Experiment-Based Reverse Engineering of Signing Protocols for Smart Cards

Dennis Evtushenko, Stefan Genchev*
*Chair of Network Architectures and Services
School of Computation, Information and Technology, Technical University of Munich, Germany
Email: ge95ved@mytum.de, genchev@net.in.tum.de

Abstract—**Smart cards dominate the market in many different aspects, such as digital payment with a credit card or for any application for security, authentication or identification. In order for them to be used so widely, manufacturers offer interfaces to communicate with their respective cards even though they might implement completely different protocols for a variety of applications. These protocols can be public or proprietary. The main concern is a manufacturer decommissioning their product as part of the software development lifecycle, which might lead to electronic waste. This paper looks into designing a general setup and procedure to intercept and interpret the communication of any smart card implementing a PKCS#11 compatible library of a manufacturer. We introduce multiple tools for intercepting the traffic of these cards and present experiment-based strategies for reverse engineering their implementation details. Our paper could serve as a basis for drastically increasing the lifespan of smart cards. The presented strategies could then be used by the open source community to keep the cards updated, while also contributing to reducing the amount of unnecessary electronic waste.**

Index Terms—**smart card, integrated circuit card, pkcs#11**

1. Introduction

Smart cards are widely used. From being used for banking with a credit card all the way to healthcare, smart cards lay the foundation of our digital society. The ISO/IEC 7816 [1], [2] standard defines characteristics, like the commands, to avoid the issue of having incompatible Integrated Circuit Cards (ICC) across different countries and to enable interoperability in interindustry. The implementation of the protocol used by the card is not necessarily open source and can be proprietary. Therefore, the manufacturers offer an interface to be able to communicate with the card.

We follow the assumption of manufacturers complying with the product development lifecycle, where decommissioning is part of the cycle [3]. The fundamental issue of proprietary cards is the users being fully reliant on the manufacturer for updates. Once the manufacturer decommissions their product, it can lead to cards having no support for newer operating system versions, rendering them unusable without the required updates, and therefore turning them into electronic waste.

This paper looks into an experiment-based method for reverse engineering these naturally discontinued and outdated proprietary cards to make them open source, granted that the manufacturer does not already release the protocol of their card to the open source community voluntarily. The idea behind that is being able to extend the lifetime of these ICCs by having the possibility of updating their middleware.

The remainder of this paper has the following structure: First, in Section 2 we provide background information by outlining the necessary key concepts. In Section 3, we proceed with evaluating and explaining the different challenges and choices made for our approach. Section 4 presents the setup for reverse engineering and goes into detail for a variety of experiments to extract information. Lastly, we compare our paper to a related paper by A. Nicolić et al. [4] in Section 5 and then briefly summarize the most important findings in Section 6.

2. Background

In this section, we give an overview of the key concepts needed to understand the basics of the communication of smart cards.

2.1. PKCS#11

The Public-Key Cryptography Standard #11 defines a platform-independent Application Programming Interface (API), which is called "Cryptoki". The API specifies a set of functions to perform cryptographic operations. It is object-based and offers all the functionality to create, use, modify and delete cryptographic objects such as RSA key pairs, certificates and domain parameters for DSA or Diffie-Hellman. Cryptoki is essential for manufacturers because they can provide the API as a dynamically linked library for the C programming language to users, e.g., in the form of a `.dll` file. This way, the manufacturer can abstract the implementation details while still allowing to bridge the communication between the ICC and the client with the provided library. By replacing the `.dll` file, the manufacturer can update the implementation of their ICC while leaving the ICC's functionality unchanged.

Alongside cryptographic objects, the standard provides multiple relevant definitions for this paper. Tokens are defined as devices with the ability of executing cryptographic functions as well as being able to store these cryptographic objects. A Slot is a reader that can hold a Token. For example, a smart card is considered a Token, while a smart card reader is viewed as a Slot. These two can have a connection with each other, which is defined as a Session [5].

2.2. APDU

An Application Protocol Data Unit (APDU) is a data packet used for the communication between ICCs and a card reader. An APDU is a byte array containing information defined in the ISO/IEC 7816-4 standard [1].

TABLE 1: Command APDU

Field	Description	Length in bytes
CLA	Class of command	1
INS	Instruction code	1
P1-P2	Parameters	2
L_c	Length of Command data	0, 1 or 3
Command data	Command data	Variable length
L_e	Max. length of Response data	0, 1, 2 or 3

TABLE 2: Response APDU

Field	Description	Length in bytes
Response data	Response data	Variable, at most L_e
SW1-SW2	Status bytes	2

There are command and response APDUs. Their format is shown in Tables 1 and 2, respectively. The first is responsible for sending the necessary data of the operation, which entails the instruction, parameters and more as described in Table 1. Some operations do not require any parameters or data. The latter returns the response to the host machine. This response contains two status-bytes SW1-SW2 indicating whether the command has been successful or not. Depending on the operation, this response can contain optional data. For reference, Figure 2 in Section 4.2.1 shows an example for command and response APDUs. Each command is strictly defined with its own corresponding sections for data in the byte array. For example, NIST SP 800-73 Pt.2-5 PIV [6] has its own set of public commands, defining the byte values for the APDUs for each command, which still follow the ISO/IEC 7816 standard. On the other hand, there can be manufacturers keeping these specifications private [1], [7].

3. Analysis

The problem statement is to reverse engineer as many details of the signing protocol of ICCs as possible. For example, this includes all of the supported mechanisms and parameters for each ICC as well as the corresponding mapping to the byte values in the command and response APDUs. The difficulty of reverse engineering lies in finding a starting point and commonalities within the protocols. Manufacturer-independent interfaces such as PKCS#11 and Minidriver [8] can solve these problems. They define the functionality as well as mechanisms used for cryptographic functions, as described in Section 2.1. Additionally, they offer a selection of variable parameters that can be modified and used to extract information out of the protocols, which is further explained in Section 3.3. The structure provided by the manufacturer independent interfaces is what makes the problem statement approachable. This lays the foundation for the different experiment-based approaches for reverse engineering the structure of commands presented in this paper.

3.1. PKCS#11 and Smart Card Minidriver

Smart card minidrivers [8] are interfaces, similar to PKCS#11, that can be written by smart card manufacturers. They are exclusive to Microsoft, which is a disadvantage in comparison to PKCS#11 because not all manufacturers or providers support it. Additionally, there is no option to debug the communication on Linux. In comparison to PKCS#11, minidriver offers a less clear command mapping. In conclusion, the minidriver is less flexible and more difficult to reverse engineer. For these reasons, we exclusively cover ICCs with an existing implementation of PKCS#11 in the context of this paper.

3.2. Virtual Card Reader and Logging

The process of reverse engineering in the context of this paper involves intercepting the data sent between the card reader and the ICC. We identified two viable possibilities to intercept and view this data, which are the virtual card reader and logging.

A virtual card reader implements a driver interface of a card reader with no underlying hardware. This software can only relay the data to a card reader, resulting in the possibility of intercepting the sent data. More specifically, this can then be used to view and interpret the data, which consists of command and response APDUs. An example for an implementation of a virtual card reader is *vpcd* [9]. This approach would be compatible with the existing PKCS#11-compatible software, therefore not requiring a test driver.

Logging, on the other hand, is intercepting the data between the card and the card reader using software such as *pcscd* on Linux or *APDUPlay* on Windows. Therefore, both Windows and Linux are viable options for this approach.

Both presented options for intercepting are suitable, however, in the context of this paper, we decided to use logging. On Linux, logging APDUs can be done with *pcscd*, which is a Personal Computer/Smart Card Daemon, using the `--apdu` and `--debug` options [10]. An alternative to pcscd is *APDUPlay* for Windows [11].

Since the PKCS#11 API is provided in C, the code for the experiments is also written in C. A different option is to use a wrapper such as *TUGraz IAIK* [12] in order to be able to use the manufacturers library in a different programming language such as Java or Python. However, using a wrapper is only preference and not necessary since the functionality stays the same.

3.3. Approach to Interpret Commands

In this section, we give an overview of selected PKCS#11 functionality, which serves as a basis on how to:

- Find functions that are able to be reverse engineered.

- Find modifiable parameters within these functions.
- Choose fitting and relevant values for these parameters in order to extract information.
- Interpret the data of these commands.
- Design and conduct further experiments for targeted information disclosure.

Firstly, we need to choose a command or a series of commands to send to a card. The APDUs have different fields where we can input different parameters.

Since the ICC follows the protocol of a vendor specific library implementing the PKCS#11 standard, there is a given list of all possible parameters for all functions defined in the standard. In order to reverse engineer the implementation of the ICC, all possible combinations of the functions and parameters have to be tried. The goal is to find out all of the supported functionalities as well as the mappings between them and their respective bytes.

`C_GenerateKeyPair` generates new private and public key objects. It is possible to choose the mechanism, such as RSA, and create a template for the respective keys, where the attributes can be further specified. When considering RSA for example, the private or public exponent as well as the key length (modulus bits) can be set to their desired values. Additionally, the attributes can specify what each key will be used for: The private key can be set to be used for signing, while the public key can be set to be used for verifying. This is intended for preventing improper uses. Furthermore, the amount of attributes for each template is a parameter of the function, which can be changed for different key pairs.

Similarly to `C_GenerateKeyPair`, `C_GenerateKey` generates a secret key for, e.g., AES or a set of domain parameters for, e.g., Diffie-Hellman. Analogously to the generation of key pairs, the mechanism, the attributes and the amount of the attributes can be specified. Some examples for attributes in this context are specifying the uses for keys to encryption or decryption as well as the key length and key type.

There are cryptographic functions defined by the PKCS#11 standard, such as `C_Encrypt`, `C_Decrypt`, `C_Digest`, `C_Sign` and `C_Verify`. Despite having different functionalities, these functions share many similarities by being split into multiple function calls and because of their parameters. Each function can be suffixed with `Init`, to initialize the respective operation, such as `C_EncryptInit`. This step is required and speficies the mechanism, such as `RSA` or `Elliptic Curve Cryptography (ECC)` for the `C_Sign` operation as well as the respective key for the selected operation. Afterwards, the operation itself is called where the data and the location of the output with their respective lengths can be defined [5].

PKCS#11 also defines multiple hashing and signing mechanisms, where the padding can be changed, such as `CKM_SHA256_RSA_PKCS`, performing SHA-256 hashing and RSA signing with PKCS#1 v1.5 padding or `CKM_SHA256_RSA_PKCS_PSS` with PSS padding [13].

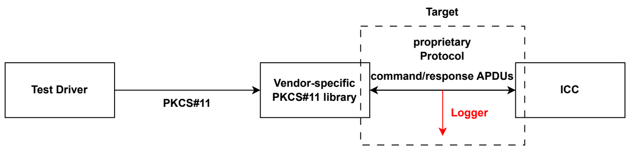
All of these various parameters for each command can then be used to interpret the bytes of the command APDUs as well as the response APDUs. Since we know exactly which parameters we put into the function we can specifically look for the changing bytes in the log. Considering there are many different options, the approach yielding the most consistent results is to only change one parameter value at a time. Upon only changing a single parameter value, most of the bytes within the APDU stay the same, while specific values will differ with every change of the respective parameter, narrowing down the location of the bytes of the input parameter over multiple function calls. This limitation allows for a more precise interpretation of these values and mapping them to their respective algorithm in both the command and response APDU.

4. Design

In this section, we present a specific soft- and hardware setup to be able to conduct experiments. These experiments are designed to extract information out of the proprietary implementation details of a card with a corresponding vendor library implementing the PKCS#11 standard.

4.1. Software and Hardware Setup

Figure 1: Experimental Setup



The experiments are conducted as shown in Figure 1. Firstly, a card with an instance of the PKCS#11 standard, such as PIV [6] or ISO/IEC 7816-15 [14] is needed. Additionally, a card reader is required to be able to communicate with the card. This reader is connected to a machine, where the raw communication between the card and reader is intercepted and logged. An application interacting with the PKCS#11 library on the computer is the client, which is acting as a test driver in this case. This test driver is responsible for the control and management of the experiment. The test driver is a program calling a series of selected commands for the current experiment in order to communicate with the card. These commands should be appropriately selected to disclose the details of the routines of the card as well as the concrete values of the APDU fields, as discussed in Section 3.3.

4.2. Experimental Approach

The following experiments are designed by the authors for possible approaches for the problem statement. All experiments are conducted using the soft- and hardware setup described in Section 4.1. Each experiment should be designed in a way such that a single operation with all of the associated functionality can be disclosed. For example, this would include not only encryption but also the initialization of the operation as well as the creation of the respective key with specified attributes. This potentially leads to a lot of data in the form of multiple APDUs, that has to be analyzed. Therefore, it can be effective to keep the same environment, such as keys or data for multiple iterations as well as to minimize the changes of parameters. In the following, we present step by step approaches to extract information of the implementation of any card using the PKCS#11 protocol.

4.2.1. Password Encoding. The ISO/IEC 7816-15 [14] standard defines attributes for passwords, which are also used in ICCs following protocols of vendor specific libraries implementing the PKCS#11 standard. It defines five different possible encodings for passwords: `binary coded decimal`, `half-nibble binary coded decimal`, `ascii-numeric`, `utf8` and `iso9564-1`. Since passwords can have variable lengths, padding can be necessary. PKCS#11 allows password values to contain any valid UTF-8 character, which can be restricted to a subset depending on the Token. In this context, we assume that passwords can only hold numerical values with six characters.

The `VERIFY` command, which can be used to translate the `C_Login` command, is defined in ISO 7816-4 [1]. It is more like a recommendation for the structure and values of the APDUs. Therefore, manufacturers can make their implementations completely different. This fact makes reverse engineering more difficult but not impossible.

There are two relevant PKCS#11 functions for this experiment: `C_OpenSession` and `C_Login`. The first one is responsible for creating a Session between the Token and the Slot. The latter is for sending the password to log the user into the Token [5].

The login allows for a limited amount of attemps to enter the correct password before locking the user out of the Token. Once locked out, the supervisor can unlock the card again. Since we are working on a proprietary card, we do not require access to the supervisor actions. Therefore, we need to send different passwords while also avoiding locking the card in order to find the encoding. For this experiment, we choose an arbitrary password such as `123456` and call `C_Login`.

Figure 2: Possible APDUs for C_Login

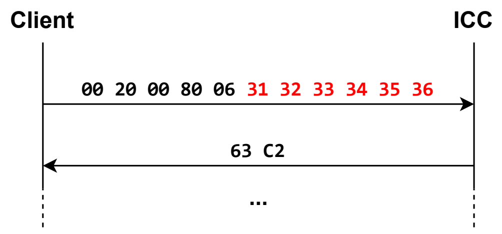


Figure 2 shows possible command and response APDUs for the password `123456`. In this case it is trivial to find the password within the bytes, which is the last 6 bytes of the command APDU marked in red. The password in this example is encoded as `ascii-numeric`. If it is not clear where the password is located on the first attempt, we can modify a single digit, e.g., `123455` and call `C_Login` once more. The resulting bytes can now be compared again. For more data, we create the initial situation, meaning the correct password has to be entered, to reset the password retry counter and `C_Logout` has to be called, to log the user out of the Token. Afterwards, the process can be repeated again while changing a single digit of the password. With enough collected data, the sent password's byte values can now be mapped to the password encoding used by this card.

4.2.2. C_Sign. The goal for this experiment is to iterate through all combinations of mechanisms that are compatible with a fixed key pair. In order to compute signatures, we have to generate a key pair and provide data that should be signed. For this experiment, the data can be fixed in the beginning, since there are no further implementation details that can be disclosed with more data sets. The next step is to choose a key type, such as RSA or ECC, for which we can generate a private key and public key pair using `C_GenerateKeyPair`. However, before calling the function, we define the template for the key pair. When considering RSA for example, this can include setting attributes such as the private and public exponent to valid arbitrary values, e.g., 3 as well as the key length to, e.g., 2048 or 4096. With the specified parameters, we can now create a Session using `C_OpenSession`, followed by logging into the Token with `C_Login`, if required, and call the `C_GenerateKeyPair` function.

The relevant parameters for the signature function are the mechanism, the signature key and the data. The generated key pair can now be used for multiple signatures. For each generated key pair from the previous step, we iterate through all possible mechanisms such as `CKM_SHA256_RSA_PKCS_PSS`, `CKM_SHA384_RSA_PKCS` and `CKM_ECDSA`, where the scheme, digest and padding are dynamic. Upon specifying all of the needed parameters, the signature can be done using `C_SignInit` and `C_Sign` and we can proceed with creating a new key pair and repeating the steps. For the purpose of reverse engineering the signature operation itself, it is sufficient to generate a fixed key pair for each iteration through the mechanisms. For more details about the structure and values of the attributes for keys, there should be a new key pair generated for every single possibility of valid options provided by the PKCS#11 standard.

Since cards implementing a PKCS#11 compatible library do not have to support all mechanisms defined by that standard, the return value can be checked to see if the selected mechanism is supported by the card [5], [13]. Granted it is supported, we gain some information about the implementation with this approach. We can inspect the APDUs to find out the mapping for each mechanism in the command and response APDUs. This process can be continued, by trying all possibilities for mechanisms and attributes during the key creation. However, the changes each iteration should be kept to a minimum to maximize the amount of information that can be confidently mapped.

5. Related Work

The research presented in the paper of A. Nikolić et al. [4], describes tools and strategies for reverse engineering smart card middleware of proprietary manufacturers. In contrast to this paper, their work focusses on the middleware to gain more information about the Windows smart card architecture as opposed to our research, where we present tools and techniques for analyzing smart cards implementing PKCS#11 specifically. Nikolić et al. show multiple different approaches such as performing static analysis through disassembly, dynamic analysis and the analysis of the communication traffic, consisting of command and response APDUs, as described in this paper. It should be noted, however, that their research is limited to smart card middleware following the Windows Smart Card Minidriver Specification. Their paper successfully applies their described methods to the Serbian Electronic ID Card, where a platform-independent library has been developed to allow for use of this card on other operating systems.

6. Conclusion

This paper addresses the problem of manufacturers decommissioning their products without publishing their implementation for possible updates. In our research, we discuss multiple tools to intercept the flow of communication between smart cards and card readers. Additionally, we highlight the challenges of reverse engineering smart cards and their solutions through manufacturer independent libraries such as PKCS#11. Most importantly, our research introduces multiple step by step strategies to extract information about the implementation details of smart cards. This paper can serve as a basis for reducing the amount of electronic waste by enabling the possiblity of updates for the middleware.

Future research may include the reverse engineering of smart cards following different standards, using similar concepts as described in this paper. Furthermore, there is the possiblity to automate parts of the experiments by creating a program that can automatically generate all possible parameter combinations for the operations.

References

[1] ISO/IEC 7816-4:2020(E), “Identification cards — integrated circuit cards — part 4: Organization, security and commands for interchange,” International Organization for Standardization, Geneva, CH, Standard, May 2020.

[2] ISO/IEC 7816-8:2021(E), “Identification cards — integrated circuit cards — part 8: Commands and mechanisms for security operations,” International Organization for Standardization, Geneva, CH, Standard, August 2021.

[3] SAND2015-9022, “Model of the product development lifecycle,” Sandia National Laboratories, Albuquerque, New Mexico, Report, September 2015.

[4] A. Nikolić, G. Sladić, and B. Milosavljević, “Reverse engineering smart card middleware,” Novi Sad, Serbia, 2013.

[5] PKCS11-base-v2.40, “Pkcs #11 cryptographic token interface base specification version 2.40,” Edited by S. Gleeson and C. Zimmerman, OASIS Standard, April 2015, http://docs.oasis-open.org/pkcs11/pkcs11-base/v2.40/os/pkcs11-base-v2.40-os.html.

[6] H. Ferraiolo, K. Mehta, S. Francomacaro, R. Chandramouli, and S. Gupta, “Interfaces for personal identity verification: Part 2 – piv card application card command interface,” National Institute of Standards and Technology, Gaithersburg, MD, NIST Special Publication (SP) NIST SP 800-73pt2-5, 2024, https://doi.org/10.6028/NIST.SP.800-73pt2-5.

[7] Yubico, “Apdus,” https://docs.yubico.com/yesdk/users-manual/yubikey-reference/apdu.html, n.y., [Online; accessed 21-December-2024].

[8] “Smart card minidriver specification, v7.07,” https://learn.microsoft.com/en-us/previous-versions/windows/hardware/design/dn631754(v=vs.85), Microsoft Corporation, 2017, [Online; accessed 16-December-2024].

[9] F. Morgner, “Virtual smart card,” https://frankmorgner.github.io/vsmartcard/virtualsmartcard/README.html, 2024, [Online; accessed 17-December-2024].

[10] D. Corcoran and L. Rousseau, “pcscd - pc/sc smart card daemon,” https://linux.die.net/man/8/pcscd, n.y., [Online; accessed 12-December-2024].

[11] P. Svenda, “Pc/sc apdu inspection and manipulation tool (apduplay),” https://github.com/crocs-muni/APDUPlay, 2019, [Online; accessed 12-December-2024].

[12] “Iaik pkcs#11 wrapper 1.6.9 api,” https://javadoc.sic.tech/pkcs11_wrapper/current/index.html, IAIK at Graz University of Technology, 2023, [Online; accessed 21-December-2024].

[13] PKCS11-curr-v2.40, “Pkcs #11 cryptographic token interface current mechanisms specification version 2.40,” Edited by S. Gleeson and C. Zimmerman, OASIS Committee Specification 01, September 2014, https://docs.oasis-open.org/pkcs11/pkcs11-curr/v2.40/cs01/pkcs11-curr-v2.40-cs01.html. Latest version: http://docs.oasis-open.org/pkcs11/pkcs11-curr/v2.40/pkcs11-curr-v2.40.html.

[14] ISO/IEC 7816-15:2016(E), “Identification cards — integrated circuit cards — part 15: Cryptographic information application,” International Organization for Standardization, Geneva, CH, Standard, May 2016.