fitness function which determines how fit each individual solution is and
3. the reproduction function which is responsible for the crossover between two parent solutions and the mutation of the descending ones.

Evolutionary-based fuzzers use GA’s in order to craft fuzzed input data. They are initialized with a pool of candidate solutions. In every run the fitness function tags every candidate solution with a score. This score is dependent to fuzzing metrics. It could be related to the code coverage every solutions succeeds, if the target crashes or not or to the responses of the target. According to this score the reproduction function chooses a number of solutions in pairs, performs a crossover in each pair and finally mutates the derived solutions [47]. Assume for example that we have a function with 2-byte inputs. The candidate solutions which scored the best code coverage are:
• parent 1: 1001001000001110 and
• parent 2: 1100100100010001.

The crossover would result in:
• offspring 1: 100100\mid0100010001
• offspring 2: 110010\mid1000001110.

Finally after the mutation the final solutions would be:
• offspring 1 mutated: 1001000101010001
• offspring 2 mutated: 1100101001001110.

The evolutionary-based fuzzing does not try to find the protocol message structure in order to use it (mutatively or generatively). What it does is to produce the fittest solution according to a given fitness function. Evolutionary Fuzzing System (EFS) is a fuzzing framework which is based on GA’s [49].

Automatic Protocol Discovery

In this paragraph we describe a family of fuzzing methods named as Automatic Protocol Discovery.

The fuzzers developed under this method try to extract the protocol messages structure with various techniques, to some of which we refer in this section. This can be useful in cases when the protocol which the target uses is proprietary and there is no publicly available information for it or when the modeling of the protocol from the specification is tedious.

Automated Protocol Modeling

In [50] the authors (OUSPG group of the University of Oulu) propose a method for automated protocol modeling as part of the PROTOS Genome project [6]. Their solution takes as input some training material (message captures) and produces a description of the message structures using a Context Free Grammar (CFG) [53] by using the custom made MADAM algorithm.

In Figure 2.1 an example of the MADAM algorithm is illustrated. In the box at the right side is the CFG rules (denoted as THE GRAMMAR). The non-terminals 0, 1, 2 stand for the original training messages as they were introduced to the algorithm (left side of Figure 2.1). Terminals 3, 4, 5 represent the strings that were recognized in the training messages more than once. After the production of the CFG rules the fuzzed data generation phase takes place as illustrated in Figure 2.2.

The left side scheme does not contain any fault injection as it is the creation of one of the training messages. The right side scheme contains a mutation in it as the second "GABBA_" string is omitted thus resulting to the fuzzed data "GABBA_HEY". The mutation rate can be manually
CHAPTER 2. FUZZING METHODS CLASSIFICATION

Figure 2.2: Regular and mutated data generation runs using the CFG inferred in Figure 2.1 (reproduced from [50])

![Diagram of regular and mutated data generation runs]

Figure 2.3: Incoming message with length field value

![Diagram of message structure with length field]

Disassembly Heuristics  The previous method inferred the protocol messages structures by analyzing the messages themselves. Apart from the input itself, information can be emitted also by the processing of the input.

In [20] the authors propose a black-box approach which is based on the concept that the way an application processes input data leaks out information about the message format. The proof of concept tool of this solution is called Polyglot and it keeps track of the program binary execution given known input data. It does this by attaching a debugger to the target system and by making use of dynamic taint analysis. During dynamic taint analysis [45] input data is marked (tainted) upon arrival and the instructions (arithmetic, logical or move instruction) that operate on that data propagate the taint information to the destination.

For example assume that the target application receives a message whose structure is depicted in Figure 2.3. It consists of a 2 byte Length Field which indicates the length of the Variable-length field, which in our case is 4 bytes, and a Fixed-length field. Here we identify the value at positions 12-13 as a length field because it is used (probably in a pointer increase) to access the value at positions 18-20. The indication of variable length fields though is not performed solely with the use of direction fields\(^4\). Another way is to use a separator\(^5\). Such an example is depicted in Figure 2.4.

In positions 14-15 there is the "\r\n" (0d0a in ASCII) separator. The algorithm will probably have the line

\[ \text{while(current\_byte} \neq \text{"\r}\text{ AND current\_byte}+1 \neq \text{"\n"}\{\text{process}\}. \]

The assembly code corresponding to the line above contains numerous comparisons with the 0d0a value. By that we can deduce that the "\r\n" string is used as a separator.

The advantage of this method is that, since it "follows" the instructions which are dependent to the tainted data, the output (message format) has not only syntactic but also semantic information. That is it can identify fields such as checksums.

Other tools which are based on the disassembly heuristics method is the commercial beSTORM [1]

---

\(^3\)https://www.ee.oulu.fi/research/ouspg

\(^4\)Direction fields are fields that store information about another field in the message which is called target field.

\(^5\)Separators are elements used to sign the boundaries of variable-length fields.
CHAPTER 2. FUZZING METHODS CLASSIFICATION

Figure 2.4: Incoming message with separators

fuzzer [49] and the PaiMei reverse engineer framework [47].

**Proxy Fuzzing**  A proxy fuzzer as its name implies sits in between the client and the server. Both client and server see the fuzzer as the other end of the communication. In this setting, the fuzzer initially just relays the messages and records them. As soon as the fuzzer identifies sequences of plaintext (i.e. characters that fall within the valid ASCII range) it ignites the mutation engine to distort the field or replaces it with a string known to be malicious (e.g. a format string sequence). The proxy fuzzing method can leverage valid transactions and generate test cases valid enough to reach the target [47].

In [46] the authors propose a proxy fuzzing technique where the fuzzer uses the Lempel-Ziv (LZ) compression algorithm to estimate packet structure units. The proof of concept tool is called LZFuzz and it is based on the General Purpose Fuzzer (GPF) for the creation of the test cases.

**Bioinformatics** Bioinformatics or computational biology is defined as the use of applied mathematics, informatics, statistics, computer science and engineering to solve biological problems usually on the molecular level [51]. The common ground (at least the one we are interested in) between biology and computer science is the processing of long yet structured sequences of information. In the case of biology we refer to the gene sequence and in the case of computer science we refer to the field sequence within the protocol messages.

One algorithm which succeeds in aligning such sequences is the Needleman-Wunsch (NW) algorithm whose details are beyond the scope of this research. Given two amino acid sequences, ACATTACAGGA and ACATTCCTACAGGA, the algorithm would align them as follows [47]:

\[
\begin{align*}
\text{sequence one:} & \quad A \, C \, A \, T \, T \, A \, C \, G \\
\text{sequence two:} & \quad A \, C \, A \, T \, C \, C \, T \, A \, C \, G \, G \, A \\
\end{align*}
\]

The \_ character stands for gaps inserted by the algorithm to maximize the similarity of the sequences.

In [14] Beddoe presented an experimental Python framework named Protocol Informatics. The objective of the framework was to deduce the field boundaries of the various protocol messages by observing and aligning large quantities of data.

For example given the two HTTP sequences "GET /index.html/ HTTP1.0" and "GET / /HTTP1.0" the framework would result in the following output:

\[
\begin{align*}
\text{sequence one:} & \quad G \, E \, T \, / \, i \, n \, d \, e \, x \, . \, t \, m \, t \, / \, H \, T \, T \, P \, 1 \, . \, 0 \\
\text{sequence two:} & \quad G \, E \, T \, / \, . \, . \, . \, . \, . \, . \, . \, . \, . \, . \, / \, H \, T \, T \, P \, 1 \, . \, 0 \\
\end{align*}
\]

By evaluating this output a fuzzer can infer that "index.html" is variable field and either mutate it or replace it with a string from its anomaly library.

**Block-based and model-based fuzzing approach**

Even though the original definition of fuzzing [37] is based on a completely randomized approach, block-based and model-based fuzzing use the knowledge of the message structure and sequence in order to either generate semi-valid data or mutate initial valid data.
In every network protocol the lower layers should know the length of the message which they are about to encapsulate. This makes the construct of the fields of the lower layers dependent from the content of the higher layers. So when it comes to fuzzing these protocols the tester should manually calculate or at least create a routine that calculates the length of the different layer messages. This process is somehow tiresome and renders the fuzz scripting inefficient.

The block-based approach models each tested protocol as blocks consisting of length fields and data fields [8]. In that way the target will not discard messages because of faulty length fields and the fuzzer will generate protocol messages compliant to the input format.

According to block-based fuzzing the fuzzer can construct a certain message and fuzz (insert random or semi-random) data to a specific location [42]. For example a common GET request template would look like this:

```
GET /file.name HTTP/1.1
```

In the following GET request the fuzzer chooses to fuzz the block signified by [ ]:

```
GET /[]file.name HTTP/1.1
```

The most common representative fuzzing framework of this category is SPIKE. SPIKE is an open source tool written in C and provides a scripting environment to the tester to develop fuzzers for network protocols. In this environment the tester can declare the different blocks within a message without having to calculate each block's length.

For example consider the following SPIKE script that could be used to fuzz the inputvar variable in php script testme.php via a POST request to testserver.example.com [3]:

```c
//fixed strings//
s_string("POST /testme.php HTTP/1.1\r\n");
s_string("Host: testserver.example.com\r\n");
s_string("Content-Length: ");
//adds a string 5 characters long which represents the size of block1 declared later
s_blocksize_string("block1", 5);
//fixed string
s_string("\r\nConnection: close\r\n\r\n");
//declare of start block block1
s_block_start("block1");
s_string("inputvar=");
s_string_variable("inputvar");//the fuzzed string declare of end block
s_block_end("block1");
```

The above script results in the following message:

```
POST /testme.php HTTP/1.1
Host: testserver.example.com
Content-Length: [size\_of\_data]
Connection: close
inputvar=[fuzz\_string]
```

The [fuzzle\_string] SPIKE\(^6\) will be replaced by random or strings of lengths known to cause erroneous behavior. SPIKE will automatically replace the [size\_of\_data] according to the resulting length of block1 and thus disencumber the tester from the burden of calculating manually the length.

One other feature of SPIKE is the fact that it can integrate well with other C libraries which calculate checksums or perform cryptographic functions [9].

The block-based fuzzing method does not need access to the source code of the target. The tester only needs to know the format of the input data in order to generate the test cases. Know-

\(^6\)The injected fuzzed data in the messages are called SPIKES
knowledge of the underlying protocol is not necessary but it helps the creation of a more complete test case set. Other tools which fall into this category are Sulley [12] and Peach among others.

In [44] Schieferdecker, Grossmann and Schneider propose a fuzzing method during which the sequence of otherwise valid messages is fuzzed (behavior fuzzing).

In many cases the protocol can be modeled as a FSM. The proposed method examines the robustness of targets, the requirements of which are authentication of users or avoidance of replay of previous messages in order not to redo the transaction. For example if the security requirement of an application is to tag every transaction with a previously generated random number, then a vulnerable system would perform the transaction without checking the random number. Summarizing what the behavior fuzzing does is to mutate the valid state diagram of the protocol in order to exercise the target’s response.

**Mini-simulation**

In [33] Kaksonen proposes the *mini-simulation* method. According to this method one can create several different models (mini-simulations) of the protocol written in an attribute grammar based on the Backus-Naur Form (BNF) language just by using only one protocol specification. The goal of the method is to create a large amount of messages with one or few exceptional elements but otherwise legal content. Each of these messages or group of messages (test cases) are created by using multiple miniature simulations rather than a single complete one which most of the times tends to be complex. The fuzzer parses each one of these mini-simulations and sends a deriving test case to the SUT and keeps track of which message elements violate the protocol specification.

Figure 2.5 depicts the mini-simulation concept. The master specification contains a simple description of typical and/or error free messages. The description should be simple since the mini-simulation method is not a design aid (like Unified Modeling Language (UML)) but rather a
testing and debugging aid. Including error messages\(^7\) is a good idea since they produce test cases which exercise the error handling routines of the protocol.

The rule library contains a set of semantic rules for the attribute grammar and communication rules which instruct the communication of the model and the external world. The rule library is implemented in Java. The semantic rules are used to implement constructs which are difficult or impossible to describe using only pure grammar, such as lengths or checksums. For example the semantic rule SequenceNumber implements a sequence of numbers. The rule's property number indicates where the number is going to be placed and the property step increments the number counter. The communication rules are means of data exchanging between the model and external entities. An example of communication rule is to decode from an input steam from a TCP socket and encode to the output stream.

The configuration (or configuration scripts) specifies the required components and sets their parameters (written in Tcl).

The operations are used for the grammar mutations.

The configuration parses the master specification and derives the protocol message syntax as well as the message interaction. It does that by using the operations and rule library and by modifying the input grammar to the interaction model. The Defensics fuzzing tool distributed by Codenomicon is software based on the mini-simulation approach.

**Whitebox fuzzing approach**

The whitebox fuzzing approach is a fuzzing method and we use this name after [29]. The whitebox class as explained in the beginning simply means that the tester knows everything about the target. In order to avoid misunderstandings we will refer to the fuzzing class as "white-box" and to the fuzzing method as "whitebox".

The black-box approach of fuzzing suffers from some limitations concerning the low code coverage that a fuzzer can succeed [28]. For example consider the following C code fragment:

```c
int foo(int x)
{
  // x is an input
  int y = x + 3;
  if (y == 13) abort(); // error
  return 0;
}
```

Listing 2.1: sample code for the whitebox fuzzing approach

In order for a fuzzer to exercise the if branch it needs to hit the \(x = 10\) input. This hit has \(\frac{1}{2^{32}}\) chances of succeeding and so does the execution of the if branch if \(x\) is a random 32-bit value.

In [29] Godefroid, Levin and Molnar propose a whitebox fuzz testing approach to overcome the above mentioned limitations. Their approach makes use of the concepts of symbolic execution\(^8\) and dynamic test generation in order to execute paths not easily discovered by a blackbox fuzzer. The proof of concept of their approach is the SAGE (Scalable, Automated, Guided Execution) fuzzing tool for x86 Windows applications.

**Dynamic Test Generation**  Dynamic test generation consists of the following steps:

1. real execution of the program with an initial (preferably well-formed) input
2. utilization of symbolic execution on the input from step 1 to collect path constraints

\(^7\)By error messages we mean messages that indicate an error and not messages that contain errors.
\(^8\)Symbolic execution or symbolic evaluation is a means of analyzing a program to determine what inputs cause each part of the program to execute (from wikipedia.org). For example in the above code symbolic execution on the input \(x = 0\) generates the path constraint \(x \neq 10\). As soon as this path constraint is negated it yields \(x = 10\) (for more complex code a constraint solver is used) which is the input that exercises the then branch of the if statement.

14 Evaluation of the robustness of payment terminals with the use of fuzzing
CHAPTER 2. FUZZING METHODS CLASSIFICATION

Figure 2.6: Search space for the code in listing 2. Each node contains one of the path constraints in \( p = \langle i_0 \neq b, i_1 \neq a, i_2 \neq d, i_3 \neq \rangle \). The left branches contain the inputs that comply with that constraint while the right branches contain the inputs that do not. Along with input there is its corresponding \( \text{cnt} \) value. The inputs with \( \text{cnt} \geq 3 \) exercise line 7 in listing 2.

3. utilization of a constraint solver (e.g. Z3 software) to infer variants of the previously used input in order to guide subsequent program executions aiming to better code coverage and

4. replace the input in step 1 with the output in step 3 and repeat until all (or specific) paths have been executed.

```c
    void top (char input [4]) {
        int cnt = 0;
        if (input[0] == 'b') cnt++;
        if (input[0] == 'b') cnt++;
        if (input[0] == 'b') cnt++;
        if (input[0] == 'b') cnt++;
        if (cnt >= 3) abort(); // error
    }
```

Listing 2.2: sample code for the dynamic code generation demonstration

In the above listing the probability of \( \text{if (cnt} \geq 3) \text{ abort(); // error} \) line being executed is \( (2^{10} - 3)/2^{32} \) considering a 4 byte input which is unlikely for a blackbox fuzzer.

The dynamic test generation in this example would start with input\[]='good' (step 1). The first leaf in Figure 2.6 depicts the first run of the program under that input and the resulting \( \text{cnt} \) value. Along with the real execution a symbolic execution on the same input would result in a path constraint \( p = \langle i_0 \neq b, i_1 \neq a, i_2 \neq d, i_3 \neq \rangle \) (step 2). A constraint solver would infer a new input\[]='goo!' by negating the last path constraint \( i_3 \neq \rangle \) (step 3). The second leaf of Figure 2.6 depicts the path of the new input\[]='goo!' along with the corresponding \( \text{cnt} \) value. After repeating steps 1 to 3

9 In order for (cnt \geq 3) == true there are 5 sets of input.

\( S_0 = \{ (\text{char})0 < \text{input}[0] < (\text{char})2^8, \text{input}[1] = 'a', \text{input}[2] = 'd', \text{input}[3] = '!' \} \),
\( S_1 = \{ \text{input}[0] = 'b', (\text{char})0 < \text{input}[1] < (\text{char})2^8, \text{input}[2] = 'd', \text{input}[3] = '!' \} \),
\( S_2 = \{ \text{input}[0] = 'b', \text{input}[1] = 'a', (\text{char})0 < \text{input}[2] < (\text{char})2^8, \text{input}[3] = '!' \} \),
\( S_3 = \{ \text{input}[0] = 'b', \text{input}[1] = 'a', \text{input}[2] = 'd', (\text{char})0 < \text{input}[3] < (\text{char})2^8 \} \),
\( S_4 = \{ \text{input}[0] = 'b', \text{input}[1] = 'a', \text{input}[2] = 'd', \text{input}[3] = '!' \} \).

The cardinalities of these sets are \( |S_0| = |S_1| = |S_2| = |S_3| = 2^8 \), and \( |S_4| = 1 \). If we add them altogether we get: \( 4 \times (2^8 - 1) + 1 \) (we subtract 1 because the \( S_4 \) set is included in all the other sets). So the probability of (cnt \geq 3) == true is \( 4 \times (2^8 - 1) + 1/2^{32} = (2^{10} - 3)/2^{32} = 0.0000002377200872 \)

Evaluation of the robustness of payment terminals with the use of fuzzing 15
all the 16 resulting values are shown in Figure 2.6. The authors propose the use of a generational search algorithm whose goal is to prevent overlap in the search of new inputs and maximize code coverage with the goal of finding bugs faster.

**SAGE High-level Architecture**  
SAGE uses the algorithm explained in the previous section along with the generational search algorithm [29] to improve the results. Figure 2.7 depicts the high-level architecture of SAGE. By using Input0 as the initial input the tool first checks for bugs (AppVerifier). If no bugs are found it calculates the code coverage in terms of unique program instructions that were executed in this run (Nirvana) and then it generates the constraints of the path (TruScan). Finally after the constraint solver (Z3) is executed it produces a new set of inputs (Input1, Input2, ..., InputN) which are tested for bugs and assigned a score according to how many new instructions they executed in relation to Input0.

SAGE and in general the whitebox fuzzing technique does not necessarily need knowledge about the input format (it is most efficient if it is available) but it assumes source code access. With this in mind we would say that the limitations of white-box fuzzing are:

- Source code is not always available or easy to find especially when it comes to proprietary code.
- White-box fuzzing has a rather complex implementation. The instrumentation of the tools depicted in Figure 2.7 is beyond the simple nature of the fuzzing concept.
- The test case generation is dictated by the path constraints. This feature by itself is not problematic, but when it comes to fuzzing we want the test cases to stand somewhere between the total agreement with the specifications and random inputs.

One advantage of this technique is the great code coverage that it succeeds and the efficient generation of the test cases. Another positive aspect is the fact that it can be used in combination with a mutative fuzzer. SAGE could provide a representative test case set which exercises all the possible paths in the code and the mutative fuzzer mutates each test case in order to create the fuzzed input data.

**Grammar-based whitebox fuzzing**
The previously discussed whitebox fuzzing approach has limited efficiency when the targets are applications which use highly-structured input, such as compilers and interpreters. In [29] Godefroid, Kiezum and Levin propose a white-box fuzzing method which makes use of the input grammar.

Compilers and interpreters are classical environments the input of which passes a lot of phases (lexer, parser, code generator and finally interpreter) before being interpreted or compiled. In these situations we encounter conditions such as:

- code explosion: the different code paths are increasing exponentially and
- in some cases the whitebox fuzzer cannot even reach a certain code path (i.e. given a constraint \( x=\text{hash}(y) \) and a value for \( x \), the computation of a value for \( y \) that passes this constraint is impossible)

that defeat the whitebox approach.

The grammar-based extension of whitebox fuzzing:

- requires a grammar that describes valid program inputs
- uses a higher-level symbolic execution which derives grammar-based constraints expressed in the form of symbolic grammar tokens and a constraint solver which checks for solution which not only satisfy the constraints but also are compliant with the grammar rules
- runs more efficiently by getting rid of the tree paths (Figure 2.6) that are not compliant with the grammar.

For example consider the following grammar subset of Javascript:

```plaintext
1 FunDecl ::= function id ( Formals ) FunBody
2 FunBody ::= { SrcElems }
3 SrcElems ::= empty_string
4 SrcElems ::= SrcElem SrcElems
5 Formals ::= id
6 Formals ::= id , Formals
7 SrcElem ::= . . .
```

Listing 2.3: sample code for the grammar-based whitebox fuzzing demonstration

empty string stands for empty string and nonterminals have names starting with uppercase.

By running the Javascript interpreter with initial input:

```javascript
function f() {}
```

would trigger a symbolic execution the result of which would be symbolic grammar token sequence:

- \( \text{token}_0 = \text{function} \)
- \( \text{token}_1 = \text{id} \)
- \( \text{token}_2 = ( \)
- \( \text{token}_3 = ) \)
- \( \text{token}_4 = \{ \)
- \( \text{token}_5 = \}

If we negate the 4th token we get a new sequence of constraint:

- \( \text{token}_0 = \text{function} \)
- \( \text{token}_1 = \text{id} \)
- \( \text{token}_2 = ( \)
- \( \text{token}_3 \neq ) \)

From all the possible ways to satisfy this constraint the only valid one (according to the input grammar) is to set:

- \( \text{token}_3 = \text{id} \)
- \( \text{token}_4 = ( \)
- \( \text{token}_5 = \{ \)
- \( \text{token}_6 = \}

So the generated input that corresponds to this path constraint is function f(id).
If we chose to negate the 3rd token instead we would have something like:

\[ \text{token}_0 = \text{function} \]
\[ \text{token}_1 = \text{id} \]
\[ \text{token}_2 \neq ( \]

For which there are no grammar compliant inputs to satisfy it. That way grammar-based whitebox fuzzing approach prunes in one iteration all the possible inputs corresponding to this negation. That way fuzzing becomes more efficient than the whitebox approach since it needs less executions of the SUT in order to define the valid and invalid inputs. By that, the fuzzer obtains knowledge of which input will be rejected and the rules behind this rejection.

### 2.4 Discussion

After having presented the various fuzzing methods proposed in scientific publications we discuss the advantages, the disadvantages as well as the situations in which each one of the approaches are best suited. In Figure 2.8 we give a visual summary of the classification of the various methods presented earlier.

The **Oblivious** method has the advantage of being the simplest and fastest one to implement. By saying that, we mean that it requires no knowledge concerning the target and the fuzzed data is very easy to produce since they are totally random strings. The drawback is that this method is by far the least effective [47] since the fuzzed inputs will likely be rejected. A possible use of such a method is as a quick once-over in order to determine if an SUT has completely awful code [47].

The **Heuristic-Based** fuzzing method, as stated previously, has the advantage that the same fuzzer can be used to test several implementations of the same protocol. The disadvantage is that they do not incorporate any random elements and they use an anomaly library instead. This leads to a condition known as the *pesticide effect*\(^\text{10}\) This fuzzing method is more difficult to implement than the oblivious one. It is useful in situations when the tester knows the location and the type of the variable data into the input string or the file to be fuzzed.

The **Evolutionary-based** fuzzing method’s advantage is that it tries to maximize the efficiency of the fuzzed data without using any knowledge about the SUT but by leveraging the feedback. The disadvantage is that the fitness function may not always easy to determine in order to provide fine grained feedback to the fuzzer. The usage of such a technique is to assist a fuzzer in mutating inputs to traverse a small subpath in order to increase code coverage when necessary [47].

The **Automatic protocol discovery** drawbacks is the complexity of a task to break down a protocol into its basic building blocks. It is of no surprise that several of the protocol discovery techniques mentioned are still in a research phase. The advantage is that the fuzzed data produced are very effective in finding vulnerabilities since they incorporate the intelligence of the protocol discovery. This method is suitable for cases where the protocol to be fuzzed is proprietary, complex or undocumented.

The **Block-based fuzzing** has the advantage that it calculates fields such as lengths or checksums automatically. Another advantage of this technique is that several of the fuzzing frameworks (SPIKE, Sulley and Peach) are built based on it. The disadvatagage of this technique is that the tester has to write the fuzzer himself which is not always an easy task. This technique can be useful in situations when the protocol to be fuzzed has a known message structure and the tester wants to write a fuzzer based on an existing fuzzing framework.

The **mini-simulation’s** approach disadvantage is the one previously explained as the pesticide effect [49] which is common for solutions which do not contain randomness or do not get updated (the PROTOS tool which is based on the mini-simulation approach does not contain randomness—we cannot say if it is still updated or not).

**Whitebox fuzzing** approach suffers from the fact that it needs the source code of an SUT. The upside is that it succeeds great code coverage. The situations where this approach would come in handy is when a developer wants to test its own implementation.

---

\(^{10}\) Widespread use of the tool will result in vendors fixing the bugs that the tool is looking for. So the same tool will become less effective.

---

18 Evaluation of the robustness of payment terminals with the use of fuzzing
The **Grammar-based white-box fuzzing**, the advanced version of the previous approach, succeeds in defeating the limitations posed by code explosion issues. It is one of the most complex approaches presented so far in this very research. The situations where the use of this approach is advised is when not only the SUT’s source code is available but also when the SUT is a complex system e.g. a compiler. The final conclusion is that there is no "winner" among the approaches. Each one of them has its own pros and cons. The tester should take into account several factors before starting fuzzing the SUT. Some of the most important are:

- the time needed for the implementation of the fuzzing program and the running of the tests
- the SUT’s features (e.g. language used and OS)
- the kind of bugs the tester is looking for
- the available knowledge about the SUT.

---

**Figure 2.8: Classification of fuzzing techniques.**

<table>
<thead>
<tr>
<th>random</th>
<th>mutative</th>
<th>generative</th>
<th>robustness testing</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>white-box</strong></td>
<td>grammar-based</td>
<td>mini-simulation</td>
<td>heuristic-based</td>
</tr>
<tr>
<td></td>
<td>whitebox fuzzing</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>whitebox fuzzing</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>grey-box</strong></td>
<td>block-based and model-based</td>
<td>automated protocol discovery</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>black-box</strong></td>
<td>evolutionary-based</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>oblivious</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Chapter 3

Payment Terminals

3.1 Payment Terminals

In this section we give a high-level overview description of the payment terminals. Firstly we introduce some definitions. Secondly we explain why the security of a payment terminal is important. Thirdly we outline the interfaces that can be found in such devices as well as the protocols supported by each one of them. Lastly we summarize the OS and Programming Languages used by developers to write the applications that run in the payment terminals.

3.1.1 Related definitions

Payment Terminals go with many names such as PIN Entry Device (PED), Point of Sale (POS) terminal, Payment Application (PA), PIN Transaction Security (PTS) to name but a few. In [30] Gomzin makes a good separation between the various terms.

Definition PA is a software application created for electronic payment processing. PA implements all functions associated with acceptance, processing, storage, and transmission of sensitive card data.

Definition Point of Interaction (POI) (also called PED, PIN pad, pinpad, payment terminal) is a device that combines several functions and subsystems, such as Magnetic Stripe Reader (MSR), PIN Entry Keyboard, Customer Display, Tamper-Resistant Security Module (TRSM), and others.

Definition POS (also called register, lane, point of service) is a software application and/or hardware device that processes and records transactions (including payments) between the merchant and its customers.

For the purpose of this research when we refer to the term Payment Terminal we mean either the combination of a PA and a POI or a POS. If we want to specifically refer to one of the above we will use its defined name.

3.1.2 Assets of the payment terminal

In this subsection we will enumerate and give a brief explanation of the several assets of the payment terminals. This enumeration will help us to understand why this part of the financial transaction (the other is the smart card or even the bank) is important and it requires protection. Personal card details (Magnetic Stripe (MS) data and cardholder data). During the financial transaction the smart card sends to the terminal several details including the Primary Account Number (PAN), start and expiry dates and a copy of the data as they are stored in the Magnetic Strip of the card. These details are of interest to fraudster since they use them in order
CHAPTER 3. PAYMENT TERMINALS

to clone Magnetic Strip Cards with this data and use them in countries which do not yet use the EMV (or Chip and PIN) protocol [15].

**PIN.** The EMV protocol’s advent tried to mitigate the threat explained previously. Now the cardholder has to prove her identification by providing a PIN code. Now the security of the transaction had been split in two parts: the smart card (Chip) and the PIN. Since the customer of a store enters her PIN into the payment terminal the expectations from the device is to treat the PIN securely.

**Application code.** The bytecode and source code of the application that process the data and performs the financial transaction should not be tampered with. With this said we conclude that the integrity of the application code is extremely important.

**Device peripherals** (pin pad, display, interfaces). The customer’s interaction with the payment terminal consists of her interaction with a number of the terminal’s interfaces, like the pin pad (keypad), display or ICC Interface. Should one of them gets compromised then the financial transaction’s integrity gets compromised [25].

**Cryptographic keys stored in the device and public key certificates.** Inside the payment terminal are stored keys and public key certificates used for the creation of session specific keys to prevent replay attacks or when the authentication is performed offline (without sending the PIN to the issuer). These keys are stored in a Security Application Module (SAM) inside the terminal. The integrity and confidentiality of these keys should not get compromised.

**Passwords for device administration.** The terminal, during its lifecycle, needs to get configured. An example of such configuration is for the issuer to store or change the keys which reside into the terminal. In order to do that, special access passwords are needed which are stored (hashed). The integrity and confidentiality of these passwords are important.

The above list is by no means either exhaustive nor detailed. We just want to point out how important is the security of the Payment Terminals especially when they have to perform in hostile environments (for example unattended terminals at gas stations).

### 3.1.3 Architecture, interfaces, protocols and standards

Payment terminals’ system architecture consists of [41]:

- a CPU chip (usually of 8 or 16-bit architecture)
- a NPU chip (a supplementary Numerical Processing Unit works as a cryptoprocessor for the rapid computation of secret-key or public-key algorithms)
- a voltage converter and a clock generator (not present in the diagram)
- RAM (usually in the order of few megabytes) indicated as volatile memory in the diagram
- EEPROM or Flash EEPROM (terminals do not have their own disk drives) indicated as nonvolatile memory in the diagram

The terminal in order to communicate with the outside world utilizes interfaces and protocols. By outside world we mean the customer, the cashier or the issuer (the bank). Here we list and describe the most common interfaces¹, protocols and standards that run on each one of them (if any²).

**Security module/Security feature interface.** As stated above, the terminal contains cryptographic keys and other cryptographic primitives which needs to be either confidential or unmodified. In order to guarantee their security and confidentiality at all times, they are not stored in the normal electronic circuitry of the terminal, but in a separate security module within the terminal that has special mechanical and electrical protection [41].

---

¹some terminals also use a bar code reader but we think this is outside the scope of this research

²there are no protocols on display or on the keypad
CHAPTER 3. PAYMENT TERMINALS

Figure 3.1: Diagram of a payment terminal’s architecture

**Display.** This interface provides the visual information to the user. Care should be taken so as the integrity and the availability of the displayed info to be ensured. In other words, the user should somehow trust the display that will show her the correct amount and will not reveal under any circumstances her PIN. Nowadays, the display is not uncommon to get replaced by a touch-screen which acts as an interface to insert data to the terminal.

**Keypad.** This is the interface via which the user can insert the transaction amount or the PIN or can choose the operation she wants to perform. As in the case of the display, the keypad should transmit to the terminal what the user is intended to transmit and it should not reveal by any means the keystrokes executed by the user.

**Printer Interface.** This interface simply prints out the invoice of the transaction. Again here the integrity of the printed information is crucial.

**MS Interface or MSR.** The MS interface as its name implies scans the MS part of the card and scans the data. Modern MS interfaces have encryption capabilities and can be used in Point-to-Point Encryption (P2PE) solutions [30].

The data format of the magnetic tracks is defined by the ISO 7813 [32]. There are 3 tracks (called Track 1, 2 and 3 accordingly). For financial transactions only Tracks 1 and 2 are used.

**Track 1 contains:**
- the PAN which is the identification number of the account
- the expiration date of the account
- the cardholder’s first and last name
- the service code which gives an indication about the type of business environment in which the card is authorized for use in a financial transaction [40] and
- the Card Verification Value (CVV) which verifies the PAN of the card.

**Track 2 contains:**
• the PAN
• the expiration date of the account
• the service code and
• the CVV.

**Interface for cash register or similar.** In this category of interfaces we include the Universal Serial Bus (USB) and the Serial interfaces. We group them as one since they are used for connection with the cash register in order to enter the charging amount or for administrative and configuration purposes. The serial interface is used to connect the terminal to the cash register. The USB interface is used as a trusted way to download or update the software of the terminal.

**Chip Interface.** The chip interface is the one that communicates with the chip card (smart card). The chip card is a plastic card which incorporates a MS and an integrated circuit which is a single chip computer. The chip interface is equipped with a reader device and it acts as the client in a client server model with the smart card being the server [40].

The two standards that instruct the operation of this interface on payment terminals is the ISO/IEC 7816 and the EMV standard.

Chip cards are used not only in the payment industry but also in other fields. Telecommunications, pay TV, public transport, health care, passports and IDs are some of them. So there needs to be a standard in order to instruct the functionality of the ICC independently of the underlying application. The functionality of the ICC is based on the standard ISO/IEC 7816 [55]. ISO/IEC 7816 is comprised of 16 parts [55] the most important of which are analyzed in Chapter 4. For now we provide an overview of the standard.

ISO/IEC 7816-3 "Cards with contacts-Electrical Interfaces and Transmission Protocols" [41] specifies the electrical contacts via which the chip card is connected to the terminal. These 6\(^3\) contacts are:

• I/O: Input/Output for serial communications.
• Vpp: Programming Voltage (no longer used).
• Vcc: Supply Voltage.
• GND: Ground.
• CLK: Clock Input.
• RST: Reset Input.

ISO/IEC 7816-4 "Organization, Security and Command for Interchange" [41] specifies the format of the C-APDU and R-APDU which are the structures used by the terminal and the chip card respectively in order to communicate. Apart from the Application Protocol Data Unit (APDU) structure ISO/IEC 7816-4 defines the filesystem of the chip card’s OS. This is comprised of:

• the Dedicated Files which resembles to the directories in a UNIX like filesystem
• the Master File which can be considered as the root directory and
• the Elementary Files which can be considered as the files of a UNIX like filesystem.

Lastly, the ISO/IEC 7816-4 specifies some C-APDUs which are the same across many chip card applications and usages. These commands instruct the selection of files, the read access to the file’s records and the verification of the cardholder among others.

The other standard related to the ICC interface is the EMV protocol. EMV is built upon the ISO/IEC 7816 standard. It is the standard responsible for the actual financial transaction. Among other it specifies [27]:

\(^3\)one can count more than 6 contacts as some of them are deprecated or for future use
• the commands sent by the terminal and the interpretation of the response data sent by the chip card
• the cryptographic processes both ends should perform
• the authentication mechanisms imposed for the card, the cardholder and the terminal.

Modem. Modem is used for the communication of the payment terminal with the other end of the transaction which is the issuer (bank). The following table summarizes the various technologies used along with the protocols in order to achieve this connection [30]:

<table>
<thead>
<tr>
<th>Connection</th>
<th>Communication Protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dial-up</td>
<td>Serial</td>
</tr>
<tr>
<td>WAN</td>
<td>TCP/IP</td>
</tr>
<tr>
<td>Internet</td>
<td>TCP/IP, IPsec, VPN</td>
</tr>
</tbody>
</table>

As explained earlier the serial interface is used to connect the terminal with the cash register but in some cases it is used as a fallback mechanism in case of the main connection failure.

Infrared Data Association (IrDA) or Bluetooth interface. This interface is used for the communication between the terminal and another device such as a mobile phone. Along with these interfaces some modern terminals employ an Near Field Communication (NFC) interface which is essentially a collection of several protocols and standards (ISO/IEC 18000-3, ISO/IEC 14443-1, ISO/IEC 13157-1) [30].

3.1.4 OS’s and programming languages

In this subsection we first talk about the PA and then we discuss about the POS. We have to separate the PA from the POS (since both are considered as Payment Terminals). This is because the former is hosted in a device (e.g. a Personal Computer (PC)) which is connected to the POI while the latter integrates both the application and the interfaces as well as it provides an OS. In the Payment Card Industry (PCI) Security Standards Council website [5] one can find a complete list of PAs and POS terminals (named as PTS) approved by the PCI. For every PA, information concerning the OS with which is compatible is provided. Following is a list of the OS’s:

- CentOS
- Debian
- Fedora
- Gentoo
- FreeBSD
- Linux
- Windows 2000
- Windows 98
- Windows NT
- Windows Server 2003
- Windows Server 2008
- Windows Vista
- Windows XP
- Windows POSReady 2009
- Windows POSReady 7
- Windows CE
- Linux Mint
- Mac OS X
- Mac OS 9
- MEPIS
- NetBSD
- Oracle Linux
- Red Hat Enterprise
- OpenBSD
- Slackware
- SUSU Enterprise
- Solaris
- Ubuntu
- Windows 8
- Windows 8.1
- Windows 10
- Windows Server 2012
- Windows Server 2008 R2
- Windows Server 2008
- XenServer
- enterprise
- openSUSE

The most common languages used to write a PA are the following [43]: C/C++, Python, Ruby, Java, Perl and .NET framework.

For the POS we cannot draw any safe conclusion about which OS is used since this info is not always publicly available. We can only guess the use of a proprietary Embedded Operating System as for example the TeliumTetra OS by Ingenico Group [7]. The programming languages used are C/C++ or JavaPOS [4] [41].

24 Evaluation of the robustness of payment terminals with the use of fuzzing
Chapter 4

The EMV protocol

This chapter includes all the necessary information for the reader to understand the rest of this research. Section 4.1 gives an overview of the ISO/IEC 7816 standard as the ISO standard for the electronic cards with contacts upon which the EMV protocol is built. Section 4.2 introduces the EMV protocol. In Section 4.3 we discuss the C-APDU and the R-APDU as the main components of each EMV financial transaction. Section 4.4 explains the TLV structure and finally Section 4.5 enumerates the basic data elements of the EMV.

4.1 ISO/IEC 7816

Smart cards (contact and contactless) are used not only for payment applications but also in other sectors. The use of Subscriber Identity Module (SIM) cards in the telecommunication field and the use of smart cards in the public transportation field are some examples. In order for all of these applications to be independent of the physical layer as well as for compatibility reasons they are built upon the ISO/IEC 7816 family of standards.

The ISO/IEC 7816 name is "Identification Cards - Integrated Circuit(s) Cards with Contacts" and it describes the fundamental characteristics and functions of smart cards. In this section we go through the most important details of the several parts that comprise this family of standards. For the rest of the research anytime we refer to a part of the standard we do it by appending the number to the ISO/IEC 7816 name (e.g. ISO/IEC 7816-1 is the first part).

Every smart card communicates with the terminal via 8 contacts (Figure 4.1). ISO/IEC 7816-2 defines the positioning and sizes of the various contacts. ISO/IEC 7816-3 defines the basic electrical characteristics such as the supply voltage, the clock and the reset. Apart from that, it defines the possible transmission protocols (Table 4.1). This is the basic standard which describes all aspects of the communication at the physical level.

![Figure 4.1: Contact designations and functions according to ISO/IEC 7816-2](image)

ISO/IEC 7816-4 is the part which defines the filesystem organization within the smart card chip. It also specifies the basic communication unit namely APDU in which we will refer extensively.
CHAPTER 4. THE EMV PROTOCOL

Table 4.1: Summary of transmission protocols according to ISO/IEC 7816-3. T=0 and T=1 are of special interest in the EMV protocol (reproduced from [41])

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>T=0</td>
<td>Asynchronous, half-duplex, byte oriented</td>
</tr>
<tr>
<td>T=1</td>
<td>Asynchronous, half-duplex, block oriented</td>
</tr>
<tr>
<td>T=2</td>
<td>Asynchronous, full duplex, block oriented</td>
</tr>
<tr>
<td>T=3</td>
<td>Full duplex (not yet specified)</td>
</tr>
<tr>
<td>T=4</td>
<td>Asynchronous, half-duplex, byte oriented (extension of T=0, not yet specified)</td>
</tr>
<tr>
<td>T=5...T=13</td>
<td>RFU</td>
</tr>
<tr>
<td>T=14</td>
<td>For national use (not standardized in ISO)</td>
</tr>
<tr>
<td>T=15</td>
<td>RFU</td>
</tr>
</tbody>
</table>

in a later section. Finally, it describes a set of interindustry commands from the terminal to the smart card which are used for smart card management. These are the following [21]:

- READ BINARY
- WRITE BINARY
- UPDATE BINARY
- ERASE BINARY
- READ RECORD(S)
- WRITE RECORD
- APPEND RECORD
- UPDATE RECORD
- GET DATA
- PUT DATA
- SELECT FILE
- VERIFY
- INTERNAL AUTHENTICATE
- EXTERNAL AUTHENTICATE
- GET CHALLENGE
- MANAGE CHANNEL

We elaborate more on the ones that are important for the rest of the research in a later section. Finally ISO/IEC 7816-8 defines the commands specially designed for cryptographic algorithms [41].
4.2 The EMV specifications

EMV consists the technical standard which defines the specifications of the smart payment cards, payment terminals and Automatic Teller Machines (ATM’s) and goes under the name “Integrated Circuit Card Specifications for Payment Systems”. It is consisted of 4 books each of which is presented in this section. The EMV specification is expanded over 750 pages in total. It is self-evident that EMV is a complex protocol. The 4 EMV books along with a brief description are:

1. Book 1: Application Independent ICC to Terminal Interface Requirements. This book specifies the electrical and mechanical characteristics of the smart card and the terminal (in conjunction with ISO/IEC 7816-3 standard). It also defines the Answer To Reset (ATR) of the smart card to the reset command of the terminal. It specifies the transmission protocols $T = 0$ and $T = 1$ and finally it specifies the commands for the EMV application selection as well as their file structure [16].

2. Book 2: Security and Key Management. As its name implies, it specifies the security mechanisms of the protocol. These are the digital signatures, PIN encryption and secure messaging for integrity, authentication and confidentiality [17].

3. Book 3: Application Specification. This book specifies the data structures and commands for the implementation of the financial transactions [18].

4. Book 4: Cardholder, Attendant, and Acquirer Interface Requirements. It defines the specifications of the physical characteristics, interfaces and software architecture for acceptable terminal types [19].

4.3 APDU

APDU is the communication unit in EMV. We have two basic kinds of APDU, the C-APDU and the R-APDU. In this section we present their format and the fields they are consisted of as well as we introduce the most important C-APDUs. Figure 4.2 shows the basic transaction flow between the ICC and the terminal. The terminal always starts the transaction by sending a Reset. It does that by applying voltage at the RST contact of the smart card (see Figure 4.1). After that the smart card returns an ATR, the terminal issues the first C-APDU. The card then responds with the first R-APDU and so on.

4.3.1 C-APDU

Figure 4.3 displays the format of every C-APDU. The Header part is compulsory for every command. The Body part is optional.

- CLA: One byte field which specifies the class of instructions to which the command belongs.
- INS: One byte field which is the instruction code of the command within the class.
- P1 and P2: Parameters of the the instruction. These 2 bytes are instruction dependent.
- $L_c$ field: This one or three byte long field encodes the length of the Data field. If missing it means that there is no Data field.
- Data field: The byte string of length $L_c$ which represents the data of the command. An example of commands that need to send data is the SELECT FILE command which has to specify the Application ID in the Data field.
- $L_e$ field: This field (of variable length up to 3 bytes) contains the length of the expected data in the body part of the returned R-APDU.
Now we will analyze, in terms of structure and meaning, the most important C-APDUs. Their importance lies in the fact that they are used in the transaction profile we use for fuzzing (see Appendix B).

**Interindustry C-APDUs**

Interindustry C-APDUs are specified by the ISO/IEC 7816-4 standard and are application independent.

**SELECT FILE command.** The selection of a file allows to the terminal access to the logical structure of a card application hosted in this file. The C-APDU is dissected in Table 4.2. The ICC returns the File Control Information (FCI) of the application (Figure 4.4) which we refer to in Section 4.5 and marks the application as “selected”.

**READ RECORD command.** This command (Table 4.3) allows the terminal to read the public content of the file of an application. The data field of a R-APDU of a successful READ RECORD command contains the record of the application file requested form the terminal along with the Status Word 9000 (code for successful execution).

**VERIFY command.** This command (Table 4.4) instructs the ICC to compare the PIN typed by the cardholder with the one stored in the chip. The data field of this command contains the PIN either in plaintext or encrypted.
EMV specific C-APDUs

EMV specific C-APDUs are used solely in the EMV protocol.

GET PROCESSING OPTIONS command. The terminal, with this command (Table 4.5), informs the ICC that the processing of a new EMV transaction is about to begin [40].

GENERATE APPLICATION CRYPTOGRAM command. An EMV transaction can be on-line or off-line. In the first case the card needs to authenticate itself to the issuer (bank). In the second case the card needs to prove its participation in a certain transaction. The Application Cryptogram which is produced by the card after the issuance of the GENERATE APPLICATION CRYPTOGRAM command (Table 4.6) serves this dual purpose.

GET DATA command. The terminal issues this command (Table 4.7) in order to retrieve the PIN try counter. The PIN try counter indicates the remaining tries the cardholder has to enter the correct PIN.

4.3.2 R-APDU

The format of the R-APDU (see Figure 4.5) contains the Body part which is optional and the Trailer part which is compulsory.

- Data field: This byte string of length \( L_e \) contains the data requested by the terminal in the C-APDU. These data are called data elements and they are processed by the terminal in the terms of a financial transaction. The Data field of the R-APDU is a TLV data structure (see Section 4.4).

- SW1 and SW2: Status Words 1 and 2. Each status word is one byte long and they inform the terminal about the result of the previously issued C-APDU. Examples of status words are the 9000 which stands for Normal Processing, 61XX which informs the terminal that there are still XX bytes available and the 6CXX which informs the terminal that the correct length of the requested data is XX bytes.
CHAPTER 4. THE EMV PROTOCOL

Figure 4.4: FCI of an application (reproduced from [40])

Table 4.3: READ RECORD C-APDU

<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLA</td>
<td>00</td>
</tr>
<tr>
<td>INS</td>
<td>B2</td>
</tr>
<tr>
<td>P1</td>
<td>Record Number</td>
</tr>
<tr>
<td>P2</td>
<td>Reference control parameter</td>
</tr>
<tr>
<td>Lc</td>
<td>Not present</td>
</tr>
<tr>
<td>Data</td>
<td>Not present</td>
</tr>
<tr>
<td>Le</td>
<td>00</td>
</tr>
</tbody>
</table>

Table 4.4: VERIFY C-APDU

<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLA</td>
<td>00</td>
</tr>
<tr>
<td>INS</td>
<td>20</td>
</tr>
<tr>
<td>P1</td>
<td>00</td>
</tr>
</tbody>
</table>
| P2   | 80: plaintext PIN  
     | 88: enciphered PIN |
| Lc   | Variable depending on P2 |
| Data | Transaction PIN data |
| Le   | 00    |

Figure 4.5: R-APDU format

Body

[Data field]

Trailer

SW1 SW2

R-APDU

Figure 4.5: R-APDU format

Evaluation of the robustness of payment terminals with the use of fuzzing
Table 4.5: GET PROCESSING OPTIONS C-APDU

<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLA</td>
<td>80</td>
</tr>
<tr>
<td>INS</td>
<td>A8</td>
</tr>
<tr>
<td>P1</td>
<td>00</td>
</tr>
<tr>
<td>P2</td>
<td>00</td>
</tr>
<tr>
<td>(L_c)</td>
<td>Variable</td>
</tr>
<tr>
<td>Data</td>
<td>Tag'83'</td>
</tr>
<tr>
<td>(L_c)</td>
<td>00</td>
</tr>
</tbody>
</table>

Table 4.6: GENERATE APPLICATION CRYPTOGRAM C-APDU

<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLA</td>
<td>80</td>
</tr>
<tr>
<td>INS</td>
<td>AE</td>
</tr>
<tr>
<td>P1</td>
<td>Reference control parameter</td>
</tr>
<tr>
<td>P2</td>
<td>00</td>
</tr>
<tr>
<td>(L_c)</td>
<td>Variable</td>
</tr>
<tr>
<td>Data</td>
<td>Transaction related data</td>
</tr>
<tr>
<td>(L_c)</td>
<td>00</td>
</tr>
</tbody>
</table>

Table 4.7: GET DATA C-APDU

<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLA</td>
<td>00</td>
</tr>
<tr>
<td>INS</td>
<td>A4</td>
</tr>
<tr>
<td>P1</td>
<td>P2</td>
</tr>
<tr>
<td>9F17 (tag corresponding to the PIN try counter)</td>
<td></td>
</tr>
<tr>
<td>(L_c)</td>
<td>Not present</td>
</tr>
<tr>
<td>Data</td>
<td>Not present</td>
</tr>
<tr>
<td>(L_c)</td>
<td>00</td>
</tr>
</tbody>
</table>

Evaluation of the robustness of payment terminals with the use of fuzzing
4.4 TLV

"A data element is the smallest information unit that can be identified by a name, a description of its logical content, and a format. Data elements are mapped onto data objects, which are encoded according to a certain format (e.g. fixed length format, Basic Encoding Rules-Tag Length Value (BER-TLV) format, and others)" [41].

In this section we analyze the BER-TLV structure. Each data object consists of three fields: the Tag, the Length and the Value. When we refer to these fields we will use the names with the capital first letter. The Tag is the code which specifies a specific data element. The Length is the length of the Value. And the Value is the value of the data element. This structure is being used in order to maintain the interoperability among the payment applications (smart cards) from the different issuers (banks) and the payment terminal applications from the different vendors. Since every data element can be uniquely characterized by the Tag and the Length, EMV payment applications do not have to know the filesystem structure of every smart card application in order to retrieve the financial info.

There are two types of data objects. The first type is the simple data object and the second type is the constructed data object.

The simple data object is of the consists of only one Tag, Length and Value field. It does not encapsulate further any other data object. The Value field represents the value of the data element which corresponds to the Tag. The Length indicates the length of the Value. The encoding of the Length field is depicted in Table 4.8.

<table>
<thead>
<tr>
<th>1st byte</th>
<th>2nd byte</th>
<th>3rd byte</th>
<th>4th byte</th>
<th>5th byte</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 byte</td>
<td>'00' to 'FF'</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0 - 127</td>
</tr>
<tr>
<td>2 bytes</td>
<td>'81'</td>
<td>'00' to 'FF'</td>
<td>-</td>
<td>-</td>
<td>128 to 255</td>
</tr>
<tr>
<td>3 bytes</td>
<td>'82'</td>
<td>'00 00' to 'FF FF'</td>
<td>-</td>
<td>-</td>
<td>256 to 65535</td>
</tr>
<tr>
<td>4 bytes</td>
<td>'83'</td>
<td>'00 00 00' to 'FF FF FF'</td>
<td>-</td>
<td>-</td>
<td>65536 to 16777215</td>
</tr>
<tr>
<td>5 bytes</td>
<td>'84'</td>
<td>'00 00 00 00' to 'FF FF FF FF'</td>
<td>-</td>
<td>-</td>
<td>16777216 to 4294967295</td>
</tr>
</tbody>
</table>

The constructed data object recursively encapsulates other data objects. This means that the Value of a constructed data object might be other constructed data objects and/or simple data objects. The encapsulating rules (i.e. which data objects encapsulate which data object) are defined in terms of Tags. That is the EMV specification defines which Tag is encapsulated by which other Tag. The Length of a constructed data object is the length of the objects (constructed and/or simple) it further encapsulates. Figure 4.6 shows an example of the BER-TLV structure.
4.5 Data Elements

A data element is the smallest information unit. As we explain later, in Chapter 5, the data elements are fuzzed in several ways in order to deliver the fuzzing test cases to the SUT. In this section we refer to the data elements that we use for the fuzzing scenarios.

Table 4.9 lists the data elements along with their formats and lengths. The various formats are:

- The binary data elements consist of binary numbers (e.g. \(0x0013 = (19)_{10}\)) or bit combinations that are defined in the EMV specifications.

- The alphanumeric data elements (denoted as "an [\# of bytes]" in Table 4.9) contain a single character (0-9, a-z, A-Z) per byte.

- The numeric data elements (denoted as "n [\# of bytes]" in Table 4.9) consist of two numeric digits (0-9) per byte.

- The alphanumeric with a special character data elements (denoted as "ans" in Table 4.9) consist of a single character per byte.

For the purpose of this research the semantics of the data elements play a role in the interpretation of the results. By the time the SUT crashes after the delivery of a test case the semantics of the data element will indicate the source code fragment that is vulnerable.

Table 4.9 enumerates the data elements that are used in a specific EMV transaction (Appendix B). Some of the data elements are:
• Application Preferred Name (9f12): the name of the application that runs in the card.
• Application Version Number (9f08): the version of the application that runs in the card.
• Application Expiration Date (5f24): the date in which the application expires.
• Application PAN (5a): this is the identification number of the application (i.e. it is unique).
• Cardholder Name (5f20): the name of the cardholder as this is printed on the card.
• Application Label (50): mnemonic of the application associated with the Application Identifier (AID) displayed at the man-machine interface.
• Language Preference (5F2D): specifies the languages and their priority that will be used in the man-machine interface.
• Application Priority Indicator (87): indicates the priority of the various applications that exist in the smart card.
## Table 4.9: Data Elements [18]

<table>
<thead>
<tr>
<th>Tag</th>
<th>Name</th>
<th>Format</th>
<th>Length (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6F</td>
<td>File Control Information (FCI) Template</td>
<td>var.</td>
<td>var. up to 252</td>
</tr>
<tr>
<td>84</td>
<td>Dedicated File (DF) Name</td>
<td>binary</td>
<td>5-16</td>
</tr>
<tr>
<td>A5</td>
<td>FCI Proprietary Template</td>
<td>var.</td>
<td>var.</td>
</tr>
<tr>
<td>88</td>
<td>Short File Identifier (SFI)</td>
<td>binary</td>
<td>1</td>
</tr>
<tr>
<td>5F2D</td>
<td>Language Preference</td>
<td>an 2</td>
<td>2-8</td>
</tr>
<tr>
<td>9F11</td>
<td>Issuer Code Table Index</td>
<td>n 2</td>
<td>1</td>
</tr>
<tr>
<td>BF0C</td>
<td>FCI Issuer Discretionary Data</td>
<td>var.</td>
<td>var. up to 222</td>
</tr>
<tr>
<td>5F50</td>
<td>Issuer Uniform Resource Locator (URL)</td>
<td>ans</td>
<td>var.</td>
</tr>
<tr>
<td>70</td>
<td>READ RECORD Response Message Template</td>
<td>var.</td>
<td>var. up to 252</td>
</tr>
<tr>
<td>61</td>
<td>Application Template</td>
<td>binary</td>
<td>var. up to 252</td>
</tr>
<tr>
<td>4F</td>
<td>Application Dedicated File (ADF) Name</td>
<td>binary</td>
<td>5-16</td>
</tr>
<tr>
<td>50</td>
<td>Application Label</td>
<td>ans with the special character limited to space 1-16</td>
<td></td>
</tr>
<tr>
<td>87</td>
<td>Application Priority Indicator</td>
<td>binary</td>
<td>1</td>
</tr>
<tr>
<td>9F12</td>
<td>Application Preferred Name</td>
<td>ans</td>
<td>1-16</td>
</tr>
<tr>
<td>80</td>
<td>Response Message Template Format 1</td>
<td>var.</td>
<td>var.</td>
</tr>
<tr>
<td>57</td>
<td>Track 2 Equivalent Data</td>
<td>binary</td>
<td>var. up to 19</td>
</tr>
<tr>
<td>5F20</td>
<td>Cardholder Name</td>
<td>ans 2-26</td>
<td>2-26</td>
</tr>
<tr>
<td>9F1F</td>
<td>Track 1 Discretionary Data</td>
<td>ans</td>
<td>var.</td>
</tr>
<tr>
<td>90</td>
<td>Issuer Public Key Certificate</td>
<td>binary</td>
<td>144</td>
</tr>
<tr>
<td>8F</td>
<td>Certification Authority Public Key Index</td>
<td>binary</td>
<td>1</td>
</tr>
<tr>
<td>92</td>
<td>Issuer Public Key Remainder</td>
<td>binary</td>
<td>36</td>
</tr>
<tr>
<td>9F32</td>
<td>Issuer Public Key Exponent</td>
<td>binary</td>
<td>1 to 3</td>
</tr>
<tr>
<td>93</td>
<td>Signed Static Application Data</td>
<td>binary</td>
<td>144</td>
</tr>
<tr>
<td>5A</td>
<td>Application PAN</td>
<td>cn var. up to 19</td>
<td>var. up to 10</td>
</tr>
<tr>
<td>5F34</td>
<td>Application PAN sequence number</td>
<td>n 2</td>
<td>1</td>
</tr>
<tr>
<td>5F24</td>
<td>Application Expiration Date</td>
<td>n 6 (YYMMDD)</td>
<td>3</td>
</tr>
<tr>
<td>5F25</td>
<td>Application Effective Date</td>
<td>n 6 (YYMMDD)</td>
<td>3</td>
</tr>
<tr>
<td>5F28</td>
<td>Issuer Country Code</td>
<td>n 3</td>
<td>2</td>
</tr>
<tr>
<td>5F30</td>
<td>Service Code</td>
<td>n 3</td>
<td>2</td>
</tr>
<tr>
<td>9F07</td>
<td>Application Usage Control</td>
<td>binary</td>
<td>2</td>
</tr>
<tr>
<td>9F08</td>
<td>Application Version Number</td>
<td>binary</td>
<td>2</td>
</tr>
<tr>
<td>9F42</td>
<td>Application Currency Code</td>
<td>n 3</td>
<td>2</td>
</tr>
<tr>
<td>8C</td>
<td>Card Risk Management Data Object List (CDOL) 1</td>
<td>binary</td>
<td>var. up to 252</td>
</tr>
<tr>
<td>8D</td>
<td>CDOL 2</td>
<td>binary</td>
<td>var. up to 252</td>
</tr>
<tr>
<td>8E</td>
<td>Cardholder Verification Method (CVM) List</td>
<td>binary</td>
<td>10-252</td>
</tr>
<tr>
<td>9F0D</td>
<td>Issuer Action Code - Default</td>
<td>binary</td>
<td>5</td>
</tr>
<tr>
<td>9F0E</td>
<td>Issuer Action Code - Denial</td>
<td>binary</td>
<td>5</td>
</tr>
<tr>
<td>9F0F</td>
<td>Issuer Action Code - Online</td>
<td>binary</td>
<td>5</td>
</tr>
<tr>
<td>9F17</td>
<td>PIN try counter</td>
<td>binary</td>
<td>1</td>
</tr>
</tbody>
</table>

Evaluation of the robustness of payment terminals with the use of fuzzing 35
Chapter 5

Fuzzing Tool

In this chapter we describe the architecture of the fuzzing tool. First we describe our proposed methodology for fuzzing the ICC interface of the payment terminals. Secondly we describe the various test case scripts we implemented, thirdly we give a high-level overview of the main components of the tool and how they interact to each other and lastly we illustrate some of the implementation details.

5.1 General Overview

So far we have classified the various fuzzing methodologies and we have presented the EMV protocol which we wish to fuzz. The reason we do all this survey is to come up with an idea of how we can implement an EMV fuzzing tool. In this section we examine the fuzzing tool in terms of the 2 axis we presented in Chapter 2.

The fuzzing tool is categorized as a grey-box fuzzer. During our research we assumed no access to the source code of the SUT. This was done for 2 reasons. Firstly, we wanted the fuzzing tool to be able to fuzz every target that uses the EMV protocol. Building a fuzzing tool that would take into account the source code would result in a tool that would test only one specific EMV implementation. Secondly, the time given for this research would not allow us to build something that complex. Apart from these two reasons our access to the source code was restricted due to ownership constraints. The knowledge that we incorporated was the EMV protocol and ISO/IEC 7816 standard specifications.

The fuzzing tool is categorized as a grey-box fuzzer. During our research we assumed no access to the source code of the SUT. This was done for 2 reasons. Firstly, we wanted the fuzzing tool to be able to fuzz every target that uses the EMV protocol. Building a fuzzing tool that would take into account the source code would result in a tool that would test only one specific EMV implementation. Secondly, the time given for this research would not allow us to build something that complex. Apart from these two reasons our access to the source code was restricted due to ownership constraints. The knowledge that we incorporated was the EMV protocol and ISO/IEC 7816 standard specifications.

The fuzzing tool is categorized as a grey-box fuzzer. During our research we assumed no access to the source code of the SUT. This was done for 2 reasons. Firstly, we wanted the fuzzing tool to be able to fuzz every target that uses the EMV protocol. Building a fuzzing tool that would take into account the source code would result in a tool that would test only one specific EMV implementation. Secondly, the time given for this research would not allow us to build something that complex. Apart from these two reasons our access to the source code was restricted due to ownership constraints. The knowledge that we incorporated was the EMV protocol and ISO/IEC 7816 standard specifications.

The fuzzing tool is categorized as a grey-box fuzzer. During our research we assumed no access to the source code of the SUT. This was done for 2 reasons. Firstly, we wanted the fuzzing tool to be able to fuzz every target that uses the EMV protocol. Building a fuzzing tool that would take into account the source code would result in a tool that would test only one specific EMV implementation. Secondly, the time given for this research would not allow us to build something that complex. Apart from these two reasons our access to the source code was restricted due to ownership constraints. The knowledge that we incorporated was the EMV protocol and ISO/IEC 7816 standard specifications.

The fuzzing tool is categorized as a grey-box fuzzer. During our research we assumed no access to the source code of the SUT. This was done for 2 reasons. Firstly, we wanted the fuzzing tool to be able to fuzz every target that uses the EMV protocol. Building a fuzzing tool that would take into account the source code would result in a tool that would test only one specific EMV implementation. Secondly, the time given for this research would not allow us to build something that complex. Apart from these two reasons our access to the source code was restricted due to ownership constraints. The knowledge that we incorporated was the EMV protocol and ISO/IEC 7816 standard specifications.

The fuzzing tool is categorized as a grey-box fuzzer. During our research we assumed no access to the source code of the SUT. This was done for 2 reasons. Firstly, we wanted the fuzzing tool to be able to fuzz every target that uses the EMV protocol. Building a fuzzing tool that would take into account the source code would result in a tool that would test only one specific EMV implementation. Secondly, the time given for this research would not allow us to build something that complex. Apart from these two reasons our access to the source code was restricted due to ownership constraints. The knowledge that we incorporated was the EMV protocol and ISO/IEC 7816 standard specifications.

The fuzzing tool is categorized as a grey-box fuzzer. During our research we assumed no access to the source code of the SUT. This was done for 2 reasons. Firstly, we wanted the fuzzing tool to be able to fuzz every target that uses the EMV protocol. Building a fuzzing tool that would take into account the source code would result in a tool that would test only one specific EMV implementation. Secondly, the time given for this research would not allow us to build something that complex. Apart from these two reasons our access to the source code was restricted due to ownership constraints. The knowledge that we incorporated was the EMV protocol and ISO/IEC 7816 standard specifications.

The fuzzing tool is categorized as a grey-box fuzzer. During our research we assumed no access to the source code of the SUT. This was done for 2 reasons. Firstly, we wanted the fuzzing tool to be able to fuzz every target that uses the EMV protocol. Building a fuzzing tool that would take into account the source code would result in a tool that would test only one specific EMV implementation. Secondly, the time given for this research would not allow us to build something that complex. Apart from these two reasons our access to the source code was restricted due to ownership constraints. The knowledge that we incorporated was the EMV protocol and ISO/IEC 7816 standard specifications.

The fuzzing tool is categorized as a grey-box fuzzer. During our research we assumed no access to the source code of the SUT. This was done for 2 reasons. Firstly, we wanted the fuzzing tool to be able to fuzz every target that uses the EMV protocol. Building a fuzzing tool that would take into account the source code would result in a tool that would test only one specific EMV implementation. Secondly, the time given for this research would not allow us to build something that complex. Apart from these two reasons our access to the source code was restricted due to ownership constraints. The knowledge that we incorporated was the EMV protocol and ISO/IEC 7816 standard specifications.

The fuzzing tool is categorized as a grey-box fuzzer. During our research we assumed no access to the source code of the SUT. This was done for 2 reasons. Firstly, we wanted the fuzzing tool to be able to fuzz every target that uses the EMV protocol. Building a fuzzing tool that would take into account the source code would result in a tool that would test only one specific EMV implementation. Secondly, the time given for this research would not allow us to build something that complex. Apart from these two reasons our access to the source code was restricted due to ownership constraints. The knowledge that we incorporated was the EMV protocol and ISO/IEC 7816 standard specifications.

The fuzzing tool is categorized as a grey-box fuzzer. During our research we assumed no access to the source code of the SUT. This was done for 2 reasons. Firstly, we wanted the fuzzing tool to be able to fuzz every target that uses the EMV protocol. Building a fuzzing tool that would take into account the source code would result in a tool that would test only one specific EMV implementation. Secondly, the time given for this research would not allow us to build something that complex. Apart from these two reasons our access to the source code was restricted due to ownership constraints. The knowledge that we incorporated was the EMV protocol and ISO/IEC 7816 standard specifications.

The fuzzing tool is categorized as a grey-box fuzzer. During our research we assumed no access to the source code of the SUT. This was done for 2 reasons. Firstly, we wanted the fuzzing tool to be able to fuzz every target that uses the EMV protocol. Building a fuzzing tool that would take into account the source code would result in a tool that would test only one specific EMV implementation. Secondly, the time given for this research would not allow us to build something that complex. Apart from these two reasons our access to the source code was restricted due to ownership constraints. The knowledge that we incorporated was the EMV protocol and ISO/IEC 7816 standard specifications.
used a fuzzer which replays the transaction profile instead of an inline one. So that it can properly modify R-APDU1 after it has constructed the fuzzed R-APDU2.

![Diagram of the terminal and smart card interaction](image)

**Figure 5.1:** The terminal first asks for the size of the requested record or file. After the smart card informs it about the correct size it asks and receives the record or file.

The anomaly libraries of the fuzzer contain vectors which have fixed value and size known to cause problems to applications, vectors which mutate one or two bits of the valid data or vectors which are simply random data of a fixed length.

With all the above things in mind, Figure 5.2 presents the position of our proposed fuzzing EMV evaluation method among the ones presented earlier in Chapter 2.

The scenario we record and later replay (by fuzzing the R-APDUs) is the "offline plaintext PIN" transaction profile. We recorded it by using the Visa Smart Debit/Credit Acquirer Device Validation Toolkit Version 6.0.1. This toolkit contains smart cards programmed to exercise certain aspects of the payment terminal before their deployment. During the offline plaintext PIN scenario the payment terminal transmits the PIN to the smart card (not to the bank as in online) in plaintext (not encrypted) for authentication. Appendix A contains the transaction in compact form as it is described in Figure 5.1.
CHAPTER 5. FUZZING TOOL

5.2 Test Case Scripts

The tool makes use of a set of four scripts which produce the fuzzing test cases. These test cases can prove fatal if the SUT does not handle them properly.

5.2.1 Fuzz the value

In this script the tool fuzzes the value (but not the length) of the Value field in the TLV structure. The EMV specification states the length limit of the various data elements which participate in the financial transactions as well as their format (see Table 4.9). In this fuzzing script we keep the length of the fuzzed data in the anomaly library the same as the one in the original transaction profile but we change their value itself.

For example, there is a data element format called Numeric. The data elements of this format consist of 2 numeric digits (in the range 0x0 to 0x9) per byte. This means that every digit in the range 0xA to 0xF is excluded in a possible valid value. Other names for this data format are Binary Coded Decimal (BCD) or unsigned packed. An example which illustrates this case is the Issuer Country Code (Tag 5F28) which is specified as a data element of n 3 format (number 3 here indicates the number of nibbles) and of length 2 bytes. Valid Values of this data element would be "08 40" (Issuer Country Code for USA) or "03 00" (Issuer Country Code for Greece) among others. Invalid Values which contradict the specification in terms of format are for example "AB CD" or "03 01". The former is not of Numeric type and the latter does not exist in the ISO 3166 which specifies the Issuer Country Codes.

For the purposes of this fuzzing script the fuzzing tool does not have to update the various Length fields within a TLV structure since the length remains the same.
5.2.2 Fuzz the length

In this fuzzing script the tool replaces the valid R-APDU2 (see Figure 5.1) with a fuzzed one. The malicious data elements of the anomaly library here contain Values with lengths which go against the specifications. Thus the tool has to also calculate and inform the terminal for the length of the R-APDU2 via a valid R-APDU1. For this script, the fuzzing tool has to update every Length value that is dependent on the fuzzed Value so as for the new R-APDU2 to not be rejected due to inconsistencies in the TLV structure. For example in the specification is stated that the data element Issuer Country Code must be 2 bytes long. In the anomaly library there are values which go beyond this length such as 1, 3 or 10 bytes.

5.2.3 Fuzz the order

The anomaly library of this fuzzing script is the transaction profile itself. Here the fuzzing tool replaces every R-APDU from a predefined point in time on with a valid, still unexpected R-APDU. The other variant of this script does exactly the same apart from the fact that it replaces only one R-APDU in the whole transaction profile and keeps the others unchanged.

5.3 Fuzzing Tool Architecture

The fuzzing tool is consisted of 6 main components:

- the Transaction Profile
- the Normal Transaction script
- the Analyzer script
- the Template Match array
- the Anomaly Library and
- the Fuzzer.

The Transaction Profile, as its name implies, contains the C-APDU and R-APDU sequences which are required to complete the financial transaction. The Normal Transaction script replays the Transaction Profile as it is without changing it at all. The Analyzer script sniffs the transaction between the card and the terminal. By using these two scripts one can form the Transaction Profile that is compatible with each payment terminal. By saying that we mean that between the various Transaction Profiles of the various terminals there are minor differences which should be taken into account. After having the correct (compatible) Transaction Profile, the tester can use the Fuzzer in order to execute the fuzzing scripts. The Fuzzer takes as argument the fuzzed value from the Anomaly Library. It starts parsing the R-APDUs of the Transaction Profile while replaying it. As soon as it encounters the Tag of the Value about to be fuzzed it replaces it with the fuzzed Value and updates all the Length values which are dependent on it. In order to do that it uses the Template Match array. This array contains information about which Tag is encapsulated into which Tag. Figure 5.3 displays the interaction between these components.
Evaluation of the robustness of payment terminals with the use of fuzzing
5.4 Implementation Details

For the implementation of the fuzzing tool we used the Collis Conclusion Test Platform version 4.2.7.3 and for the tests we used the Collis Smartlink Box device. We will refer to the former as Conclusion and to the latter as SLBox. Conclusion uses the ETDL (Executable Test Description Language) in order for the tester to edit the test scripts. “ETDL is based on concepts from the Tree and Tabular Combined Notation or TTCN, as defined in the ISO/IEC 9694 International Standard, part 3 and Conformance Testing Theory” [from the Conclusion manual]. We run the Conclusion on a Windows XP machine with 2 GB of RAM and processing power of 1079 MHz. SLBox is a hardware platform for testing both smart card and payment terminals. It can be used for testing, intercepting, analyzing and modifying communication between a smart card and a terminal. Additionally it can be used as a smart card simulator or as a card reader.
Chapter 6

Testing Methodology

In this chapter we describe the developed testing methodology. Throughout this research we use the word tester as the user of the fuzzing tool.

6.1 Purpose of the testing methodology

The described testing methodology was developed for the purposes of the PCI PIN Transaction Security (PCI PTS) evaluations for payment terminals. PCI PTS requirements are all the physical and logical security features that the payment terminal should have in order to be considered safe for deployment.

PCI has recently integrated fuzzing as a testing procedure in order to assess the robustness of the payment terminals. The results of the fuzzing tests are further analyzed for possible vulnerabilities, they are communicated back to the vendor and finally to the PCI as evidences in order to justify the approval of the payment terminal's certification.

6.2 Usage of the fuzzing tool

As explained in Chapter 5 the fuzzer replays a recorded transaction profile. It might be the case that the transaction profiles from two payment terminals are different. EMV is designed to maintain the interoperability between the different vendors but still there might be minor differences. The tester has to make sure that the fuzzer contains the correct transaction profile. In order to do this she has to use the Analyzer script (see Section 5.3) to sniff a valid transaction between the terminal (SUT) and the smart card. The final stage of this phase is the plain replay of the transaction profile with the use of the fuzzer and the inspection of the log files for possible errors.

The next step is to perform the fuzzing by executing the three fuzzing scripts namely FuzzValue, FuzzLength and FuzzOrder the details of which were explained in Section 5.2. The tester executes the FuzzValue script first in order to test the SUT for vulnerabilities while delivering data elements the Value of which is fuzzed but the Length is the same as the valid one. The tester selects in the provided Graphical User Interface (GUI) the test cases she wants to execute (she has the option to select all or make a choice). Once the fuzzer starts the execution of a test case it instructs the tester to insert the probe smart card to the payment terminal. The terminal (if it has not crashed by the initial APDU exchange) prompts the tester to enter the transaction amount. The tester types the transaction amount and the terminal continues the transaction with the fuzzer. Once the terminal asks the tester to enter her PIN (always under the assumption that is has not crashed or rejected the transaction already) the tester does so.

At the end of the recorded transaction profile there are three possible outcomes. The payment terminal:
CHAPTER 6. TESTING METHODOLOGY

1. has accepted the transaction as valid and it displays a message like "Transaction Successful".
2. has rejected the transaction as invalid and it has stopped the communication immediately by displaying a message indication an error.
3. has rejected a R-APDU as invalid (which of course is) but still it keeps asking for the valid version of the R-APDU.

The fuzzer informs the tester that the transaction profile has come to an end so it prompts her to continue to the next test case. At the end of every test case execution the fuzzer also marks the test cases as fuzzed or not. It might be the case that because of hardware fault the fuzzer has not delivered the fuzzed data so the tester has to repeat this test case.

After the FuzzValue script the tester performs exactly the same sequence of actions for the FuzzLength script. The difference between the two test case scripts lies in the fact that the fuzzer has to update the Length fields of the fuzzed R-APDUs in the latter script.

Finally the tester completes the test cases by executing the FuzzOrder scripts. In this case the number of the test cases is the same as the number of “quartets” of APDUs as explained in Section 5.1.

6.3 Instrumentation and Monitoring of the SUT

Figure 6.1 displays the testing instrumentation. First the Collis Box acts as a plain sniffing tool in order to record the transaction profile of a Normal Card and the payment terminal. During the fuzzing execution the Collis Box transmits the APDUs between the Conclusion Software and the payment terminal. The tester has two sources of monitoring in order to determine if a test case has failed or not. The first is the log file of the Conclusion Software. This log file contains all the fuzzed communication between the fuzzer and the SUT. By observing this log file the tester can conclude whether the C-APDUs received by the terminal are the normal ones or contain errors which might have been caused by software failures.

The second source of monitoring is the observation of the payment terminal itself. The tester observes the device for crashes, hangs or display of unexpected information.

Evaluation of the robustness of payment terminals with the use of fuzzing
44 Evaluation of the robustness of payment terminals with the use of fuzzing
Chapter 7

Results

For the purposes of this research we tested the implemented fuzzing tool on 5 payment terminals from different vendors the anonymity of which is preserved throughout the rest of the study. All the devices we tested fall under the POS category as it is defined in Chapter 3. For every terminal 2 and half hours were spent in order to complete the fuzzing tests cases.

In this section the discovered problems are summarized by the terminal that showed the unexpected behavior. Some of the results can be used directly while others are vulnerabilities which can be exploited and require further research and enquiry.

1. Yellow POS:
   EMV specifies a data element called "Application Priority Indicator" and it has the tag 87 (see Table 4.9). This element helps the terminal to prioritize among the several card applications. It is 1 byte long and it takes values form 0 to 15 (bits 1 to 4). Bits 5 to 7 are RFU and bit 8 indicates that the cardholder should explicitly state her preference for this application when it is set. So when Application Priority Indicator has a value greater than 80 (1000 0000 in binary) the terminal should provide a prompt in order for the user to select the preferred application. This terminal displays the name of the application without any indication that it is a prompt like "SELECT APPLICATION" message. A possible exploitation would be for an attacker to craft a card and display a 32 byte message of her choice.

2. Orange POS:
   The EMV application (the application responsible for the EMV transactions) crashes during several test cases. To be more specific the application crashes (and in some cases after the PIN insertion) due to fuzzed data elements the length of which overexceedes the specifications. The EMV application becomes unresponsive unless the terminal reboots. If we assume an attacker who has access to the memory layout of the terminal (via a debugging tool) then she might be possible to exploit this vulnerability.
   Apart from the Denial of Service (DoS) attack this result is also interesting due to the fact that the application crashes while the PIN is in the memory. The remaining question is if the terminal fails securely (by deleting the contents of its memory) or not. In [31] the authors proved that Dynamic Random Access Memories (DRAMs) retain their contents for several seconds or minutes in room temperature. The memory remanence time increases as temperature drops. According to this fact an attacker can eject via a probe (in the form of a shim\(^1\)) the malicious vector during the transaction and cause the EMV application to crash. If we assume that the application fails insecurely the attacker can recover the PIN from the memory.

3. Pink POS:
   The terminal requests the PIN try counter from the fuzzer. The fuzzer returns a R-APDU

\(^1\)Thin flexible circuit board into the card slot, so that it lodges between the reader and the card's contact [25]
CHAPTER 7. RESULTS

Table 7.1: Number of normal transactions per terminal

<table>
<thead>
<tr>
<th>Terminal</th>
<th>number of normal transactions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yellow</td>
<td>58/174</td>
</tr>
<tr>
<td>Orange</td>
<td>129/174</td>
</tr>
<tr>
<td>Pink</td>
<td>110/174</td>
</tr>
<tr>
<td>White</td>
<td>131/174</td>
</tr>
<tr>
<td>Brown</td>
<td>128/174</td>
</tr>
</tbody>
</table>

other than the expected one but still valid. The terminal instead of aborting the transaction it considers it valid and ends it successfully. So the terminal validates a transaction without receiving the PIN try counter from the fuzzer.

4. White POS:

This terminal displays two kinds of unexpected behavior to the fuzz test cases the fuzzer delivers.

(a) As soon as the fuzzer crafts a R-APDU with an invalid PIN try counter and sends it to the terminal, the latter will inform the user that her PIN is incorrect. The strange fact in this behavior is that the user has not enter her PIN yet. The terminal will constantly display "Incorrect PIN. Try again?" every time the user presses "YES".

(b) In other test cases the EMV application crashes with the PIN in the memory and the terminal switches to the management pane. The EMV application is functional without rebooting the terminal but still the question of whether the terminal fails securely or not remains (with the same argumentation as in the case of the Orange payment terminal).

5. Brown POS:

For this terminal we did not have any findings.

6. Overall:

The number of normal transactions differs from terminal to terminal. By normal transaction we mean the transaction that the terminal would perform with a valid card which contain no errors. To sum up the normal transaction would contain the following actions: card insertion, amount insertion, PIN insertion, message that the transaction is successful. The error messages are not considered as a normal transaction but are considered as normal behavior that is why we do not count them. Table 7.1 sums up this observation. We count the number of the normal transactions out of a total of 174 test cases which is the number of the Fuzz value and Fuzz length test cases (we exclude the Fuzz order test cases since they are not the same for every terminal).
Chapter 8

Conclusions

This research was about fuzzing EMV protocol implementations. This was done by fuzzing the protocol from the ICC interface of the payment terminal. In order to do that we studied the EMV protocol and the ISO 7816 standard. We made an extensive research on the state of the art on the various fuzzing methodologies that exist. Finally we proposed an evaluation method and a proof of concept tool in order to test the robustness of the EMV implementations of the various payment terminals.

8.1 Summary and Conclusions

EMV is the protocol that regulates the financial transactions with the use of smart cards. It expands over a 700 pages document thus it can be considered as a fairly complex protocol to implement. In general one of the rules in computer security is that complexity is the enemy of security. It would be of no surprise if we encountered buggy EMV implementations due to this fact.

One way to estimate the robustness of systems is fuzz testing or fuzzing. Fuzzing delivers semi-valid data to the SUT in order to make it behave unexpectedly. Fuzzing was first introduced in 1988 (already 27 years old) and since then, and against the criticism for its “random” nature, it is a successful testing technique. There are a lot of different fuzzing “flavors” each of which fits better or worse to certain situations (e.g. absence of source code) or target types (e.g. remote targets).

We answered to the research question as posed in Section 1.2 and we implemented a fuzzing tool which replays and changes the transaction between the “probe” smart card and the ICC interface of the terminal. The transaction is modified according to an anomaly library in 3 different ways. The tool changes the values or the lengths of the data elements delivered to the SUT, or the order in which the expected R-APDUs are delivered.

The results obtained throughout this research can conclude the following points:

1. The number of ”normal transactions” (as they were defined in Chapter 7) with the exception of one terminal is high. From this we can conclude that not all the security checks are properly performed. An interesting study could be to check the specifications so as to determine the proper behavior of the terminal and then to compare these results with the ones in Table 7.1.

2. From the vulnerabilities we encountered we can say that the immediately exploitable are 2. The first is the fact that we can display a 32 byte string in the Yellow payment terminal and the other is the crash of the EMV application in the Orange and White payment terminals, which can be used as a DoS attack.

3. The unexpected behaviors of the terminals were caused by different anomaly library vectors. This means that different parts of the implementations are vulnerable in every terminal.
4. To the best of our knowledge (at least for payment terminals we tested we can say that for sure) fuzzing has not been applied to the ICC interface. This means that fuzzing can be fruitful when applied to SUT's that have not been fuzzed yet.

8.2 Future work

We tested the tool to payment terminals of different vendors in order to test its efficiency. But many questions are still unanswered regarding not only the results from our work but also the possibilities of fuzzing the specific SUT.

A possible extension of our work could be the further analysis and possible exploit of the vulnerabilities found. In order to do that we should find a better monitoring mechanism than the observation of the terminal’s display and the behavior of the terminal. So a possible research direction could be to analyze the terminal through either side channels (e.g. observation of other interfaces, power consumption etc.) or to find a way to monitor the terminal’s memory.

The fuzzing tool that we developed needs to be attended by the tester while performing the tests. It succeeds in reaching a certain level of automation but still the presence of the tester is required in order to facilitate the smart card’s mechanical movement and to observe the terminal’s behavior. A possible extension which would allow more test cases to be delivered would be to develop a platform which could automatically deliver the test cases and monitor the terminal in an instrumented manner.

The transaction profile recorded and fuzzed by the tool is the one of a specific transaction. The transaction is the offline plaintext. This means that the fuzzing tool exercises the functions related to this transaction. In order to test more functions and features of the SUT a proposed extension would be to record other transactions as well (e.g. online transaction with encrypted PIN) and fuzz them. That would require extension of the anomaly libraries as well as simulation of the cryptographic primitives (encryption etc.) of the smart card.

As we explain in Chapter 3, payment terminals are computers full of interfaces. Apart form the ICC interface we can think of ways to perform fuzzing to the other interfaces as well.

The fact that some of the malicious vectors in the anomaly library work efficiently (cause the unexpected behavior of the target) does not mean that others could not. An improvement could be to append malicious vectors to the anomaly library or to combine two or more malicious vectors to each other (i.e. to deliver two or more malicious vectors to the SUT). For this further study of how the different data elements interact with the payment terminal is required.

With this research we hope to make manufacturers and developers more aware of the fact that every interface, no matter if directly related to users, is vulnerable and it should be properly secured. Payment terminals are here to stay while the advent of the contactless payments with the use of contactless smart cards or mobile phones will give one more stepping stone to the ”bad” guys to reach their goals.
Evaluation of the robustness of payment terminals with the use of fuzzing
[23] Black Hat Conference, editor. Mission Mpossible. 3
[24] Black Hat Conference, editor. PINPADPWN. 1, 3
[26] emvco. worldwide emv deployment statistics, 2014. viii, 1, 2
[27] LLC EMVCo. Emv 4.1, june 2004. 23
[34] SRLabs (Security Research Labs). Payment terminals allow for remote pin capture and card cloning, 2012. 3
[38] Charlie Miller and Zachary NJ Peterson. Analysis of mutation and generation-based fuzzing. 2007. 7, 8
[40] Cristian Radu. Implementing electronic card payment systems. Artech House, 2002. viii, 1, 22, 23, 28, 29, 30, 32, 33

50 Evaluation of the robustness of payment terminals with the use of fuzzing


[45] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In Security and Privacy (SP), 2010 IEEE Symposium on, pages 317–331. IEEE, 2010. 10

[46] Rebecca Shapiro, Sergey Bratus, Edmond Rogers, and Sean Smith. Do-it-yourself scada vulnerability testing with lzfuzz. 11


[49] Ari Takanen, Jared D Demott, and Charles Miller. Fuzzing for software security testing and quality assurance. Artech House, 2008. 6, 7, 9, 11, 18


## Appendix A

### Offline Plaintext PIN transaction profile (compact)

<table>
<thead>
<tr>
<th>#</th>
<th>C-APDU1</th>
<th>R-APDU1</th>
<th>C-APDU2</th>
<th>R-APDU2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SELECT FILE</td>
<td>614B</td>
<td>GET RESPONSE of 75 bytes</td>
<td>response with length of 75 bytes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>75 bytes remaining</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>READ RECORD</td>
<td>6C2F</td>
<td>READ RECORD of 47 bytes</td>
<td>response with length of 47 bytes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>47 bytes remaining</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>READ RECORD</td>
<td>6A83</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td></td>
<td></td>
<td>record not found</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>SELECT FILE</td>
<td>6163</td>
<td>GET RESPONSE of 99 bytes</td>
<td>response with length of 99 bytes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>99 bytes remaining</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>GET PROCESSING OPTIONS</td>
<td>6110</td>
<td>GET RESPONSE of 16 bytes</td>
<td>response with length of 16 bytes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16 bytes remaining</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>READ RECORD</td>
<td>6C45</td>
<td>READ RECORD of 69 bytes</td>
<td>response with length of 69 bytes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>69 bytes remaining</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>READ RECORD</td>
<td>6C96</td>
<td>READ RECORD of 150 bytes</td>
<td>response with length of 150 bytes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>150 bytes remaining</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>READ RECORD</td>
<td>6C2F</td>
<td>READ RECORD of 47 bytes</td>
<td>response with length of 47 bytes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>47 bytes remaining</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>READ RECORD</td>
<td>6C96</td>
<td>READ RECORD of 150 bytes</td>
<td>response with length of 150 bytes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>150 bytes remaining</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>READ RECORD</td>
<td>6C10</td>
<td>READ RECORD of 16 bytes</td>
<td>response with length of 16 bytes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16 bytes remaining</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>READ RECORD</td>
<td>6C7F</td>
<td>READ RECORD of 127 bytes</td>
<td>response with length of 127 bytes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>127 bytes remaining</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>GET PROCESSING OPTIONS</td>
<td>6C04</td>
<td>GET PROCESSING OPTIONS of 4 bytes</td>
<td>response with length of 4 bytes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4 bytes remaining</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>VERIFY PIN</td>
<td>9000</td>
<td>Normal Processing</td>
<td>null</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>GENERATE APPLICATION CRYPTOGRAM</td>
<td>6124</td>
<td>GET RESPONSE of 36 bytes</td>
<td>response with length of 36 bytes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>36 bytes remaining</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>GENERATE APPLICATION CRYPTOGRAM</td>
<td>6124</td>
<td>GET RESPONSE of 20 bytes</td>
<td>response with length of 20 bytes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20 bytes remaining</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

52 Evaluation of the robustness of payment terminals with the use of fuzzing
Appendix B

Offline Plaintext PIN transaction profile (extended)

1. Terminal: 00a404000e315041592e5359532e4444463031 (SELECT FILE visa payment system environment)

2. Card: 614B (response is 75 bytes long)

3. Terminal: 00c000004b (GET RESPONSE of 75 bytes)

4. Card:
APPENDIX B. OFFLINE PLAINTEXT PIN TRANSACTION PROFILE (EXTENDED)

5. Terminal: 00b2010c00 (READ RECORD)

6. Card: 6c2f (response is 47 bytes long)

7. Terminal: 00b2010c2f (READ RECORD of 47 bytes)

8. Card:

---

54 Evaluation of the robustness of payment terminals with the use of fuzzing
APPENDIX B. OFFLINE PLAINTEXT PIN TRANSACTION PROFILE (EXTENDED)

T = 70
(READ RECORD Response Message Template)
L = 2d (45 bytes)

T = 61
(Application Template)
L = 2b (43 bytes)

T = 4f
(AID)
L = 07 (7 bytes)
V = a000000031010
T = 50
(Application Label)
L = 0b (11 bytes)
V = 564953412043524544954
T = 87
(Application Priority Indicator)
L = 01 (1 byte)
V = 01
T = 9f12
(Application Preferred Name)
L = 0f (15 bytes)
V = 4352...5341

SW = 9000

9. Terminal: 00b2020c00 (READ RECORD)

10. Card: 6a83 (Record not found)

11. Terminal: 00a4040007a00000031010 (SELECT FILE VISA DEBIT/CREDIT CLASSIC)

12. Card: 6163 (response is 99 bytes long)

13. Terminal: 00c0000063 (GET RESPONSE 99 BYTES LONG)

14. Card:

Evaluation of the robustness of payment terminals with the use of fuzzing 55
T = 6f
(FCI Template)
L = 61 (97 bytes)

T = 84
(DF Name)
L = 07 (7 bytes)
V = a0000000031010

T = a5
(FCI Priority Template)
L = 56 (86 bytes)

T = 50
Application Label
L = 0b (11 bytes)
V = 5649...4954

T = 5f2d
Language Preference
L = 02 (2 bytes)
V = 656e

T = 9f12
(Application Preferred Name)
L = 0f (15 bytes)
V = 4352...5341

T = bf0c
(FCI Issuer Discretionary Data)
L = 28 (40 bytes)

T = 5f50
Issuer URL
L = 25 (31 bytes)
V = 6874...3130

SW = 9000

15. Terminal: 80a80000028300 (GET PROCESSING OPTIONS)

16. Card: 6110 (response is 16 bytes long)

17. Terminal: 00c000010 (GET RESPONSE 16 BYTES LONG)

18. Card:
   T = 80 (Response Message Template Format 1)
   L = 0e (14 bytes)
   V = 5c00...0201
   SW = 9000

19. Terminal: 00b2010c00 (READ RECORD)

Evaluation of the robustness of payment terminals with the use of fuzzing
20. Card: 6c45 (response is 69 bytes long)

21. Terminal: 00b2010c45 (READ RECORD of 69 bytes)

22. Card:
   T = 70
   (READ RECORD Response Message Template)
   L = 43 (67 bytes)

   T = 57 (Track 2 Equivalent Data)
   L = 11 (17 bytes)
   V = 4761...5689
   T = 5f20 (Cardholder Name)
   L = 1a (26 bytes)
   V = 5649...3238
   T = 9f1f (Track 1 Discretionary Data)
   L = 10 (16 bytes)
   V = 3131...3030

   SW = 9000

23. Terminal: 00b2011400 (READ RECORD)

24. Card: 6c96 (response is 150 bytes long)

25. Terminal: 00b2011496 (READ RECORD of 150 bytes)

26. Card:
   T = 70
   (READ RECORD Response Message Template)
   L = 8193 (147 bytes)

   T = 90 (Issuer Public Key Certificate)
   L = 8190 (144 bytes)
   V = 8b39...0c02

   SW = 9000

27. Terminal: 00b2021400 (READ RECORD)

28. Card: 6c2f (response is 47 bytes long)

29. Terminal: 00b202142f (READ RECORD of 47 bytes)

30. Card:
   T = 70
   (READ RECORD Response Message Template)
   L = 2d (45 bytes)

   T = 8f
   (Certification Authority Public Key Index)
   L = 01 (1 byte)
   V = 95
   T = 92 (Issuer Public Key Remainder)
   L = 24 (36 bytes)
   V = 33f5...0103
   T = 9f32 (Issuer Public Key Exponent)
   L = 01 (1 byte)
   V = 03

   SW = 9000

31. Terminal: 00b2031400 (READ RECORD)
32. Card: 6c96 (response is 150 bytes long)

33. Terminal: 00b2031496 (READ RECORD of 150 bytes)

34. Card:
   T = 70
   (READ RECORD Response Message Template)
   L = 8193 (147 bytes)
   T = 93 (Signed Static Application Data)
   L = 8190 (144 bytes)
   V = 8c2b...29c4
   SW = 9000

35. Terminal: 00b2011c00 (READ RECORD)

36. Card: 6c10 (response is 16 bytes long)

37. Terminal: 00b2011c10 (RECORD NUMBER of 16 bytes)

38. Card:
   T = 70
   (READ RECORD Response Message Template)
   L = 0e (14 bytes)
   T = 5a (Application PAN)
   L = 08 (8 bytes)
   V = 4761739001010465
   T = 5f34 (Application PAN sequence number)
   L = 01 (1 byte)
   V = 01
   SW = 9000

39. Terminal: 00b2021c00 (READ RECORD)

40. Card: 6c7f (response is 127 bytes long)

41. Terminal: 00b2021c7f (READ RECORD of 127 bytes)

42. Card:

Evaluation of the robustness of payment terminals with the use of fuzzing
T = 70
(READ RECORD Response Message Template)
L = 7d (125 bytes)

T = 5f24 (Application Expiration Date)
L = 03 (3 bytes)
V = 251231
T = 5f25 (Application Effective Date)
L = 03 (3 bytes)
V = 090701
T = 5f28 (Issuer Country Code)
L = 02 (2 bytes)
V = 0840
T = 5f30 (Service Code)
L = 02 (2 bytes)
V = 0201
T = 9f07 (Application Usage Control)
L = 02 (2 bytes)
V = ff00
T = 9f08 (Application Version Number)
L = 02 (2 bytes)
V = 008c
T = 9f42 (Application Currency Code)
L = 02 (2 bytes)
V = 0840
T = 8c (CDOL 1)
L = 15 (21 bytes)
V = 9f02...3704
T = 8d (CDOL 2)
L = 17 (23 bytes)
V = 8a02...3704
T = 8e (CVM List)
L = 0e (14 bytes)
V = 0000000000000000010302031f00
T = 9f17 (PIN Try Counter)
L = 01 (1 byte)
V = 02
SW = 9000

43. Terminal: 80ca9f1700 (GET DATA)
44. Card: 6c04 (response is 4 bytes long)
45. Terminal: 80ca9f1704 (GET DATA of 4 bytes)
46. Card:
   T = 9f17 (PIN Try Counter)
   L = 01 (1 byte)
   V = 02
   SW = 9000
APPENDIX B. OFFLINE PLAINTEXT PIN TRANSACTION PROFILE (EXTENDED)

47. Terminal: 002000800824****ff...ff (VERIFY PIN)

48. Card: SW = 9000

49. Terminal: 80ae40001d0000000100000000000000084000000000008401503090020ace393
   (GENERATE APPLICATION CRYPTOGRAM)

50. Card: 6124 (response is 36 bytes long)

51. Terminal: 00c0000024 (GET RESPONSE of 36 bytes)

52. Card:
   \[ T = 80 \text{ (Response Message Template Format 1)} \]
   \[ L = 22 \text{ (34 bytes)} \]
   \[ V = 8001...1527 \]
   \[ SW = 9000 \]

53. Terminal: 80ae40001f593300000001000000000000000840000000000008401503090020ace393
   (GENERATE APPLICATION CRYPTOGRAM)

54. Card: 6114 (response is 20 bytes long)

55. Terminal: 00c0000014 (GET RESPONSE of 20 bytes)

56. Card:
   \[ T = 80 \text{ (Response Message Template Format 1)} \]
   \[ L = 12 \text{ (18 bytes)} \]
   \[ V = 4001...8000 \]
   \[ SW = 9000 \]